Tìm hiểu về LINQ

Mục lục

[I. Giới thiệu về LINQ: 4](#_Toc8588579)

[II. Giới thiệu vài nét về quá trình hình thành và phát triển của LINQ: 4](#_Toc8588580)

[II.1. Những ý tưởng ban đầu với ODBC, DAO, RDO và ODBCDirect: 4](#_Toc8588581)

[II.2. Thành phần hóa quá trình truy cập dữ liệu (OLE DB): 5](#_Toc8588582)

[II.3. .NET Framework và C# ra đời, thống nhất các components và API vào ADO: 5](#_Toc8588583)

[II.4. Lập trình cho mô hình quan niệm (ADO.NET Entities và LINQ): 6](#_Toc8588584)

[II.5. Quá trình phát triển của LINQ: 7](#_Toc8588585)

[III. Những khái niệm cấu thành nên LINQ: 8](#_Toc8588586)

[III.1. Phương thức mở rộng - Extension method: 8](#_Toc8588587)

[III.2. Gọi lại(Callback) và Ủy thác(Delegate) (Func, Predicate): 8](#_Toc8588588)

[III.3. Biểu thức Lambda - Lambda Expression: 10](#_Toc8588589)

[III.4. Mẫu - Generic: 13](#_Toc8588590)

[III.5. IEnumerable, IQueryable và Yield: 14](#_Toc8588591)

[IV. Cấu trúc và đặc tính của LINQ: 17](#_Toc8588592)

[IV.1. Cấu trúc cơ bản của LINQ và biểu thức LINQ: 17](#_Toc8588593)

[IV.1.1. Ba phần của một biểu thức LINQ: 19](#_Toc8588594)

[IV.1.2. Các dữ liệu nguồn: 21](#_Toc8588595)

[IV.1.3. Truy vấn và thực thi truy vấn: 21](#_Toc8588596)

[IV.1.4. Các biến IEnumerable có trong các câu truy vấn LINQ: 22](#_Toc8588597)

[IV.2. Các hoạt động truy vấn cơ bản: 22](#_Toc8588598)

[IV.2.1. Truy xuất dữ liệu nguồn: 22](#_Toc8588599)

[IV.2.2. Lọc (Filtering): 22](#_Toc8588600)

[IV.2.3. Sắp xếp (Ordering): 23](#_Toc8588601)

[IV.2.4. Gom nhóm (Grouping): 23](#_Toc8588602)

[IV.2.5. Kết (Joining): 24](#_Toc8588603)

[IV.2.6. Lựa chọn (Selection/Projection): 25](#_Toc8588604)

[IV.3. Chuyển đổi dữ liệu với LINQ: 26](#_Toc8588605)

[IV.4. Cú pháp truy vấn (Query Syntax) và cú pháp phương thức (Method Syntax): 26](#_Toc8588606)

[IV.5. LINQ và Kiểu mẫu (Generic Type): 27](#_Toc8588607)

[IV.6. Mô hình đối tượng quan hệ (Object Relational Mapping) và thiết kế đối tượng quan hệ (Object Relational Designer): 27](#_Toc8588608)

[IV.6.1. Mô hình đối tượng quan hệ (ORM): 27](#_Toc8588609)

[IV.6.2. Thiết kế đối tượng quan hệ (O/R desgner): 28](#_Toc8588610)

[IV.6.3. Các đặc tính của Thiết Kế Đối Tượng Quan Hệ: 30](#_Toc8588611)

[V. LINQ to SQL: 31](#_Toc8588612)

[V.1. Các Khái niệm trong LINQ to SQL: 32](#_Toc8588613)

[V.1.1. Mô hình hóa CSDL dùng LINQ to SQL(dùng thiết kế đối tượng quan hệ): 32](#_Toc8588614)

[V.1.2. Lớp DataContext 33](#_Toc8588615)

[V.1.3. Các câu lệnh làm việc trong LINQ to SQL: 34](#_Toc8588616)

[V.2. Xây dựng chương trình với LINQ to SQL: 35](#_Toc8588617)

[V.2.1. Định nghĩa các lớp mô hình dữ liệu: 35](#_Toc8588618)

[V.2.2. Truy vấn Cơ sở dữ liệu: 37](#_Toc8588619)

[V.2.3. Cập nhật Cơ sở dữ liệu: 37](#_Toc8588620)

[V.2.4. Lấy dữ liệu dùng Stored Procedure: 37](#_Toc8588621)

[V.2.5. Cập nhật dữ liệu dùng Stored Procedure: 37](#_Toc8588622)

[V.2.6. Thực thi các biểu thức SQL tùy biến: 37](#_Toc8588623)

[V.3. Tổng kết: 37](#_Toc8588624)

[V.4. Giao dịch: 38](#_Toc8588625)

[V.5. Lệnh gọi SQL trực tiếp: 38](#_Toc8588626)

[V.6. Các tham số: 39](#_Toc8588627)

[V.7. Cách kết nối một CSDL (LINQ to SQL): 39](#_Toc8588628)

[V.8. Cách tự động tạo ra một CSDL (LINQ to SQL): 40](#_Toc8588629)

[V.9. Bạn có thể làm gì với LINQ to SQL: 41](#_Toc8588630)

[V.9.1. Lựa chọn (Selecting): 41](#_Toc8588631)

[V.9.2. Thêm (Inserting): 42](#_Toc8588632)

[V.9.3. Cập nhật (Updating): 42](#_Toc8588633)

[V.9.4. Xóa (Deleting): 43](#_Toc8588634)

[V.10. Thủ tục lưu trữ (Stored Procedures): 43](#_Toc8588635)

[V.10.1. Trả về RowSets: 43](#_Toc8588636)

[V.10.2. Sử dụng Stored Procedures với tham số đầu vào: 44](#_Toc8588637)

[V.10.3. Sử dụng Stored Procedure ánh xạ cho nhiều dạng kết quả: 45](#_Toc8588638)

[VI. LINQ to Object: 46](#_Toc8588639)

[VI.1. Các khái niệm trong LINQ to Object: 46](#_Toc8588640)

[VI.1.1. Object: 46](#_Toc8588641)

[VI.2. Xây dựng chương trình với LINQ to Object: 47](#_Toc8588642)

[VI.2.1. LINQ to String: 47](#_Toc8588643)

[VI.2.2. : 47](#_Toc8588644)

[VI.3. Tổng kết: 47](#_Toc8588645)

[VII. Tổng quan về LinQ to XML: 47](#_Toc8588646)

[VIII. LINQ to XML so với DOM (C #): 51](#_Toc8588647)

[IX. LINQ to XML so với các công nghệ XML khác: 54](#_Toc8588648)

[X. Tài liệu tham khảo: 56](#_Toc8588649)

# Giới thiệu về LINQ:

LINQ là viết tắt của từ Language – Integrated Query tạm dịch là truy vấn tích hợp ngôn ngữ là một sự nâng cấp lớn trong C#3.0(phiên bản C# đã được Microsoft giới thiệu cùng với Visual Studio 2008 và .NET Framework 3.5), là cầu nối giữa thế giới của các đối tượng(objects) với thế giới của dữ liệu(data).

Theo truyền thống, các câu truy vấn dữ liệu được thể hiện một cách đơn giản giống như các chuỗi kí tự mà không cần đến kiểm tra kiểu tại thời điểm biên dịch hoặc sự tham gia của trình hỗ trợ trực quan. Hơn nữa bạn cần phải tìm hiểu một ngôn ngữ truy vấn riêng cho mỗi loại dữ liệu nguồn khác nhau như: Cở sở dữ liệu SQL, tài liệu XML, các dịch vụ Web(Web services). LINQ là truy vấn *first-class* xây dựng cho ngôn ngữ C# và Visual Basic. Bạn viết một câu truy vấn dựa trên tập hợp các đối tượng bằng cách sử dụng ngôn ngữ, các từ khóa và các toán tử quen thuộc.



# Giới thiệu vài nét về quá trình hình thành và phát triển của LINQ:

## Những ý tưởng ban đầu với ODBC, DAO, RDO và ODBCDirect:

Trong những năm đầu thập niên 90 của thế kỉ 20, Microsoft đã nghĩ đến việc xây dựng một API có thể giúp lập trình viên truy cập đến bất kì CSDL nào. Năm 1992, họ đã thành công khi xây dựng nên ODBC(Open Database Connectivity) để truy vấn tới cơ sở dữ liệu quan hệ SQL. Và sau đó là ODBC ISAM(Indexed Sequential Access Method) với khả năng làm việc với các CSDL không hỗ trợ truy vấn(XML) hay CSDL không tích hợp quan hệ(text, excel, dBase, Paradox files)

Cũng trong thời gian này, các khái niệm như Data Access Object (DAO) và Remote Data Objects (RDO) cũng lần lượt ra đời tạo tiền đề cho Microsoft xây dựng nên ODBCDirect: một chế độ cho phép DAO có thể tham gia truy vấn độc lập vào CSDL sử dụng ODBC

## Thành phần hóa quá trình truy cập dữ liệu (OLE DB):

Trong giai đoạn này, mục tiêu của các kĩ sư Microsoft là xây dựng nên các component truy cập dữ liệu có thể tái sử dụng và và phù hợp với kiến trúc thành phần COM(Component Object Model), và OLE DB(Object Linking and Embedding Database) ra đời.

Là một giao thức *first-class* dựa trên COM để làm việc với dữ liệu, OLE DB được xây dựng để hỗ trợ cho các khái niệm về CSDL thành phần hóa(Federated - Componentized) Database. Mục tiêu của bộ thư viện này là tạo nên một cấu trúc cơ sở chung “Rowset” có thể sử dụng để trình bày các bộ dữ liệu(các hàng) bất kể dữ liệu nguồn là gì và chúng đại diện cho cái gì: một kết quả truy vấn, bảng hoặc giao diện, ISAM, text, excel, dữ liệu dạng sổ, email, directory services,... Trên nền tảng của Rowset, chúng ta có thể sử dụng các QueryInterface của COM để xác định bất cứ khi nào các chức năng bổ sung(như cuộn, cập nhật, hoặc điều hướng theo chỉ mục) được hỗ trợ.

Với cách biểu diễn dữ liệu này, Microsoft có thể xây dựng các component chung để làm việc với các kiểu dữ liệu, như một công cụ nhập chung, công cụ kiểm duyệt chung, công cụ truy vấn chung trên cả CSDL quan hệ và CSDL không quan hệ.

Cũng giống như ODBC, OLE DB cũng là một inteface “Cấp thấp” phải sử dụng con trỏ, quản lý bộ nhớ rõ ràng cà cần kiểm soát vòng đời. Và cũng giống như ODBC, nó yêu cầu nhiều gói API riêng biệt để có thể hoạt động với các ngôn ngữ tự động như VB. Vì thế các kĩ sư Microsoft cần tạo ra một định nghĩa mới để sử dụng cho bộ thư viện này, họ đã đảo vị trí các kí tự của DAO và cuối cùng lựa chọn cái tên ADO(ActiveX Data Objects).

ADO áp dụng cùng mô hình kết nối/lệnh/bản ghi của DAO, nhưng sử dụng một công nghệ con trỏ cục bộ tiên tiến hơn của FOX database team. “bản ghi ngoại tuyến”(Disconnected Recordset) của ADO sử dụng các con trỏ chỉ chuyển tiếp qua một luồng kết quả từ CSDL và các thuộc tính trên bản ghi để xác định xem dữ liệu có cuộn được hay cập nhật được hay không, cách li các dữ liệu từ kết quả và các dữ liệu từ kho dữ liệu. Bản ghi ngoại tuyến sử dụng dữ liệu từ truy vấn ban đầu để tạo các lệnh Insert, up date, delete và truyền chúng trở lại kho dữ liệu.

## .NET Framework và C# ra đời, thống nhất các components và API vào ADO:

Tháng 8 năm 1998, một nhóm các kĩ sư Microsoft đã tổ chức một buổi gặp mặt tại Semiahmoo, WA và thống nhất tạo nên BOLT(Beachside Offsite for Lightning Technology). Từ “Lightning” được sử dụng thay thế cho “COM3” và cuối cùng trở thành .NET Framework.

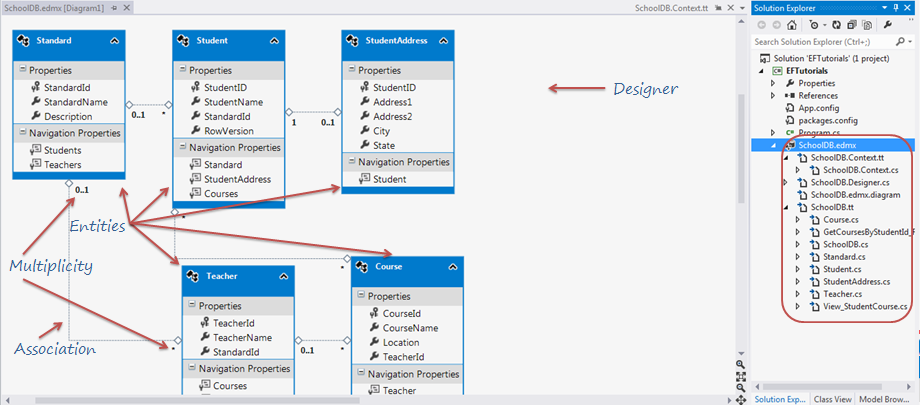
Trong giai đoạn này, các kĩ sư Microsoft đã tìm cách kế thừa nhiều nhất có thể những kiến trúc từ ADO/OLE DB và tái tạo thành ADO.NET. Cùng với sự phổ biến của Internet, disconected programming đã dần trở nên quan trọng và trở nên thông dụng hơn, XML trong giai đoạn này cũng trở thành ngôn ngữ phổ biến để làm việc với dữ liệu. ADO.NET giải quyết những thách thức mới này bằng cách phân tách rõ ràng truy vấn dữ liệu có kết nối(thông qua nhà cung cấp dữ liệu xác định “Data Provider”) với truy vấn dữ liệu không kết nối(thông qua bộ nhớ dữ liệu dùng chung “Dataset”) cùng với một cơ chế rõ ràng và có thể mở rộng “DataAdapter” để kết nối cả hai.

Nhờ thành công từ sức mạnh của sự tách biệt kết nối và không kết nối, việc kế thừa hợp lý từ ADO/OLE DB, và một sự tiến bộ lớn về nền tảng gốc với RAD(rapid application development), COM/DCOM, một .NET Framework gắn kết,.. ADO.NET đã nhận được nhiều sự hưởng ứng từ giới lập trình viên

## Lập trình cho mô hình quan niệm (ADO.NET Entities và LINQ):

1. ADO.NET Entities:

ADO.NET Entities được xây dựng bằng cách thêm khả năng để viết các ứng dụng dựa trên mô hình dữ liệu thực thể(Entity Data Model - EDM). Mô hình dữ liệu thực thể mở rộng mô hình dữ liệu quan hệ với cấu trúc quan hệ thực thể (Entity Relationship - ER) để mô hình hóa quan niệm thế giới thực như kế thừa(nhân khẩu thường trú và nhân khẩu tạm trú là các nhân khẩu), quan hệ(nhân khẩu có tiểu sử) và các thành phần phức tạp( địa chỉ một nhân khẩu bao gồm số nhà, tên xã phường, quận huyện, tỉnh thành phố). Một cấu trúc SQL mở rộng gọi là “Entity SQL” cho phép bạn truy vấn trực tiếp lược đồ quan hệ của mình, tận dụng sự kế thừa, truy cập các thành phần phức tạp và điều hướng các mối quan hệ. Trong nhiều trường hợp, việc xây dựng các khái niệm này thành lược đồ khái niệm và ngôn ngữ truy vấn sẽ giảm bớt sự cần thiết của các phép nối(join), liên hiệp(union), và các truy vấn con phức tạp để đơn giản hóa các truy vấn.



Các lược đồ khái niệm này được thể hiện và truy vấn thông qua một “Entity Client”. Entity Client này là một ADO.NET Data Provider xây dựng các câu truy vấn hướng tới nhà cung cấp dữ liệu nhất định theo hướng nhìn đọc/ghi của client. Các truy vấn và cập nhật được viết dựa trên các khung nhìn khái niệm này được mở rộng bởi Entity Client và được thực hiện dưới dạng các truy vấn của các nhà cung cấp dữ liệu nhất định. Tất cả việc thực hiện truy vấn thực tế được thực hiện trong store(không phải trong client) và các kết quả được tập hợp thành các kết thừa, đa hình, các kết quả với các thành phần lồng nhau và kết hợp. Sự tách biệt này giữa mô hình khái niệm mà ứng dụng hướng đến và lược đồ lưu trữ của CSDL là một khái niệm vô cùng mạnh mẽ mà các kĩ sư Microsoft tin rằng sẽ đơn giản hóa quá trình xây dựng và bảo trì các ứng dụng CSDL.

Trình diễn hướng nhìn Client thông qua ADO.NET Data Provider cho phép Microsoft giữ lại mô hình lập trình ADO.NET, tận dụng các đặc tính ở code, tools và hiểu biết đã có trước đó về ADO.NET. Trên thực tế, Entity Client sử dụng các ADO.NET Data Provider sẵn có (và mở rộng để phục vụ các tiểu chuẩn truy vấn biểu diễn hình cây mới) xây dựng dựa trên một cộng đồng luôn ủng hộ và liên tục tạo các tùy chỉnh cho ADO.NET Data Provider.

1. Đối tượng/Quan hệ (Object/Relational):

Kể từ phiên bản 1.0 của .NET Framework, người dùng đã liên tục tìm kiếm về một giải pháp đối tượng/quan hệ. Microsoft đã có một số nỗ lực tạo ra một số giải pháp O/R cho .NET, đáng kể nhất là “ObjectSpaces” mặc dù không được pháp hành tuy nhiên cũng được công bố dưới dạng “technical preview” tại PDC lúc họ giới thiệu về .NET Framework.

Tương tự hầu hết các giải pháp O/R hiện nay, ObjectSpaces đã cố gắng hỗ trợ một tập ánh xạ và kịch bản phong phú thông qua việc tạo các truy vấn tùy chỉnh. Thêm hỗ trợ cho một kiểu ánh xa kế thừa mới, với ý nghĩa là thêm code cho trình tạo truy vấn để chèn các điều kiện nối (join conditions) cần thiết vào các vị trí phù hợp. Dễ thấy rằng cấu trúc này sẽ khiến cho sự tương tác với các phép nối(join), phép chiếu(projection), liên kết(unions) và vị từ(predicates) được thêm vào trong suốt truy vấn để mô hình hóa cho các cấu trúc tương tự đối tượng khác làm cho code trở nên phức tạp và dễ vỡ hơn. Cố gắng để nắm bắt và cập nhật đối với các truy vấn phức tạp như vậy vừa khó khăn lại vừa dễ khiến cho nó trở nên phức tạp hơn nữa. Tệ nhất ở phương pháp này đó là sự khó khăn trong xác minh truy vấn tạo ra dựa trên các kết hợp này là đúng hay chưa.

ADO.NET Entities có một cách tiếp cận khác. Bằng cách mô hình hóa lược đồ khái niệm thông qua phía client thực hiện truy vấn và cập nhât khung nhìn, Entity Client tận dụng sự đầu tư quan trọng và nghiên cứu đã được thực hiện để tạo ra lý thuyết CSDL quan hệ. Chẳng hạn, việc cập nhật được thực hiện bằng cách áp dụng các kỹ thuật bảo trì khung nhìn để cập nhật khung nhìn để tạo ra một tập hợp các biểu thức delta được kết hợp với các chế độ truy vấn để tạo các biểu thức cập nhật. Kết quả của các khung nhìn truy vấn là xử lý cập nhật đều có thê kết hợp và kiểm chứng.

1. LINQ:

LINQ là viết tắt của cụm từ Language Intergrated Query, LINQ thống nhất các khái niệm truy vấn vào các ngôn ngữ lập trình, cho phép sử dụng code để truy vấn dữ liệu được kiểm tra bởi trình biên dịch và các công cụ dành cho nhà phát triển giúp cho lập trình viên viết các câu truy vấn dễ dàng hơn. Điều này, cùng với các mô hình khái niệm cấp cao hơn(như Entities), góp phần giảm bớt tính không tương đồng giữa ứng dụng và dữ liệu.

LINQ được hỗ trợ như một thành phần first-class trong ADO.NET Entity Framework thông qua “LINQ to Entities”. LINQ to Entities là một phần của lớp đối tượng dịch vụ cho phép người dùng viết các truy vấn thông qua các biểu thức ngôn ngữ và các truy vấn tích hợp trong C# và VB như các câu truy vấn SQL. Điều này có nghĩa rằng các khái niệm khung nhìn client tương tự như các mẫu code của ADO.NET hiện có hoặc thông qua việc sử dụng đối tượng dịch vụ bằng cách sử dụng các truy vấn văn bản đặc biệt hoặc truy vấn tích hợp ngôn ngữ.

## Quá trình phát triển của LINQ:

Trong các phiên bản cập nhật kể từ .NET Framework 3.5, Microsoft không trực tiếp cập nhật vào LINQ, họ thường cập nhật các library liên quan tác động đến LINQ, điều này giúp thay đổi và cải thiện LINQ

# Những khái niệm cấu thành nên LINQ:

LINQ mở rộng ngôn ngữ C# bằng cách thêm các biểu thức truy vấn gần giống với các câu lệnh SQL và có thể sử dụng được để trích xuất và xử lý dữ liệu một cách thuận tiện từ các mảng(arrays), các lớp Enumerable, tài liệu XML, các CSDL quan hệ, và các CSDL của bên thứ ba. LINQ còn có thể dùng các biểu thức truy vấn như các câu lệnh chính thức trong bộ thư viện .NET Framework giúp cho việc truy vấn dữ liệu được thực hiện trực tiếp trên code một cách đễ dàng và linh động. Nó cũng định nghĩa một tập hợp các tên phương thức(được gọi là toán tử truy vấn tiêu chuẩn hoặc toán tử chuỗi tiêu chuẩn), cùng với các quy tắc dịch giúp trình biên dịch dịch các biểu thức truy vấn một cách trôi chảy thành các truy vấn trên các tên phương thức đã định nghĩa này, các biểu thức lambda và các kiểu ẩn danh(anonymous types). LINQ có được các khả năng này nhờ được tổng hợp từ nhiều khái niệm hay ho và phức tạp của ngôn ngữ C#, những khái niệm đó bao gồm:

## Phương thức mở rộng - Extension method:

Trong một số trường hợp của việc lập trình, ta muốn thêm method(phương thức) vào cho một class đã được niêm phong(Sealed Class), hoặc class từ các library(thư viện) khác. Với một số ngôn ngữ, điều này là không thể thực hiện được, nhưng với C#, chúng ta có thể dùng Extension method.

Xét ví dụ sau, chúng ta có class Student từ một library khác, không thể sửa code, ta muốn thêm method Print:

public class Student

{

    public string Name { get; set; }

    public int Age { get; set; }

}

Chúng ta tạo 1 extenstion class, class này phải là static class, method cũng phải static, params đầu tiên truyền vào là class cần extention, với từ khóa this.

public static class StudentExtension

{

   public static void Print(this Student student)

   {

      Console.WriteLine(student.ToString());

   }

}

//Sử dụng

var student = new Student();

student.Print();

## Gọi lại(Callback) và Ủy thác(Delegate) (Func, Predicate):

1. Nhắc lại về khái niệm Callback:

Callback la truyền một đoạn code (hàm A) vào một đoạn code khác (hàm B). tới một thời điểm nào đó của hàm B, hàm A này sẽ được gọi tới(Callback)

Ví dụ về Callback trong javascript:

function tangQua(qua) {

   return console.log("Đã tặng " + qua);

}

function oNha(vo, tangQua){

   var qua = "Quà đã nhận";

   tangQua(qua);

}

1. Delegate:

Để sử dụng được Callback thì hàm cần được gọi tới như một param tức là hàm đó cần được lưu như một object và có kiểu dữ liệu trả về. Thử viết lại hàm trên ở C# ta dễ dàng nhận thấy lỗi xảy ra khi tangQua không có kiểu dữ liệu trả về:

public void tangQua(string qua) {

   Console.Write("Da tang " + qua);

}

 //Kiểu dữ liệu cho params "vo" là Person, kiểu dữ liệu cho tangQua là gì???

public void oNha(Person vo, tangQua){

   var qua = "Quà đã nhận";

   tangQua(qua);

}

Ở đây, để truyền function tangQua vào, ta phải sử dụng kiểu dữ liệu Delegate (một Con trỏ hàm - ). Ta khai báo delegate theo cú pháp sau:

public delegate void TangQuaDelegate(string qua);

//delage + kiểu trả về (void) + tên delegate + (tham số truyền vào)

public void tangQua(string qua) {

   Console.Write("Da tang " + qua);

}

public void oNha(Person vo, TangQuaDelegate tangQua)

{

   var qua = "Quà đã nhận";

   tangQua(qua);

}

1. Action, Predicate, Func trong C#:

Action, Predicate, Func(APF) là một cách đơn giản hơn để khai báo delegate:

* Action: Action<T in1, T in2, …>. Action tương đương 1 delegate với kiểu trả về là void, với in1, in2 là các params nhận vào.
* Predicate: Predicate<T in>. Predicate tương đương 1 delegate với kiểu trả về là bool, với in là các param nhận vào. Predicate chỉ có thể nhận vào 1 param duy nhất.
* Func: Func<T in1, T in2, … , T result>. Function tương đương 1 delegate với kiểu trả về do ta khai báo (result), in1, in2 là các params nhận vào. Func bắt buộc phải trả ra giá trị, không thể trả void.

Để dễ hiểu, chhãy tham khảo bảng sau. Đây là bảng so sánh các khai báo bằng delegate, cùng với cách khai báo tương ứng bằng Action, Predicate, Func:

|  |  |  |  |
| --- | --- | --- | --- |
| **DELEGATE** | **ACTION** | **PREDICATE** | **FUNC** |
| delegate void VoidDelegate(int input1, bool input2) | Action<int, bool> |  |  |
| delegate bool BoolDelegate(int input1) |  | Predicate<int> | Func<int, bool> |
| delegate int intDelegate(bool input2) |  |  | Func<bool, int> |
| delegate void HelloWorldDelegate() | Action |  |  |
| delegate bool HelloWorldBoolDelegate() |  | Predicate | Func<bool> |

Đoạn code đầu có thể được viết lại ngắn gọn như sau:

public void tangQua(string qua) {

   Console.Write("Da tang " + qua);

}

public void oNha(Person vo, Action tangQua)

{

   var qua = "Quà đã nhận";

   tangQua(qua);

}

APF kết hợp với Lambda expression và một vài thư viện khác đã góp phần tạo nên sự mạnh mẽ của LINQ

## Biểu thức Lambda - Lambda Expression:

Từ .NET 2.0, Microsoft thêm vào một khái niệm đó là Anonymous Function để hỗ trợ cho cách viết Delegate:

public delegate void TangQuaDelegate(string qua);

TangQuaDelegate dlg =

          delegate(string qua) { Console.WriteLine("Tang quà" + qua); };

Nhận thấy sự rườm rà vẫn còn tồn tại trong cách viết này, sang đến phiên bản .NET 3.0 Microsoft cho ra mắt Lambda Expression. Ta cố thể hiểu Lambda Expression là một cách viết Anonymous Function ngắn gọn hơn:

//Cách cũ

TangQuaDelegate dlg = delegate(string qua) { Console.WriteLine("Tặng quà" + qua); };

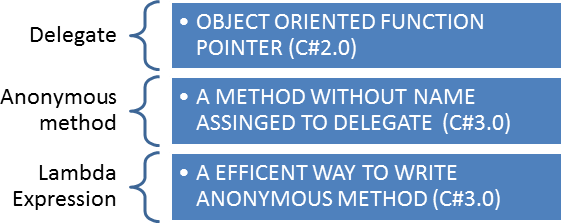
//Dùng lambda expression

TangQuaDelegate lamdaDlg = (qua) => { Console.WriteLine("Tặng quà: " + qua); }

//Câu lệnh đầy đủ của lambda expression.

//Dấu "=>" gọi là go-to

(parameters) => { statement }



Dưới đây là một số quy tắc viết Lambda Expression:

**//1. Có thể bỏ qua kiểu dữ liệu của parameter truyền vào**

(string qua) => {Console.WriteLine("Tặng quà: " + qua);}

(qua) => {Console.WriteLine("Tặng quà: " + qua);}

**//2. Nếu không có parameter, bỏ dấu () trống**

() => {Console.WriteLine("Hello");}

**//3. Nếu chỉ có 1 parameter, có thể bỏ luôn dấu ()**

(x) => {Console.WriteLine("Hello " + x);}

x => {Console.WriteLine("Hello " + x);}

**//4. Nếu có nhiều parameter, ngăn cách bằng dấu phẩy**

(x, y) => {Console.WriteLine("Hello " + x + y);}

**//5. Nếu anonymous function chỉ có 1 câu lệnh, có thể bỏ dấu {}**

x => { Console.WriteLine("Hello " + x); }

x => Console.WriteLine("Hello " + x)

**//6. Nếu chỉ return 1 giá trị, có thể bỏ chữ return.**

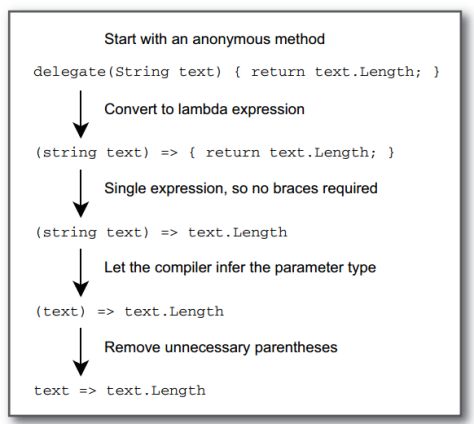
//4 lambda expression sau tương đương nhau

(x) => { return x > 4; }

x => { return x > 4; }

x => return x > 4

x => x > 4



*Các bước để rút gọn một Anonymous Function thành một Lambda Expression*

Lambda Expression kết hợp với APF tạo nên các hàm LINQ(Where(), Select(), First(),…). Ta sẽ xét ví dụ sau về hàm Where() để hiểu rõ hơn:

var studentList = new List<Student>();

//Sử dụng lambda expression

var students = studentList.Where(stu => stu.Age > 20);

//Nếu không có Lambda Expression, ta phải sử dụng Anonymous Function

var student = studentList.Where(new delegate(Student stu) { return stu.Age > 20; });

//Hoặc tệ hơn

public bool FindStudentWithAge(Student stu) { return stu.Age > 20; }

var student = studentList.Where(FindStudentWithAge);

Ở Ví dụ trên các tham số đầu vào của hàm Where là một Delegate Func, Lambda Expression cho phép chúng ta khai báo một cách ngắn gọn và dễ hiểu Func này giúp câu truy vấn LINQ trở nên ngắn gọn hơn.

## Mẫu - Generic:

Generic tham gia vào 50% những dòng code LINQ mà chúng ta viết

1. Generic là gì:

Rất nhiều người trong chúng ta sử dụng Generic mà không biết rằng đó là Generic, xét ví dụ về tạo một danh sách các học sinh như sau:

//Một list chứa các object là Student

List<Student> students = new List<Student>();

students.Add(new Student()); //Code đúng

students.Add(new Car()); //Compile lỗi

//Lấy học sinh đầu tiên.

//Compiler tự hiểu kết quả là Student

Student first = students.First();

Generic trong ví dụ trên chính là phần trong 2 dấu ngoặc nhọn <>

Để dễ hiểu, ta hãy quay lại thòi .NET 1.0, khi Generic chưa xuất hiện:

//Không có generic

//Một list chứa các object

List students = new List();

students.Add(new Student()); //Compile bình thường

students.Add(new Car()); //Compile bình thường

//Lấy object đầu tiên, phải ép kiểu sang Student

Student first = (Student)students.First();

Không có generic, compiler không thể check lỗi lúc compile. Do đó, ta có thể thêm 1 object Car và list gồm các object Student. Khi lấy 1 phần tử ra, ta cũng phải ép kiểu, vì compiler chỉ hiểu nó là 1 object. Vì những lý do đó, generic đã được thêm vào ở .NET 2.0. Tác dụng của generic:

* Giúp tái sử dụng code. Ví dụ: Ta chỉ cần viết class List<T>, T ở đây có thể là bất kì class gì.
* Hỗ trợ compiler bắt lỗi trong quá trình compiler (Hạn chế được tình trạng như dòng new Car()).
* Không còn phải ép kiểu từ object.

1. Sử dụng Generic:

Generic được ứng dụng nhiều trong các class List, Dictionary và LINQ,… sử dụng Generic cho LINQ giúp chúng ta có thể tái sử dụng hàm cho nhiều kiểu dữ liệu, xét ví dụ sau về hàm swap:

Nếu không có Generic:

public void swap (int ref a, int ref b);

public void swap (double ref a, double ref b);

public void swap (bool ref a, bool ref b);

public void swap (object ref a, object ref b);

.........

//Nếu muốn swap 2 student, phải viết thêm hàm

public void swap (Student ref a, Student ref b);

Với sức mạnh của Generic, ta chỉ cần viết 1 và chỉ 1 method duy nhất như sau:

//Dấu <T> để biết method là generic method

public static void Swap<T>(ref T a, ref T b)

{

  T temp = b;

  b = a;

  a = temp;

}

//Sử dụng hàm này như sau:

int firstInt = 1;

int secondInt = 2;

Swap<int>(ref firstInt, ref secondInt);

double firstDouble = 1.5;

double secondDouble = 2.5;

//Có thể bỏ qua dấu <> vì compiler tự hiểu kiểu double

//Dựa trên tham số ta truyền vào

Swap(ref firstDouble, ref secondDouble);

Hoặc xét một ví dụ hàm với giá trị trả về thuộc kiểu chưa xác định:

public static T getRandomElement<T>(List<T> list)

{

Random ran = new Random();

int randomIndex = ran.Next(list.Count - 1);

return list[randomIndex];

}

//Sử dụng

List<int> intList = new List<int> {1,2,3,4,5,6 };

List<double> doubleList = new List<double> {0.4, 0.6,0.8,4.5,0.2 };

var randomInt = getRandomElement(intList);

var randomDouble = getRandomElement(doubleList);

Generic Method kết hợp với Extension Method góp phần giúp cho các truy vấn LINQ trở nên đa dạng và nhất quán hơn.

## IEnumerable, IQueryable và Yield:

1. IEnumerable:

IEnumerable là một mảng read-only nằm trong namespace **System.Collections** với các tính chất sau:

* + - * + Enumerable có thể duyệt các phần tử chỉ 1 chiều tiến lên, nó không thể duyệt ngược lại giữa các phần tử.
        + IEnumerable hoạt động tốt nhất khi truy vấn từ một collection in-memory tức là trong bộ nhớ RAM như List, Array…
        + Khi truy vấn dữ liệu từ database, IEnumerable thực thi câu lệnh select trên server sau đó tải toàn bộ dữ liệu về client rồi mới lọc dữ liệu.
        + IEnumerable phù hợp với Linq to Object và Linq to XML
        + IEnumerable không hỗ trợ custom query
        + IEnumerable không hỗ trợ lazy loading(không load khi khởi tạo, chỉ load khi có yêu cầu) vì thế không phù hợp với trường hợp phân trang.

Ví dụ về IEnumerable:

MyDataContext dc = new MyDataContext ();

IEnumerable<Student> list = dc. Student s.Where(p => p.Name.StartsWith("S"));

list = list.Take< Student >(10);

Câu lệnh của đoạn code sẽ được generate sang SQL như sau:

SELECT [t0].[ID], [t0].[Name], [t0].[Class] FROM [Student] AS [t0] WHERE [t0].[Name] LIKE @p0

Chú ý là trong câu lệnh được tạo ra, sẽ không có “TOP 10” vì IEnumerable sẽ lọc các bản ghi ở dưới client. Nên toàn bộ bản ghi trước khi load ra top 10 sẽ được tải về client.

1. IQueryable:

IQueryable nằm trong namespace **System.Linq** được tạo ra để khắc phục các thiếu sót của IEnumerable, với các tính chất mới như sau:

* + - * + IQueryable cũng chỉ có thể di chuyển 1 chiều tiến lên trong collection, nó không thể move back lại.
        + IQueryable hoạt động tốt nhất với những truy vấn dữ liệu out-memory như là database.
        + Khi truy vấn, IQueryable thực thi câu lệnh truy vấn và lọc dữ liệu trực tiếp trên Server.
        + IQueryable phù hợp cho Linq to SQL
        + IQueryable hỗ trợ custom query sử dụng phương thức CreateQuery và Execute.
        + IQueryable hỗ trợ lazy loading. Vì thế nó phù hợp cho trường hợp phân trang.

Ví dụ về IQueryable:

MyDataContext dc = new MyDataContext ();

IQueryable<Student> list = dc. Student.Where(p => p.Name.StartsWith("S"));

list = list.Take<Student>(10);

Câu lệnh sẽ được Generate như sau:

SELECT TOP 10 [t0].[ID], [t0].[Name], [t0].[Class] FROM [Student] AS [t0] WHERE [t0].[Name] LIKE @p0

Chú ý: vì IQueryable thực thi câu lệnh và lọc dữ liệu trực tiếp trên server vì thế câu lệnh được tạo ra có thêm chữ “TOP 10”. Cũng vì lí do này, câu lệnh IQueryable sẽ không thực thi ngay khi khởi tạo mà sẽ nó đợi cho đến khi ToList() được gọi hoặc được đưa vào duyệt thì mới thực thi.

1. Yield:

Ta xét trường hợp sau, đọc danh sách học sinh từ file:

public List<Student> ReadStudentsFromFile(string fileName)

{

  string[] lines = File.ReadAllLines(fileName);

  //Tạo một list trống

  List<Student> result = new List<Student>();

  foreach (var line in lines)

  {

    Student student = ParseTextToStudent(line);

    result.Add(student); //Thêm student vào list

  }

  return result; // Trả list ra

}

var students = ReadStudentsFromFile("students.txt");

foreach(var student in students) {};

Đoạn code này không có gì sai. Tuy nhiên ta thấy việc tạo list, thêm phần tử vào list và trả về list có thể được rút gọn với từ khóa Yield như sau:

//Đổi kiểu trả về là IEnumerable

public IEnumerable<Student> ReadStudentsFromFile(string fileName)

 {

   string[] lines = File.ReadAllLines(fileName);

   foreach (var line in lines)

   {

      Student student = ParseTextToStudent(line);

      yield return student; //YIELD

   }

 }

//Dùng như cũ

var students = ReadStudentsFromFile("students.txt");

foreach(var student in students) {};

Việc sử dụng từ khóa Yield ở đoạn code trên không chỉ giúp số dòng code giảm đi mà còn mang lại rất nhiều lợi ích khác.

Chúng ta đều biết điều cơ bản nhất khi viết 1 method: Từ khóa return sẽ kết thúc method, trả ra kết quả, ko chạy thêm bất kì câu lệnh gì phía sau:

public int GetNumber() { return 5; }

Console.WriteLine(GetNumber());

Vậy xét trường hợp nếu ta dùng yield return 3 lần thì sao:

public IEnumerable<int> GetNumber()

{

yield return 5;

yield return 10;

yield return 15;

}

foreach(int i in GetNumber())Console.WriteLine(i);//5 10 15

Ta có thể hiểu luồng chạy của chương trình như sau:

1. Khi gọi method GetNumber, lấy phần từ đầu tiên, chương trình chạy tới dòng lệnh số 3, lấy ra kết quả là 5, in ra console.
2. Duyệt tiếp phần từ tiếp theo, chương trình chạy vào dòng lệnh số 4, lấy kết quả 10, in ra màn hình.
3. Tương tự với phần tử cuối cùng, sau khi in ra, chương trình kết thúc.

Xét với trường hợp 2 method Student ở đầu:

* + - * + Ở method đầu, ta trả về kết quả sau khi đã chạy hết hàm for, đưa kết quả vào trong 1 list mới, hàm ReadStudentsFromFile kết thúc.
        + Ở method thứ 2, kết quả được trả về ngay sau khi parse được student đầu tiên, với mỗi vòng lặp tiếp theo, chương trình sẽ chạy tiếp vào method YieldReadStudentsFromFile, lấy kết quả ra dần dần.

Sau khi hiểu bản chất, ta có thể ứng dụng yield vào các trường hợp như sau:

* + - * + Cần method trả về là một danh sách read-only, chỉ đọc, không được thêm xóa sửa.
        + Như trường hợp trên, giả sử ta có 50 dòng, hàm ParseTextToStudent tốn 1s 1 lần. Với cách cũ, khi gọi hàm ReadStudentsFromFile, ta phải đợi 50s. Với hàm YieldReadStudentsFromFile, hàm ParseTextToStudent chỉ được chạy mỗi khi ta đọc thông tin của học sinh, do đó tăng performance lên rất nhiều (Nếu ta chỉ lấy 5 học sinh đầu chỉ cần đợi 5s).
        + Trong một số trường hợp, danh sách trả về có vô hạn phần tử, hoăc lấy toàn bộ phần tử rất mất thời gian, ta phải sử dụng yield để giải quyết.

Thực tế khi dùng hàm yield, C# sẽ compile method đã viết lại thành 1 state machine, implement các method Next, Current, Max,… của Ienumerator. Chi tiết tại: http://coding.abel.nu/2011/12/return-ienumerable-with-yield-return/

# Cấu trúc và đặc tính của LINQ:

## Cấu trúc cơ bản của LINQ và biểu thức LINQ:

Namespace **System.Linq**;

Namespace **System.Data.Linq**;

Thực tế LINQ là một danh sách các Extension method, thêm một số method cho Interface IEnumerable (các class List, Dbset, ArrayList, Dictionary,… đều kế thừa từ IEnumerable do đó chúng cũng có thể gọi Linq method).

Linq cơ bản bao gồm 3 signature: **Where, Select và First.**

public static class Enumerable{

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

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

   public static TSource **First**<TSource>(this IEnumerable<TSource> source, Func<TSource, bool> predicate);

}

Các method này nhận đầu vào là 1 delegate Func<> với tham số thuộc kiểu Generic(Tsource, TResult) do đó ta có thể khai báo kiểu gì cũng được.

Chúng ta cũng có thể tự tạo ra được Linq method cho riêng mình bằng cách viết Extension method cho IEnumerable. Với signature tương tự như Linq, ví dụ:

public static class MyLinq

{

  //Extension method

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

  {

    //Loop toàn bộ cái item trong danh sách truyền vào

    foreach (var item in source)

    {

      // Callback lại hàm đã truyền vào, hàm này trả giá trị boolean

      // Nếu hàm callback return true, đưa item đó vào IENumerable kết quả

      // Xem lại vài IEnumerable và yield

      if (predicate(item))  yield return item;

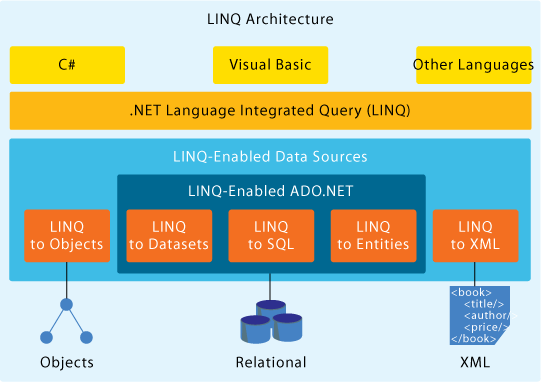
    }

  }

}

Nhờ đặc tính này cũng như một số đặc tính mềm dẻo khác của LINQ, rất nhiều thư viện LINQ mở rông đã được xây dựng (LINQ to JSON: <https://www.newtonsoft.com/json/help/html/LINQtoJSON.htm>, LINQ for MongoDB: <https://mongodb.github.io/mongo-csharp-driver/1.11/linq/>, GraphQL to Linq.IQueryable: <https://github.com/ckimes89/graphql-net>,...)

Trong một truy vấn LINQ, bạn phải luôn luôn làm việc với các đối tượng. Bạn sử dụng giống như một truy vấn cơ bản và mã hóa, chuyển đổi dữ liệu sang các tài liệu XML, cơ sở dữ liệu SQL, ADO.NET DataSet và cho bất kì một định đạng nào mà một nhà cung cấp thư viện LINQ cho phép.

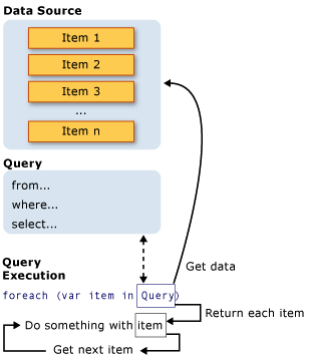


### Ba phần của một biểu thức LINQ:

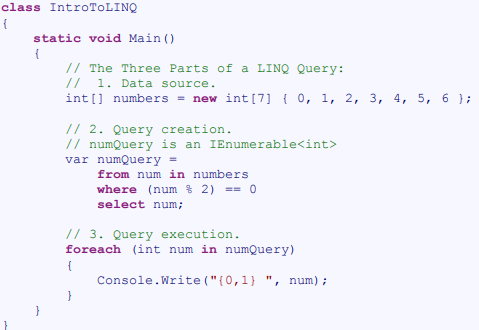
Tất cả các biểu thức LINQ làm việc theo ba thao tác:

1. Có được các dữ liệu nguồn.
2. Tạo các truy vấn.
3. Thực thi các truy vấn.

Minh họa sau đây cho thấy các hoạt động truy vấn tìm kiếm được hoàn tất. Trong LINQ các truy vấn thực hiện riêng biệt từ bản thân câu truy vấn. Nói cách khác bạn không lấy ra bất kỳ dữ liệu nào bằng cách tạo ra một biến truy vấn.



Ví dụ sử dụng một mảng số nguyên như là một dữ liệu nguồn (khái niệm này có thể áp dụng cho tất cả các dữ liệu nguồn). Ví dụ này sẽ được sử dụng trong suốt phần còn lại của chủ đề này.



### Các dữ liệu nguồn:

Trong ví dụ trước vì dữ liệu là một mảng, nó hoàn toàn có các điểm chung của IEnumerable. Do đó chúng ta có thể thực hiện truy vấn LINQ lên nó. Một truy vấn được thực hiện trong một câu lệnh foreach và foreach yêu cầu đầu vào là IEnumerable hay IEnumerable<T>. Các kiểu có hỗ trợ IEnumerable<T> hoặc các Interface như IQueryable<T> được gọi là các loại queryable. Một loại queryable không yêu cầu phải sửa đổi hay xử lý đặc biệt để phục vụ LINQ. Nếu một CSDL chưa được định nghĩa trong bộ nhớ như là một queryable, một nhà cung cấp LINQ phải xay dựng nó. Ví dụ, LINQ to XML giúp đưa dữ liệu XML vào một queryable Xelement

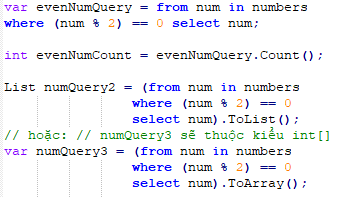
Với LINQ to SQL trước tiên bạn phải thiết kế một đối tượng quan hệ, công việc này được làm thủ công hoặc bằng cách sử dụng trình thiết kế đối tượng quan hệ(O/R Designer).

### Truy vấn và thực thi truy vấn:

Truy vấn trong ví dụ trước trả về tất cả các số từ mảng số nguyên. Các biểu thức truy vấn chứa ba mệnh đề: **from**, **where**, **select**(Vị trí của các mệnh đề có đôi chút thay đổi so với trong SQL).

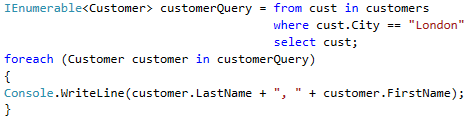
Mệnh đề **from** dùng để xác định dữ liệu nguồn, mệnh đề **where** dùng để lọc dữ liệu, mệnh đề **select** dùng để chọn ra những phần tử được trả về

* + - 1. Thực thi truy vấn:
         * Cũng giống như trạng thái trước, biến truy vấn chỉ chứa các lệnh truy vấn. Sự thực thi của các truy vấn sẽ hoãn lại đến khi bạn gọi tới biến truy vấn trong câu lệnh foreach. Cái này làm cơ sở đệ quy cho hoãn thực thi(Lazy Loading).
         * Câu lệnh foreach là nơi các kết quả truy vấn được trả về, ở ví dụ trên, biến num được lặp và nắm giữ mỗi giá trị trong trình tự trả về.
         * các biến truy vấn tự nó không bao giờ chứa kết quả truy vấn, bạn có thể thực hiện nó nhiều lần nếu muốn
      2. Thực thi bắt buộc tức thời:
         * Là truy vấn thực hiện kết hợp các chức năng trên một loạt các phần tử nguồn đầu tiên và phải lặp đi lặp lại trên danh sách các nhần tử. Ví dụ như các truy vấn Count, Max, Average, và First.
         * Những thực thi này không có một câu lệnh foreach nào rõ ràng bởi vì các truy vấn tự nó phải sử dụng foreach để trả về một kết quả.



### Các biến IEnumerable có trong các câu truy vấn LINQ:

Các biến kết quả trong câu truy vấn LINQ sẽ có kiểu IEnumerable<T> hoặc có kiểu IQueryable<T>. Ví dụ bạn viết truy vấn với biến kiểu IEnumerable<Customer>, nó đơn giản là các thức truy vấn, khi nó được thực hiện sẽ tạo ra một chuỗi gồm không hoặc nhiều đối tượng Customer.



## Các hoạt động truy vấn cơ bản:

Cấu trúc các truy vấn của LINQ chủ yếu tương tự như ở các ngôn ngữ truy vấn như SQL hoặc XQuery

### Truy xuất dữ liệu nguồn:

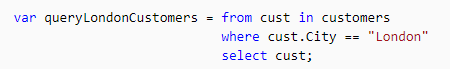
Trong một truy vấn LINQ, bước đầu tiên là xác định dữ liệu nguồn. Trong C# cũng như trong hầu hết các ngôn ngữ lập trình một biến phải được khai báo trước khi nó có thể được sử dụng. Trong một truy vấn LINQ, mệnh đề from đứng đầu tiên để giới thiệu các dữ liệu nguồn (customer) và các phạm vi biến (cust).



Phạm vi biến giống như các biến lặp trong một vòng lặp foreach, ngoại trừ không có một vòng lặp thực sự nào xảy ra trong một biểu thức truy vấn. Khi truy vấn được thực thi, phạm vi biến sẽ là một tham chiếu lần lượt đến các các phần tử trong mỗi customers. Bởi vì trình biên dịch có thể phỏng đoán kiểu của cust, bạn không cần phải xác định nó trước. Ngoài ra phạm vi các biến cũng có thể được khai báo bởi một mệnh đề let.

### Lọc (Filtering):

Hoạt động truy vấn dữ liệu thông dụng nhất là áp dụng một bộ lọc vào biểu thức logic Boolean. Các bộ lọc giúp các truy vấn trả về những phần tử mà qua nó biểu thức trả về kết quả true. Kết quả được trình bày bằng cách sử dụng mệnh đề where. Các bộ lọc có tác dụng xác định các yếu tố đó để loại trừ liên tục từ dữ liệu nguồn. Trong ví dụ sau, sẽ chỉ trả về những khách hàng có địa chỉ ở London



Khi truy vấn, chúng ta có thể sử dụng các biểu thức tương tự như trong C# với các biểu thức logic AND và OR để áp dụng cho bộ lọc có nhiều điều kiện trong mệnh đề where. Ví dụ, chỉ trả về các khách hàng có địa chỉ tại"London" và có tên là "Devon" bạn sẽ viết đoạn code sau đây:

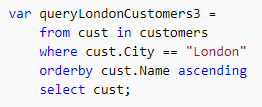


Để trả về khách hàng có địa chỉ ở London hay Paris, bạn sẽ viết mã sau:



### Sắp xếp (Ordering):

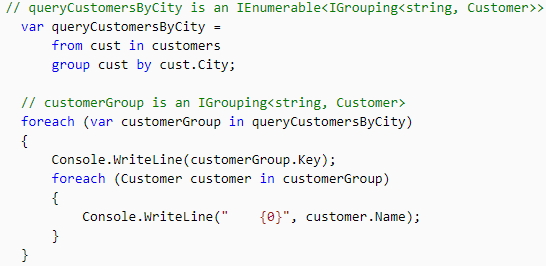
Mệnh đề orderby sẽ giúp các phần tử trong chuỗi trả về để được sắp xếp theo loại và so sánh với kiểu của dữ liệu được khác định. Ví dụ, truy vấn sau đây được viết thêm phân loại các kết quả dựa trên thuộc tính Name. Bởi vì thuộc tính Name là một chuỗi, mặc định nó sẽ so sánh và thực hiện sắp xếp theo thứ tự chữ cái từ A đến Z.



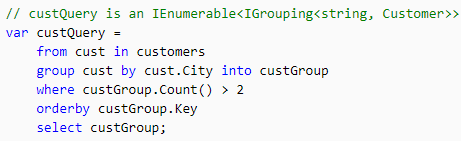
Để các kết quả trong thứ tự đảo ngược trật tự, từ Z to A, sử dụng mệnh đề orderby ...descending.

### Gom nhóm (Grouping):

Mệnh đề group cho phép bạn nhóm các kết quả lại dựa trên một khóa mà bạn chỉ định. Ví dụ, bạn có thể xác định rằng các kết quả cần được nhóm lại theo thuộc tính City để tất cả các khách hàng từ London, Paris hay cá nhân đang có trong nhóm. Trong trường hợp này, cust.City chính là khóa.

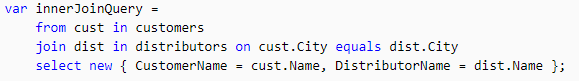


Khi bạn kết thúc một truy vấn với mệnh đề group, các kết quả của bạn sẽ sẽ được trả về một danh sách các danh sách. Mỗi phần tử trong danh sách là một đối tượng trong đó có một Key và danh sách các phần tử được gom nhóm bởi key này. Khi bạn duyệt qua một truy vấn mà kết quả là một nhóm có thứ tự, bạn cần phải sử dụng một vòng lặp foreach. Nếu bạn muốn tham khảo các kết quả thực thi của việc gom nhóm, bạn có thể sử dụng từ khóa into để tạo ra một định danh có thể truy vấn được. Dưới đây là những truy vấn trả về chỉ những nhóm có chứa nhiều hơn hai khách hàng



### Kết (Joining):

Thao tác Join tạo ra sự kết hợp giữa nhiều bảng không được mô hình hóa rõ ràng trong các dữ liệu nguồn. Ví dụ, bạn có thể thực hiện một thao tác để tìm tất cả các khách hàng và nhà phân phối ở cùng một địa điểm. Trong LINQ mệnh đề Join luôn luôn tham gia các hoạt động trên tập đối tượng thay vì các bảng CSDL.



*Trong LINQ bạn không nhất thiết phải sử dụng mệnh đề Join thường xuyên như trong SQL bởi vì các khóa ngoại LINQ miêu tả trong mô hình như là một tập chứa danh sách các đối tượng liên quan. Ví dụ, một đối tượng Customer có chứa một tập các đối tượng Order. Thay vì sử dụng join, bạn có thể truy cập tập dữ liệu bằng dấu chấm*



### Lựa chọn (Selection/Projection):

Mệnh đề Select đưa ra các kết quả trả về cho một câu truy vấn và xác định "hình dạng" hoặc kiểu của mỗi phần tử trả về. Ví dụ, bạn có thể xác định cho dù kết quả của bạn sẽ bao gồm tất cả các đối tượng Customer, chỉ cần một thành phần, một nhóm của các thành phần, hoặc một nhóm kết quả loại hoàn toàn khác nhau dựa trên truy vấn, tính toán để tạo thành một kiểu đối tượng mới. Khi mệnh đề Select trả về một cái gì đó khác với một bản sao của các phần tử nguồn, thao tác được gọi là phép chiếu(projection). Việc sử dụng các phép chiếu để chuyển đổi dữ liệu là một khả năng của biểu thức truy vấn LINQ

**Bảng các toán tử thông dụng của LINQ:**

|  |  |  |
| --- | --- | --- |
| Filtering | Where, OfType | |
| Join | Join, GroupJoin | |
| Projection | SelectMany |
| Sorting | OrserBy, OrderByDescending, ThenBy, ThenByDescending, Reverse |
| Grouping | GroupBy, ToLookup |
| Conversion | AsEnumerable, AsQueryable, Cast, OfType, ToArray, ToDictionary, ToList, ToLookup |
| Concatenation | Concat |
| Aggretion | Aggregate, Average, Count, LonCount, Max, Min, Sum |
| Quantifier | All, Any, Contains |
| Partition | Skip, SkipWhile, Take, TakeWhile |
| Generation | DefaultIfEmpty, Empty, Range, Repeat |
| Set | Distinct, Except, Intersect, Union |
| Equality | SequenceEqual |
| Element | ElementAt, ElementAtOrDefault, First, FirstOrDefault, Last, LastOrDefault, Single, SingleOrDefault, DefaultIfEmpty |

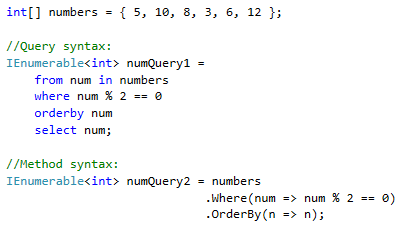
## Chuyển đổi dữ liệu với LINQ:

LINQ không chỉ là truy xuất dữ liệu. LINQ còn là một công cụ mạnh mẽ cho việc biến đổi dữ liệu. Bằng cách sử dụng một truy vấn LINQ, bạn có thể sử dụng một Danh sách dữ liệu nguồn vào và sửa đổi nó bằng nhiều cách để tạo ra một danh sách mới. Bạn có thể sửa đổi trình tự của nó mà không sửa đổi các phần tử bên trong bằng cách phân loại và gom nhóm. Tính năng mạnh mẽ nhất trong của các câu truy vấn LINQ là khả năng tạo ra kiểu mới(sử dụng Anonymous Type). Tính năng này được áp dụng nhiều nhất trong mệnh đề select.

## Cú pháp truy vấn (Query Syntax) và cú pháp phương thức (Method Syntax):

Hầu hết các truy vấn LINQ thông dụng đều được viết bằng cách sử dụng cú pháp truy vấn(Query Syntax) theo LINQ. Tuy nhiên, cú pháp truy vấn phải được dịch thành các lệnh gọi phương thức dành cho .NET common language runtime (CLR) khi code được biên dịch. Các lệnh gọi phương thức này sẽ sử dụng các toán tử truy vấn tiêu chuẩn, các toán tử này có các tên Where, Select, GroupBy, Join, Max và Average. Bạn có thể gọi chúng trực tiếp bằng cách sử dụng cú pháp phương thức(Method Syntax) thay vì cú pháp truy vấn.

Query Syntax và Method Syntax giống hệt nhau về mặt ngữ nghĩa, nhưng nhiều người cho rằng Query Syntax đơn giản hơn và dễ đọc hơn. Một số truy vấn phải được thể hiện dưới dạng các lệnh gọi phương thức. Ví dụ: bạn phải sử dụng lệnh gọi phương thức để thể hiện một truy vấn truy xuất số lượng phần tử phù hợp với một điều kiện nhất định. Bạn cũng phải sử dụng lệnh gọi phương thức cho truy vấn truy xuất phần tử có giá trị tối đa trong chuỗi nguồn. Tài liệu tham khảo cho các toán tử truy vấn tiêu chuẩn trong namespace **System.Linq** thường sử dụng Query Syntax. Do đó, ngay cả khi bắt đầu viết các truy vấn LINQ, việc làm quen với cách sử dụng Query Syntax trong các truy vấn và trong các biểu thức truy vấn là rất hữu ích.



## LINQ và Kiểu mẫu (Generic Type):

Như đã nêu ở mục III.4, Các câu truy vấn LINQ được dựa trên các kiểu dạng chung hay kiểu mẫu(Generic type), đã được giới thiệu trong phiên bản 2.0 của. NET Framework. Bạn không cần thiết phải có các kiến thức chuyên sâu về các Generic Type này trước khi bắt đầu viết các truy vấn. Tuy nhiên, bạn nên hiểu rõ hai khái niệm cơ bản:

1. Khi bạn tạo một thể hiện từ danh sách kiểu Generic như List<T>, bạn thay thế "T" với các loại đối tượng trong List đó sẽ chứa. Ví dụ, một danh sách các xâu kí tự được thể hiện như List<string> , và một danh sách Customer các đối tượng khách hàng được thể hiện như List<Customer>. Một danh sách các Generic Type cung cấp nhiều lợi ích cho một tập hợp khi xem các phần tử của chúng như đối tượng. Nếu bạn cố gắng để thêm một Customer vào trong một List<string>, bạn sẽ nhận được một lỗi compile. Sử dụng các tập hợp có chung đặc điểm rất dễ dàng và tiện lợi vì bạn không cần phải thực hiện ép kiểu trong thời gian chạy chương trình(run-time type-casting).
2. IEnumerable<T> là một Interface cho phép tạo các lớp Generic Collection có thể liệt kê được bằng cách sử dụng câu lệnh foreach. Các lớp Generic Collection hỗ trợ IEnumerable<T> tương tự như cách các lớp non-generic collection như ArrayList hỗ trợ IEnumerable.

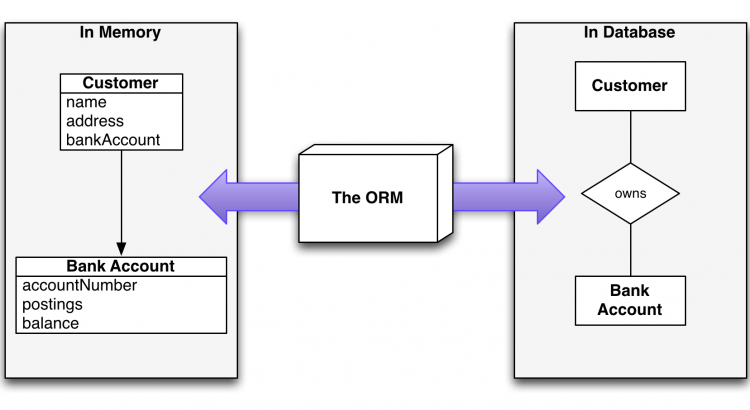
## Mô hình đối tượng quan hệ (Object Relational Mapping) và thiết kế đối tượng quan hệ (Object Relational Designer):

### Mô hình đối tượng quan hệ (ORM):

#### Tổng quan:

Object Relational Mapping hay ORM là tên gọi chỉ việc ánh xạ các bản ghi dữ liệu trong hệ quản trị cơ sở dữ liệu sang dạng đối tượng mà mã nguồn định nghĩa trong class. Là một khái niệm phổ biến, được cài đặt trong tất cả các loại ngôn ngữ hiện đại ngày nay như: java, php, node.js, swift, C# … Do đó, chúng ta có thể dễ dàng có thể cài đặt ORM hoặc sử dụng các thư viện mã nguồn mở về ORM trong bất cứ dự án nào mình thích.

Để cài đặt kỹ thuật này đơn giản là chúng ta cài đặt một lớp thực hiện việc kết nối CSDL và các chức năng tương tác với CSDL như Select, Insert, Delete, Update và các tương tác trên các đối tượng dữ liệu có quan hệ với nhau.



object relational mapping tự động hóa chuyển đổi từ object sang table và ngược lại từ table sang object rất nhanh chóng giúp giảm thời gian cũng như chi phí phát triển. Object relational mapping cũng cần ít code hơn sotre procedures. Object relational mapping có khả năng thay thế số lượng lớn store procedudres cần phát triển. Bên cạnh đó, ORM cũng giúp tăng tốc thực thi của hệ thống.

#### Những ưu điểm và nhược điểm của ORM:

* Ưu điểm:
  + **Portable – tính năng động:** ORM được sử dụng để bạn viết cấu trúc của bạn một lần và lớp ORM sẽ xử lý câu lệnh cuối cùng phù hợp với DBMS được cấu hình. Đây là một lợi thế tuyệt vời khi thao tác truy xuất dữ liệu đơn giản như giới hạn được thêm vào dưới dạng ‘limit 0,100’ ở cuối câu lệnh Select trong MySQL, trong khi đó với cách truy xuất thông thường bạn phải viết là ‘Select Top 100 From Table’ trong MS SQL.
  + **Nesting Of Data – truy xuất lồng dữ liệu:** trong trường hợp database bạn có nhiều bảng và các bảng này liên hệ rối rắm phức tạp về dữ liệu thì ORM sẽ tự động lấy dữ liệu cho bạn một cách đơn giản (ở đây đang nói tới việc đơn giản trong truy xuất dữ liệu, không bàn về vấn đề tối ưu truy xuất)
  + **Single Language – không cần biết SQL:** thật vậy với nguyên lý thiết kế là ánh xạ toàn bộ dữ liệu lấy được từ DBMS sang bộ nhớ nên việc thao tác truy suất bây giờ chỉ phụ thuộc vào ngôn ngữ lập trình bạn đang xử dụng, bạn chẳng cần quan tâm phía đằng sau của ORM sẽ làm gì sinh ra mã SQL như thế nào khi truy xuất SQL, và kết quả là chúng ta chỉ cần nhuần nhuyễn ngôn ngữ lập trình đang dùng.
  + **Adding is like modifying – thêm sửa dữ liệu là như nhau:** đối với ORM, nó không phân biệt giữa thêm mới và cập nhật mọi tác vụ có liên quan đến sửa đổi hay chèn dữ liệu đều được xem là định nghĩa thêm mới, hai tác vụ này được xem như là một.
* Nhược điểm:
  + **Slow – chậm:** trong tất cả các bài đánh giá và so sánh về tốc độ truy suất dữ liệu bạn sẽ thấy ORM dù cho được tối ưu đến đâu cũng sẽ không bao giờ nhanh hơn được so với truy vấn trực tiếp từ SQL.
  + **Tunning – khó điều chỉnh:** Tối ưu hóa các câu lệnh SQL là điều nên làm đối với các dữ liệu lớn, tuy nhiên đối với ORM quá trình này rất khó khăn và phức tạp, bởi vì bạn phải tuân theo thiết kế của ORM và ưu điểm mà nó mang lại là hạn chế và cô lập quá trình thao tác dữ liệu trực tiếp lên DBMS.
  + **Complex Queries – truy vấn phức tạp:** mặc dù được thiết kế để đơn giản hóa tuy nhiên một vài ORM thao tác dữ liệu rất rối rắm, các đoạn code rất phức tạp và khó hiểu hơn nhiều lần so với SQL thuần.
  + **Studing – khó học:** Đa phần các hệ thống với các tác vụ nặng và phức tạp thư viện ORM thuần túy được cung cấp không đảm bảo về hiệu năng, và bạn cùng các thành viên trong team chắc chắn phải nghiên cứu từng dòng code để tối ưu cho hệ thống của mình.

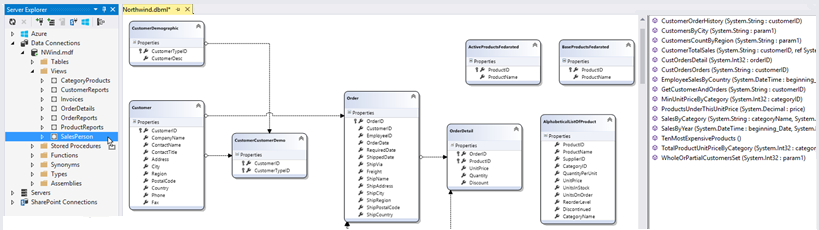
### Thiết kế đối tượng quan hệ (O/R desgner):

#### Tổng quan:

Thiết kế đối tượng quan hệ (O/R Designer) cung cấp một thiết kế trực quan cho việc tạo **LINQ to SQL** để tổ chức các thực thể và sự kết hợp (các mối quan hệ) dựa trên các đối tượng trong một cơ sở dữ liệu. Nói cách khác, các O/R được thiết kế sử dụng để tạo ra một mô hình đối tượng trong một ứng dụng để ánh xạ các đối tượng trong một cơ sở dữ liệu. Nó cũng tạo ra một kiểu DataContext rõ ràng để gửi và nhận dữ liệu giữa các lớp thực thể và cơ sở dữ liệu.

O/R Designer cũng cung cấp khả năng ánh xạ các thủ tục lưu trữ và các hàm để thực hiện các phương thức trong DataContext trả về các dữ liệu và các lớp thực thể. Ngoài ra, O/R Designer cung cấp khả năng thiết kế thừa kế các mối quan hệ giữa các lớp thực thể.

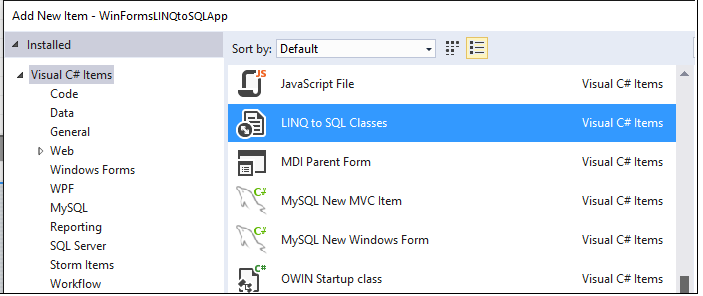
Giao diện thiết kế O/R Designer gồm hai khu vực riêng biệt: các thực thể(Entity) trong của sổ bên trái, và các phương thức(Method) trong cửa sổ bên phải. Cửa sổ các thực thể chính là giao diện thiết kế lớp thực thể, các liên kết, và các bậc kế thừa. Các phương thức trong của sổ bên phải là giao diện kế hiển thị DataContext các phương thức ánh xạ để lưu trữ các thủ tục và hàm.



#### Phương pháp thiết kế đối tượng quan hệ:

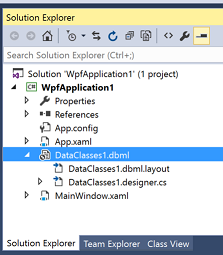
Việc xây dựng đối tượng quan hệ trong LINQ bao gồm ba thao tác:

1. Chọn Project > Add new item. Tại màn hình khởi tạo lựa chọn Data -> LINQ to SQL Classes, Đặt tên và nhấp chọn Add



1. Visual Studio sẽ tạo một file .dbml và thêm nó vào solution. Đây là file ánh xạ XML và các mã liên quan.

Lưu ý: Tên của biến DataContext được sử dụng cho việc truy vấn sau này sẽ có cấu trúc là: [tên\_file]+DataContext



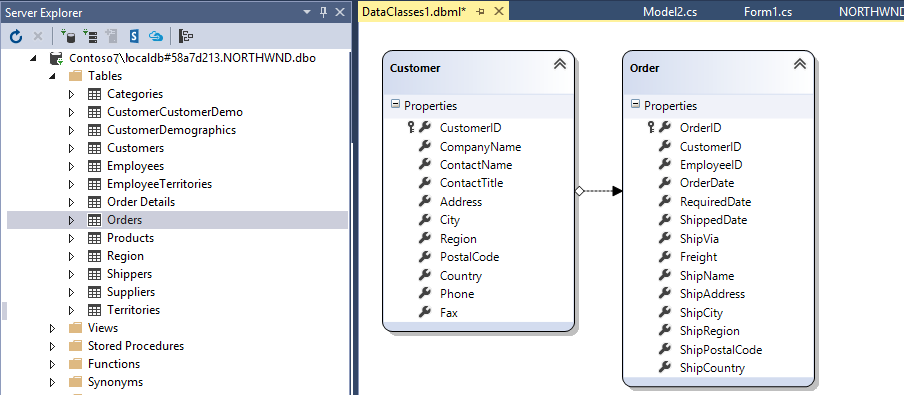
1. Khi bạn mở tệp .dbml, Visual Studio hiển thị giao diện trình thiết kế O/R cho phép bạn tạo mô hình một cách trực quan:
   * 1. Cấu hình và khởi tạo DataContext:

Sau khi bạn thêm một lớp LINQ to SQL cho một Project và mở thiết kế O/R, giao diện sẽ trống rỗng, DataContext sẵn sàng để được cấu hình. các DataContext được cấu hình sẽ kết nối với các thông tin được cung cấp bởi các thực thể đầu tiên được kéo vào cho việc thiết kế. Vì vậy, các DataContext được cấu hình bằng cách sử dụng kết nối thông tin từ các thực thể đầu tiên được kéo vào bản thiết kế.

* + 1. Tạo các lớp thực thể và kết hợp chúng với bảng dữ liệu:

Bạn có thể tạo các lớp thực thể được ánh xạ từ các bảng và các view bằng cách kéo thả các Tables và các Views từ Server Explorer/Explorer Database lên các bản thiết kế O/R. Như đã nói trong phần trước, DataContext được cấu hình để kết nối với các thông tin được cung cấp bởi các thực thể đầu tiên được kéo thả vào bản thiết kế. *Nếu một thực thể sử dụng một kết nối khác được thêm vào bản thiết kế O/R, bạn có thể thay đổi kết nối thủ công cho thực thể này trong DataContext, O/R designer cho phép tạo thiết kế đối tượng quan hệ từ nhiều CSDL khác nhau.*

Mệnh đề **from in** dùng để xác định dữ liệu nguồn, mệnh đề **where** dùng để lọc dữ liệu, mệnh đề **select** dùng để chọn ra những phần tử được trả về



* + 1. Tạo phương thức DataContext để chứa các thủ tục và hàm:

Bạn có thể tạo DataContext chứa các phương thức mà gọi (được ánh xạ tới) các thủ tục(StoredProcedure) và các hàm(Function) lưu trữ của CSDL bằng cách kéo chúng từ Server Explorer/Explorer Database lên các O/R Designer. Các thủ tục lưu trữ và các hàm này sẽ được sử dụng như các phương thức của DataContext.

* + 1. Cấu trúc DataContext để chứa các thủ tục lưu trữ dữ liệu giữa các lớp thực thể và cơ sở dữ liệu:

Như đã nêu trên, bạn có thể thêm các thủ tục lưu trữ(StoredProcedure) và các hàm(Function) và sử dụng chúng như một phương thức của DataContext. Ngoài ra, bạn cũng có thể chỉ định các thủ tục lưu trữ này để làm phương thức mặc định cho LINQ to SQL để thực hiện hành động insert, update, và delete.

### Các đặc tính của Thiết Kế Đối Tượng Quan Hệ:

Khóa: Các khóa chính và khóa ngoại trong O/R Designer đều có chức năng như các khóa chính và khóa ngoại trong CSDL(SQL). Đặc biệt, khi truy vấn đối tượng dữ liệu có tồn tại khóa ngoại, O/R Designer còn hỗ trợ truy vấn đến dữ liệu bảng chứa thông tin của khóa ngoại đó.

Hệ thống các quan hệ (Relationship): Giống như các đối tượng khác, các lớp LINQ to SQL có thể sử dụng được tính kế thừa và chuyển hóa từ các lớp khác. Trong một CSDL, các mối quan hệ thừa kế được tạo ra trong một số bảng. O/R Designer hỗ trợ các khái niệm về đơn(bảng) kế thừa như trong các hệ thống quan hệ của ngôn ngữ lập trình C#. Liên hệ (Association) giữa các thực thể trong LINQ to SQL cũng tương tự như mối quan hệ giữa các bảng trong CSDL, bạn có thể sử dụng hộp thoại Asociation Editor để tạo các quan hệ này.

Các truy vấn LINQ to SQL: Các lớp thực thể tạo bởi O/R Designer được thiết kế để sử dụng với các truy vấn LINQ.

Phân tách lớp DataContext và lớp thực thể(Entity) vào các namespace khác nhau: O/R Designer cung cấp các lựa chọn **Context Namespace** và **Entity Namespace** trên DataContext. Các thuộc tính này xác định namespace mà DataContext và Entity class được tạo. Thông thường lựa chọn này sẽ rỗng và DataContext, Entity class sẽ được tạo ở namespace của phần mềm. Để tạo code vào namespace khác với namespace của phần mềm, bạn có thể nhập giá trị vào Properties của Context Namespace và/hoặc Entity Namespace

# LINQ to SQL:

using **System.Linq**; **System.Data.Linq**; **System.ComponentModel**;

LINQ to SQL là một thành phần của .NET Framework (3.5 trở lên) cung cấp cơ sở hạ tầng thời gian thực để giúp quản lý các dữ liệu quan hệ như các đối tượng.

*Chú ý*: Các dữ liệu quan hệ sẽ có giao diện như là một danh sách của các bảng 2 chiều (các quan hệ hoặc các flat file), nơi mà các cột bảng liên quan đến nhau. Để sử dụng LINQ to SQL một cách có hiệu quả, bạn cần phải làm quen với một số khái niệm cơ bản về cơ sở dữ liệu quan hệ.

Trong LINQ to SQL, mô hình dữ liệu của cơ sở dữ liệu quan hệ được ánh xạ tới mô hình đối tượng được mô tả trong ngôn ngữ lập trình của lập trình viên. Khi các ứng dụng khởi chạy, LINQ to SQL dịch các truy vấn LINQ trong các mô hình đối tượng thành các truy vấn SQL và gửi chúng đến CSDL để thực thi. Khi cơ sở dữ liệu trả về kết quả, LINQ to SQL chuyển chúng trở lại thành các đối tượng để có thể sử dụng cho ngôn ngữ lập trình. LINQ to SQL hỗ trợ đầy đủ transaction, view và các stored procedure (SP).

Các lập trình viên sử dụng Visual Studio thông thường sẽ sử dụng các Object/Relational Designer (thết kế đối tượng quan hệ), thứ có thể cung cấp giao diện người dùng giúp họ thực thi nhiều tính năng của LINQ to SQL.

LINQ to SQL là một phần của dòng công nghệ ADO.NET. Nó được xây dựng dựa trên các dịch vụ được cung cấp bởi mô hình ADO.NET provider. Do vậy, bạn có thể phối hợp code LINQ to SQL với các ADO.NET application sẵn có và sử dụng các ADO.NET Provider cho LINQ to SQL. Ví dụ minh họa sau cung cấp cái nhìn cơ sở về các mối quan hệ

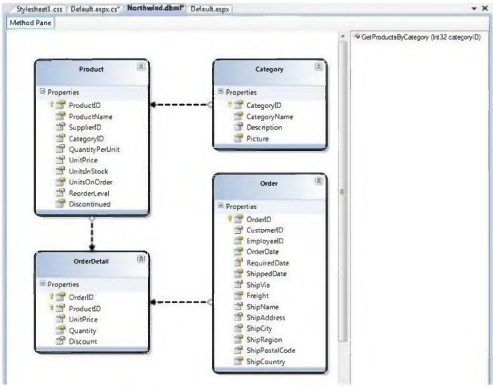


## Các Khái niệm trong LINQ to SQL:

### Mô hình hóa CSDL dùng LINQ to SQL(dùng thiết kế đối tượng quan hệ):

Như đã được giới thiệu ở mục VIII, từ phiên bản 2008 Visual studio đã tích hợp thêm một trình thiết kế LINQ to SQL như một công cụ dễ dàng cho việc mô hình hóa một cách trực quan các CSDL dùng LINQ to SQL.

Bằng cách dùng trình thiết kế LINQ to SQL, tôi có thề dễ dàng tạo một mô hình cho CSDL mẫu “Northwind” giống như dưới đây:



Mô hình LINQ to SQL ở trên định nghĩa bốn lớp thực thế: Product, Category, Order và OrderDetail. Các thuộc tính của mỗi lớp ánh xạ vào các cột của bảng tương ứng trong CSDL. Mỗi instance của một lớp biểu diễn một dòng trong bảng dữ liệu.

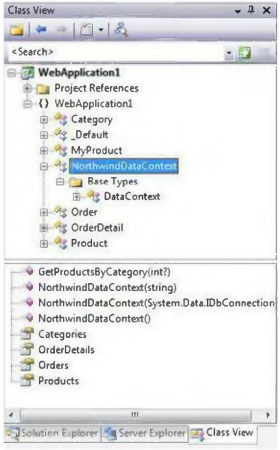
Các mũi tên giữa bốn lớp thực thể trên biểu diễn quan hệ giữa các thực thế khác nhau, chúng được tạo ra dựa trên các mối quan hệ primary-key/foreign-key trong CSDL. Hướng của mũi tên chỉ ra mối quan hệ là một - một hay một - nhiều. Các thuộc tính tương ứng sẽ được thêm vào các lóp thực thể trong các trường hợp này. Lấy ví dụ, lớp Category ở trên có một mối quan hệ một nhiều với lớp Product, điều này có nghĩa nó sẽ có một thuộc tính “Categories” là một tập hợp các đối tượng Product trong Category này. Lớp Product cũng sẽ có một thuộc tính “Category” chỉ đến đối tượng “Category” chứa Product này bên trong.

Bảng các phương thức bên tay phái bên trong trình thiết kế LINQ to SQL ớ trên chứa một danh sách các SP để tương tác với mô hình dữ liệu của chúng ta. Trong ví dụ trên tôi đã thêm một thủ tục có tên “GetProductsByCategory”. Nó nhận vào một categorylD và trả về một chuồi các Product. Chúng ta sẽ xem bằng cách nào có thế gọi được thủ tục này trong các đoạn code bên dưới.

### Lớp DataContext

Khi bạn bấm nút “Save” bên trong màn hình thiết kế LINQ to SQL, Visual Studio sẽ lưu các lớp .NET biểu diễn các thực thế và quan hệ bên trong CSDL mà chúng ta vừa mô hình hóa. Cứ mồi một file LINQ to SQL chúng ta thêm vào solution, một lớp DataContext sẽ được tạo ra, nó sẽ được dùng khi cần truy vấn hay cập nhật lại các thay đối. Lớp DataContext được tạo sẽ có các thuộc tính để biểu diễn mối bảng được mô hình hóa từ CSDL, cũng như các phương thức cho mỗi SP mà chúng ta đã thêm vào.

Lấy ví dụ, dưới đây là lớp NorthwindDataContext được sinh ra dựa trên mô hình chúng ta tạo ra ở trên:

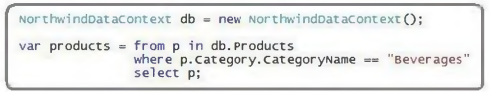


### Các câu lệnh làm việc trong LINQ to SQL:

Một khi đã mô hình hóa CSDL dùng trình thiết kế LINQ to SQL, chúng ta có thế dễ dàng viết các đoạn lệnh đế làm việc với nó. Dưới đây là một vài ví dụ về các thao tác chung khi xử lý dữ liệu:

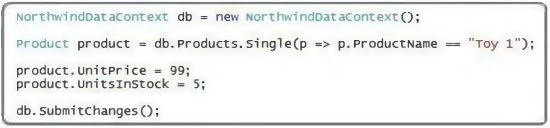
1. Truy vấn dữ liệu từ CSDL:

Đoạn lệnh dưới đây dùng cú pháp LINQ để lấy về một tập Enum erable các đối tượng Product. Các sán phấm được lấy ra phải thuộc phân loại “Beverages”:



1. Cập nhật dữ liệu vào CSDL:

Đoạn lệnh dưới đây cho thấy cách lấy một sản phấm, cập nhật lại giá tiền và lưu lại CSDL.



1. Chèn thêm dữ liệu mới vào CSDL:

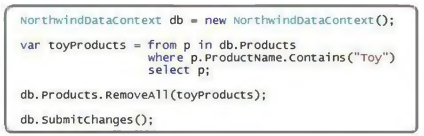
Đoạn mã dưới đây biểu diễn cách tạo một phân loại mới, và tạo hai sản phấm mới và đưa chúng vào trong phân loại đã tạo. Cá ba sau đó sẽ được đưa vào cơ sở dữ liệu. Chú ỷ rằng tôi không cần phái tự quán lý các mối quan hệ primary key/foreign key, thay vào đó, tôi chỉ đơn giản thêm các đối tượng Product vào tập hợp Products của đối tượng category, và rồi thêm đối tượng category vào tập hợp Categories của DataContext, LINQ to SQL sẽ biết cách thiết lập các giá trị primary key/foreign key một cách thích hợp.

(Add đã được thay đối bằng InsertOnSubmit trong phiên bản hiện tại)



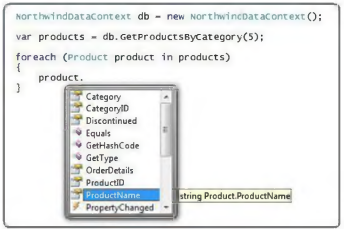
1. Xóa dữ liệu ở CSDL:

Doan ma sau se bieu dien cach xoa tat ca cac san pham Toy khoi CSDL: (RemoveAll da được thay doi bang DeleteAllOnSubmit trong phien ban hien tai)



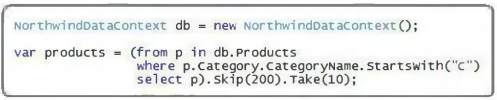
1. Gọi một thủ tục:

Doan ma duai day bieu dien cach lay cac thvrc the Product ma khong dung cu phap cua LINQ, ma goi den thu tuc “GetProductsByCategory” chung ta da them vao truac day. N ha rang mot khi da lay ve ket qua, toi co the cap nhat/xoa va sau do goi db.SubmitChanges() de cap nhat cac thay doi tra lai CSDL.



1. Thực hiện truy vấn có phân trang:

Đoạn mã dưới đây biểu diễn cách phân trang trên server như một phần của câu truy vấn LINQ. Bằng cách dùng các toán tử Skipo và Take(), chúng ta sẽ chỉ trả về 10 dòng tò CSDL - bắt đầu từ dòng 200.



## Xây dựng chương trình với LINQ to SQL:

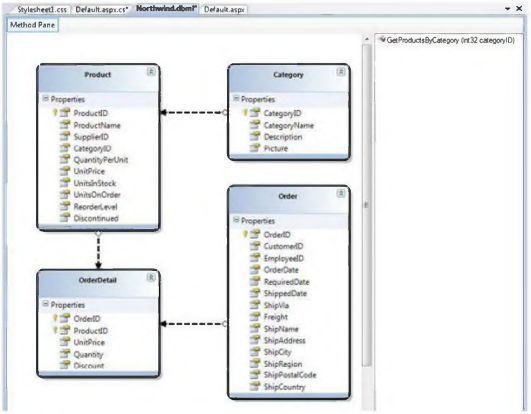
### Định nghĩa các lớp mô hình dữ liệu:

Trong phần mở đầu, chúng ta đã thảo luận về “LINQ to SQL là gì” và cung cấp một cái nhìn cơ bản về những trường hợp chúng ta có thể sử dụng nó.

Trong phần V.2 cũng đã cung cấp các đoạn code mẫu để biểu diễn cách xử lý dữ liệu dùng LINQ to SQL, bao gồm:

* Cách truy vấn dừ liệu
* Các cập nhật dữ liệu
* Cách chèn và tạo quan hệ các dòng trong một CSDL
* Cách xóa các dòng trong một CSDL
* Cách gọi một thủ tục
* Cách lấy dừ liệu và phân trang trên server

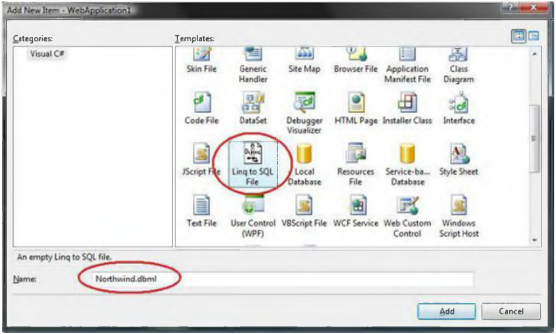
Chúng ta đã thực hiện tất cả các thao tác dữ liệu đó bằng cách dùng một mô hỉnh dữ liệu LINQ to SQL giống như dưới đây:



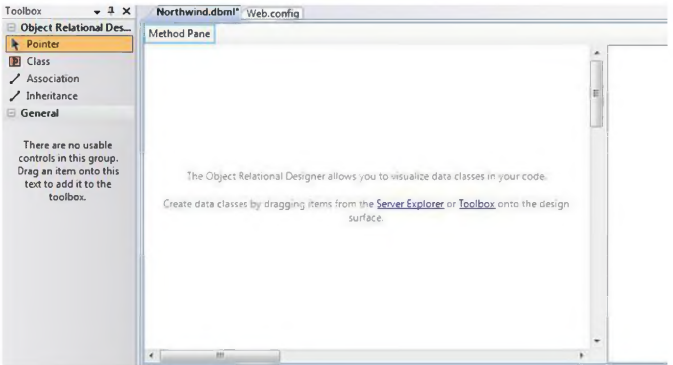
Trong bài này, Chúng sẽ đi vào chi tiết cách tạo ra một mô hình dữ liệu LINQ to SQL giống như trên.

1. Tạo ra một mô hình dữ liệu LINQ to SQL:

Bạn có thế thêm một mô hình dữ liệu LINQ to SQL và một dự án ASP.NET, Class Library hay Windows bang cách dùng tùy chọn “Add New Item” bên trong Visual Studio và chọn “LINQ to SQL”:



Việc chọn mục “LINQ to SQL” sẽ khởi chạy LINQ to SQL designer, và cho phép bạn mô hình hóa các lớp mà nó biếu diễn một CSDL quan hệ. Nó cũng sẽ tạo ra một lớp kiểu “DataContext”, trong đó có các thuộc tính đế biểu diễn mồi bảng mà chúng ta mô hình hóa trong CSDL, cũng như các phương thức cho mỗi Stored Procedure mà chúng ta mô hình hóa. Như tôi đã mô tả trong phần 1 của loạt bài này, lớp DataContext là thành phần trung tâm của mô hình, toàn bộ các thao tác truy vấn hoặc cập nhật dừ liệu đều được thực hiện thông qua lớp này. Dưới đây là ảnh chụp màn hình của một của số thiết kế LINQ to SQL, và cũng là cái mà bạn sẽ thấy ngay khi tạo ra một mô hình dừ liệu LINQ to SQL:



1. Các lớp thực thể:

LINQ to SQL cho phép bạn mô hình hóa các lớp ánh xạ từ CSDL. Các lớp này thường được là “Entity Class” (lớp thực thể) và các instance của nó thường được gọi là “Entity” (thực thế). Các lớp entity ánh xạ vào các bảng bên trong một đối tượng CSDL. Các thuộc tính của các lớp thông thường ánh xạ vào các cột trong bảng. Mỗi instance của một lớp thực thể biếu diễn một dòng trong bảng.

Các lớp thực thế trong LINQ to SQL không cần phải kế thừa từ một lớp đặc biệt nào khác, điều đó cho phép bạn có thế cho chúng thừa kế từ bất cứ đối tượng nào bạn muốn. Tất cả các lớp được tạo ra dùng LINQ to SQL designer đều được định nghĩa như “partial class'’ - có nghĩa là bạn có thế viết thêm code đế thêm các thuộc tính, phương thức và sự kiên cho chúng.

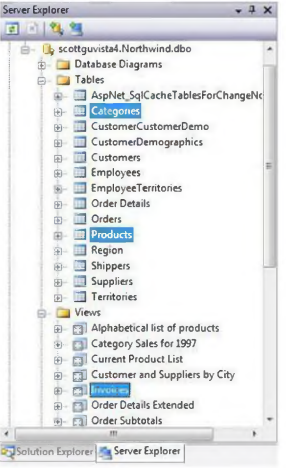
Không giống như chức năng DataSet/TableAdapter có trong vs 2005, khi dùng LINQ to SQL designer, bạn không cần chỉ ra câu truy vấn SQL được dùng đế tạo ra mô hình và lớp truy xuất dữ liệu.

Thay vào đó, bạn tập trung chủ yếu vào việc định nghĩa các lớp thực thể, cách chúng ánh xạ vào CSDL, và mối quan hệ giữa chúng. Trình LINQ to SQL cụ thể mà bạn dùng sẽ đảm bảo việc sinh ra các lệnh SQL thích hợp vào lúc chạy khi bạn tương tác và làm việc với các thực thể dữ liệu. Bạn có thế dùng cú pháp truy vấn LINQ để chỉ ra cách bạn muốn truy vấn dữ liệu.

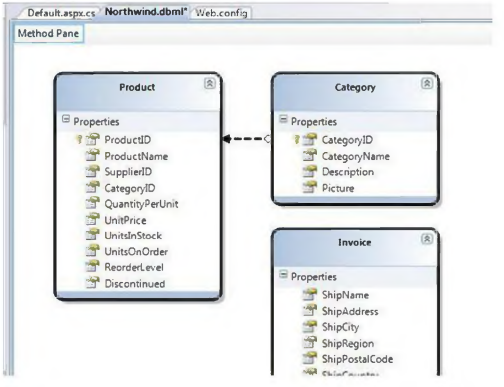
1. Tạo các lớp thực thể từ CSDL:

Nếu đã có cấu trúc CSDL, bạn có thể dùng nó để tạo các lớp thực thể LINQ to SQL một cách nhanh chóng.

Cách dễ dàng nhất để làm điều này là mở CSDL trong cửa sổ Server Explorer bên trong Visual Studio, chọn các table và view mà bạn muốn mô hình hóa, và kéo thả chúng lên trên của số LINQ to SQL designer.



Khi bạn thêm 2 bảng (Categories and Products) và 1 view (Invoices) từ CSDL “Northwind” vào cửa số LINQ to SQL designer, bạn sẽ có thèm 3 lớp thực thể được tạo ra một cách tự động:



Dùng các lớp mô hình hóa dữ liệu ở trên, bạn có thể chạy tất cả các đoạn lệnh mẫu được một tả trong phần V.1.3. Chúng ta có thể thực hiện được các thao tác query, insert, update, delete và phân trang mà không cần thêm bất kì đoạn code nào khác.

1. Cách đặt tên và ngữ pháp số nhiều:

LINQ to SQL có khả năng tự động chuyến tên bảng và cột thành dạng số nhiều khi tạo các lớp thực thể. Lấy ví dụ: Bảng “Products” trong ví dụ của chúng ta tạo ra lớp “Product”, cũng như bảng “Categories” tạo ra lóp “Category”. Cách đặt tên này giúp mô hình thống nhất với quy ước đặt tên trong .NET.

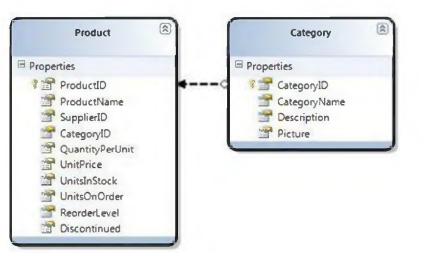
Nếu không thích tên lớp hay tên thuộc tính do trình designer sinh ra, bạn vẫn có thể sửa lại thành bất cứ tên nào bạn thích. Bạn có thể làm điều này bằng cách chỉnh sửa tên thực thể/thuộc tính bên trong trình thiết kế hoặc thông qua bảng thuộc tính.



Khả năng đặt tên cho các thực thể/thuộc tính/quan hệ khác với tên trong CSDL rất hữu dụng trong một số trường hợp, ví dụ:

1. Khi tên bảng/cột trong CSDL bị thay đối. Bởi vì mô hình thực thế có thế có tên khác với tên trong CSDL, do vậy ta có thể chỉ cần cập nhật lại các quy tắc ánh xạ mà không cần cập nhật chương trình hoặc các lệnh truy vấn để có thể dùng được tên mới.
2. Khi các thành phần bên trong CSDL được đặt tên không rõ ràng. Ví dụ: thay vì dùng “au\_lname” và “au\_fname” cho các tên thuộc tính của một lớp thực thể, ta có thế đặt tên chúng thành “LastName” và “FirstName” trong lớp thực thế và viết các lệnh để dùng với nó (mà không cần đổi tên các cột trong CSDL).
3. Quan hệ giữa các thực thể:

Khi ta kéo thả các đối tượng từ Server Explorer lên trên cửa sổ LINQ to SQL designer, VS sẽ tự động xác định các mối quan hệ primary key/foreign key giữa các đối tượng, và tự động tạo các quan hệ mặc nhiên giữa các lớp thực thể khác nhau mà nó đã tạo. Ví dụ, khi ta thêm cả hai bảng Products và Categories từ Northwind lên trên cửa số LINQ to SQL, ta có thể thấy mội mối quan hệ một nhiều giữa chúng (được biếu diễn bằng một mũi tên trên của sổ soạn thảo):



Mối quan hệ trên sẽ làm lớp thực thể Product có thêm một thuộc tính là Category, mà chúng ta có thể dùng để truy cập vào thực thể Category của một Product. Nó cũng làm lớp Category có thêm thuộc tính “Products”, đây là một tập hợp cho phép ta lấy ra tất cả các Product có trong Category đó.

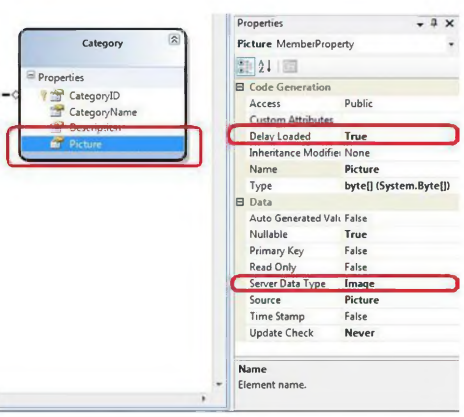


Nếu không thích cách mà trình thiết kế đã mô hình hóa hoặc đặt tên, chúng ta hoàn toàn có thế chỉnh sửa lại. Chỉ cần click lên mũi tên chỉ ra quan hệ trên của số soạn thảo và truy cập vào các thuộc tính của nó thông qua bảng thuộc tính để đổi tên, chỉnh sửa hoặc thậm chí xóa nó.

1. Delay/Lazy loading:

LINQ to SQL cho phép chỉ ra các thuộc tính của một thực thể sẽ được lấy về trước(prefetch) hay chỉ được lấy khi người dùng lần đầu truy cập (gọi là delaỵ/lazy loading). Chúng ta có thế tùy biến các quy tắc prefetch/lazy load cho các thuộc tính trong thực thể bằng cách chọn thuộc tính hay quan hệ đó, và đặt lại giá trị cho thuộc tính “Delay Loaded” thành true hoặc false.

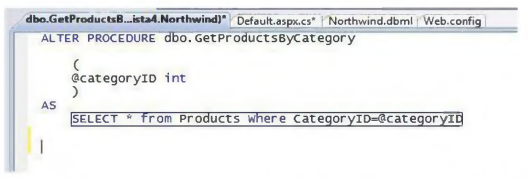
Ví dụ, cấu hình thuộc tính Picture đế nó chỉ được nạp khi dùng đến bằng cách đặt thuộc tính Delay Loaded thành true:



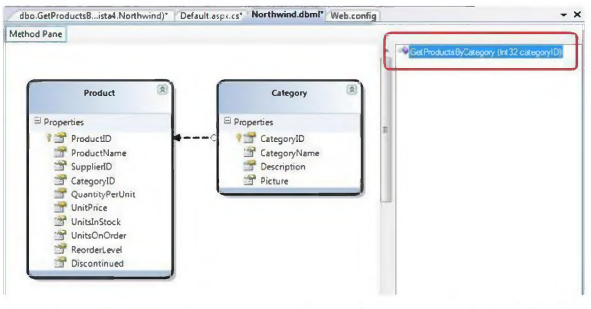
Ghi chú: Thay vì cấu hình prefetch/delay load trên các thực thế, chúng ta cũng có thế đặt lại thông qua các lệnh khi thực hiện các câu truy vấn LINQ trên lớp thực thế đó.

1. Dùng các StoredProcedure:

LINQ to SQL cho phép ta có thế mô hình hóa các thủ tục lưu trữ như là các phương thức trong lớp DataContext. Ví dụ, cho rằng chúng ta đã định nghĩa một thủ tục đơn giản có tên SPROC như dưới đây để lấy về các thông tin sản phấm dựa trên một Category ID:



Dùng Server Explorer trong VS để kéo/thả thủ tục SPROC lên trên cửa sổ soạn thảo LINQ to SQL để có thể thêm một phương thức cho phép gọi SPROC. Nếu ta thả SPROC lên trên thực thể “Product”, LINQ to SQL designer sẽ khai báo SPROC để trả về một tập kết quả có kiểu IEnumerable:

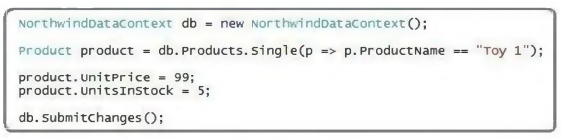


Sau đó ta có thể dùng cú pháp LINQ to SQL hoặc gọi thẳng phương thức này để lấy dữ liệu từ CSDL:

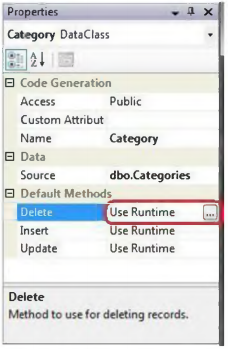


1. Dùng SPROCS để cập nhật/xóa, thêm dữ liệu:

Mặc nhiên LINQ to SQL sẽ tự động tạo ra các biếu thức SQL phù hợp mỗi khi chúng ta muốn cập nhật/xóa/thêm dữ liệu. Ví dụ, nếu bạn viết mã LINQ to SQL như dưới đây đổ cập nhật một số giá trị trên một thực thể “Product”:

Mặc nhiên, LINQ to SQL sẽ tạo và thực thi lệnh UPDATE tương ứng khi ta xác nhận thay đối (SubmitChanges).

Chúng ta cũng có thể định nghĩa và dùng các thủ tục INSERT, UPDATE, DELETE nếu muốn. để cấu hình, click vào một lớp thực thể trong cửa sổ LINQ to SQL và trong bảng thuộc tính, nhấn chuột vào nút “ ...” tìm các giá trị Delete/Insert/Update, và chọn SPROC mà ta đã định nghĩa.



Có một điều thú vị là những thay đổi trên hoàn toàn được thực hiện ở lớp ánh xạ LINQ to SQL - Có nghĩa là tất cả những đoạn code ta đã viết trước đó đều tiếp tục làm việc mà không cần thay đổi bất cứ điều gì. Điều này giúp ta tránh phải thay đổi lại code ngay cả nếu sau này muốn dùng một hàm SPROC để tối ưu thêm/xóa/sửa.

### Truy vấn Cơ sở dữ liệu:

1. Mô hình hóa CSDL Northwind dùng LINQ to SQL:
2. Lấy các sản phẩm:
3. Trực quan hóa các câu truy vấn LINQ to SQL trong trình gỡ lỗi:
4. Ghép nối các câu truy vấn LINQ to SQL vào các control LINQ to SQL:
5. Data Sharping:
6. Phân trang kết quả truy vấn:

### Cập nhật Cơ sở dữ liệu:

1. CSDL Northwind được mô hình hóa dùng LINQ to SQL:
2. Change Tracking và DataContext.SubmitChanges()
3. Các ví dụ Insert và Delete:
4. Thêm một sản phẩm:
5. Xóa một sản phẩm:
6. Cập nhật thông qua các quan hệ:
7. Transactions:
8. Kiểm tra dữ liệu và Business Logic:
9. Hỗ trợ kiểm tra các giá trị thuộc tính dựa trên schema của CSDL:
10. Hỗ trợ tùy biến việc kiểm tra giá trị các thuộc tính:
11. Hỗ trợ tùy biến kiểm tra tính hợp lệ của thực thể:
12. Tùy biến các phương thức kiểm tra việc thêm/xóa/sửa dữ liệu:
13. Nâng cao: Xem danh sách các thay đổi trong transaction:
14. Xử lý các thay đổi đồng thời với Optimistic Concurrency:
15. Dùng SPROCs hoặc tùy biến logic các câu SQL:

### Lấy dữ liệu dùng Stored Procedure:

1. Lựa chọn dùng SPROC hay không:
2. Các bước ánh xạ gọi SPROC dùng LINQ to SQL:
3. Ánh xạ một SPROC vào một DataContext của LINQ:
4. Gọi một SPROC mới được tạo:
5. Ánh xạ kiểu trả về của phương thức SPROC vào một lớp trong mô hình dữ liệu
6. Xử lý các tham số thủ tục dạng OUTPUT:
7. Xử lý các thủ tục trả viề nhiều kiểu kết quả khác nhau:
8. Hỗ trợ các hàm do người dùng tự định nghĩa (UDF)

### Cập nhật dữ liệu dùng Stored Procedure:

1. Tạo lớp truy vấn dữ liệu (chưa dùng đến các thủ tục)
2. Dùng lớp truy cập dữ liệu (chưa sử dụng SPROC)
3. Thêm Order bằng cách dung SPROC:
4. Thực hiện cập nhật dùng SPROC:
5. Dùng Lớp DAL lần nữa:
6. Ưu điểm của Việc dùng SPROC:

### Thực thi các biểu thức SQL tùy biến:

1. Dùng các câu truy vấn SQL tùy biến với LINQ to SQL:
2. Dùng ExecuteQuery:
3. Tùy biến các biểu thức SQL và lưu vết (Tracking) các thao tác cập nhật:
4. Tùy biến biểu thức SQL với các lớp trong phần mềm:
5. Tùy biến các câu SQL cho Insert/Update/Delete:

## Tổng kết:

Bạn có thể cung cấp (cho hệ thống) một kết nối ADO.NET đã tồn tại khi bạn tạo ra một LINQ to SQL DataContext. Tất cả các hoạt động dựa vào DataContext (bao gồm cả các truy vấn) đều sử dụng kết nối đã được cung cấp này. Nếu kết nối đã được mở, LINQ to SQL vẫn sẽ để nó như vậy khi bạn đã xong việc với nó.

string connString = @"Data Source=.\SQLEXPRESS;AttachDbFilename=c:\northwind.mdf;

Integrated Security=True; Connect Timeout=30; User Instance=True";

SqlConnection nwindConn = new SqlConnection(connString);

nwindConn.Open();

Northwnd interop\_db = new Northwnd(nwindConn);

SqlTransaction nwindTxn = nwindConn.BeginTransaction();

try

{

SqlCommand cmd = new SqlCommand(

"UPDATE Products SET QuantityPerUnit = 'single item' WHERE ProductID = 3");

cmd.Connection = nwindConn;

cmd.Transaction = nwindTxn;

cmd.ExecuteNonQuery();

interop\_db.Transaction = nwindTxn;

Product prod1 = interop\_db.Products

.First(p => p.ProductID == 4);

Product prod2 = interop\_db.Products

.First(p => p.ProductID == 5);

prod1.UnitsInStock -= 3;

prod2.UnitsInStock -= 5;

interop\_db.SubmitChanges();

nwindTxn.Commit();

}

catch (Exception e)

{

Console.WriteLine(e.Message);

Console.WriteLine("Error submitting changes... all changes rolled back.");

}

nwindConn.Close();

Bạn luôn luôn có thể truy cập vào kết nối và đóng nó bằng cách sử dụng các đặc tính của Connection như sau:

db.Connection.Close();

## Giao dịch:

Bạn có thể cung cấp cho DataContext bằng giao dịch từ CSDL của chính mình khi các ứng dụng của bạn đã khởi động khả năng giao dịch và bạn muốn DataContext tham gia vào.

Phương pháp được thông dụng nhất để thực hiện giao dịch .NET Framework là sử dụng những đối tượng TransactionScope. Bằng cách sử dụng phương pháp tiếp cận này, bạn có thể thực hiện các giao dịch trên nhiều cơ sở dữ liệu và các tài nguyên bộ nhớ khác. TransactionScope yêu cầu rất ít tài nguyên để khởi động. Chúng chỉ bắt đầu phân bổ các giao dịch chỉ khi có nhiều kết nối trong phạm vi giao dịch.

using (TransactionScope ts = new TransactionScope())

{

db.SubmitChanges();

ts.Complete();

}

Bạn không thể sử dụng phương pháp tiếp cận này cho tất cả các cơ sở dữ liệu. Ví dụ, các kết nối SqlClient không thể hoạt động với các giao dịch hệ thống vì nó hoạt động dựa trên một máy chủ SQL Server 2000. Thay vào đó, nó sẽ tự động tham gia vào một giao dịch đầy đủ và đã được phân phối chức năng liên quan ngay khi thấy giao dịch này được sử dụng.

## Lệnh gọi SQL trực tiếp:

Đôi khi bạn có thể gặp tình huống mà DataContext không đủ khả năng để truy vấn hoặc gửi đi các thay đổi theo đúng yêu cầu chuyên môn mà bạn muốn thực hiện. Trong những trường hợp đó, bạn có thể sử dụng các phương pháp ExecuteQuery để thực hiện trực tiếp các lệnh SQL cho cơ sở dữ liệu và chuyển đổi kết quả truy vấn vào các đối tượng.

Ví dụ, giả định rằng dữ liệu của các Customer nằm trên hai bảng (Customer 1 và Customer 2). truy vấn sau đây sẽ trả về một kết quả cho đối tượng Customer:

IEnumerable<Customer> results = db.ExecuteQuery<Customer>(

@"select c1.custid as CustomerID, c2.custName as ContactName

from customer1 as c1, customer2 as c2

where c1.custid = c2.custid"

);

Chỉ cần các tên cột trong các kết quả phù hợp với các thuộc tính cột trong lớp thực thể của bạn, LINQ to SQL sẽ tạo ra được các đối tượng từ truy vấn SQL.

## Các tham số:

Phương pháp ExecuteQuery chấp nhận các tham số. Đoạn code sau đây thực thi truy vấn có tham số:

IEnumerable<Customer> results = db.ExecuteQuery<Customer>(

"select contactname from customers where city = {0}",

"London"

);

LINQ to SQL sẽ tạo ra được các đối tượng từ truy vấn SQL.

## Cách kết nối một CSDL (LINQ to SQL):

DataContext là đường dẫn chính để bạn kết nối, lấy dữ liệu cho các đối tượng, và cập nhật dữ liệu mới trên một CSDL. Bạn chỉ cần sử dụng DataContext tương tự như khi bạn sử dụng ADO.NET SqlConnection. Trong thực tế, các DataContext được khởi tạo với một kết nối hoặc chuỗi kết nối mà bạn cung cấp.

Mục đích của DataContext là để dịch các yêu cầu của bạn cho các đối tượng thành các truy vấn của SQL để làm việc với các cơ sở dữ liệu, và sau đó thu thập các đối tượng từ các kết quả. DataContext kết hợp với LINQ bằng cách lấy cùng những toán tử như nhau để làm chuẩn, chẳng hạn như “Where” và “Select”.

Trong ví dụ sau, DataContext được sử dụng để kết nối với cơ sở dữ liệu mẫu Northwind và để lấy thông tin của những khách hàng là người sống ở London

// DataContext takes a connection string.

DataContext db = new DataContext(@"c:\Northwind.mdf");

// Get a typed table to run queries.

Table<Customer> Customers = db.GetTable<Customer>();

// Query for customers from London.

var query =

from cust in Customers

where cust.City == "London"

select cust;

foreach (var cust in query)

Console.WriteLine("id = {0}, City = {1}", cust.CustomerID, cust.City);

Mỗi bảng trong CSDL được biểu điễn bởi một Table trong danh sách cho phép truy vấn bằng phương thức GetTable bằng cách sử dụng lớp thực thể để xác định nó.

Phương pháp tốt nhất là khai báo một kiểu DataContext thay vì dựa trên lớp DataContext cơ bản và phương pháp GetTable. Kiểu DataContext khai báo tập hợp tất cả các Table, như một thành phần của ngữ cảnh theo ví dụ sau:

public partial class Northwind : DataContext

{

public Table<Customer> Customers;

public Table<Order> Orders;

public Northwind(string connection) : base(connection) { }

}

Sau đó bạn có thể thực hiên truy vấn khách hàng từ London đơn giản hơn bằng cách sau:

Northwnd db = new Northwnd(@"c:\Northwnd.mdf");

var query =

from cust in db.Customers

where cust.City == "London"

select cust;

foreach (var cust in query)

Console.WriteLine("id = {0}, City = {1}", cust.CustomerID,

cust.City);

LINQ to SQL sẽ tạo ra được các đối tượng từ truy vấn SQL.

## Cách tự động tạo ra một CSDL (LINQ to SQL):

Trong LINQ to SQL, các đối tượng được ánh xạ vào một CSDL quan hệ. Sự ánh xạ được kích hoạt bằng cách ánh xạ các đặc tính tượng trưng hoặc sử dụng các tệp ánh xạ ngoài để mô tả cấu trúc của CSDL quan hệ. Cả 2 phương pháp này đều sẽ cung cấp đầy đủ thông tin về CSDL quan hệ để bạn có thể tạo ra một phiên bản của CSDL sử dụng phương thức DataContext.CreateDatabase.

Phương thức DataContext.CreateDatabase tạo một bản sao của cơ sở dữ liệu nằm trong phạm vi thông tin được mã hóa trong mô hình đối tượng. Ánh xạ tệp và thuộc tính từ mô hình đối tượng của bạn có thể sẽ không mã hóa mọi thứ về cấu trúc của cơ sở dữ liệu hiện có. Thông tin ánh xạ sẽ không thể bao gồm nội dung của các hàm do người dùng xác định, các hàm, thủ tục lưu trữ, trigger hoặc khả năng kiểm tra các ràng buộc.

Bạn có thể sử dụng phương thức DataContext. CreateDatabase trong mọi trường hợp, đặc biệt là nếu bên cung cấp dữ liệu có Microsoft SQL Server 2008 khả dụng. Các kịch bản điển hình bao gồm:

* Bạn đang xây dựng một ứng dụng có thể tự động cài đặt trên hệ thống của khách hàng.
* Bạn đang xây dựng một ứng dụng khách cần cơ sở dữ liệu cục bộ để lưu các trạng thái ngoại tuyến của nó.

Bạn cũng có thể sử dụng phương thức DataContext.CreateDatabase với SQL Server bằng cách sử dụng tệp .mdf hoặc tên thay thế, tùy theo chuỗi kết nối của bạn. LINQ to SQL sử dụng chuỗi kết nối để xác định CSDL nguồn và CSDL nào được tạo.

Ví dụ: Đoạn code sau đây cung cấp một ví dụ về cách bạn sẽ tạo ra một cơ sở dữ liệu mới có tên MyDVDs.mdf.

public class MyDVDs : DataContext

{

public Table<DVD> DVDs;

public MyDVDs(string connection) : base(connection) { }

}

[Table(Name = "DVDTable")]

public class DVD

{

[Column(IsPrimaryKey = true)]

public string Title;

[Column]

public string Rating;

}

Bạn cũng có thể sử dụng mô hình đối tượng để tạo CSDL như sau:

public void CreateDatabase()

{

MyDVDs db = new MyDVDs("c:\\mydvds.mdf");

db.CreateDatabase();

}

Khi bạn xây dựng một ứng dụng có thể tự động cài đặt lên hệ thống của khách hàng, hãy xem liệu CSDL của ứng dụng đã tồn tại hay chưa và xóa nó trước khi tạo một CSDL mới. Lớp DataContext cung cấp các phương thức DatabaseExists và DeleteDatabase để giúp bạn trong quá trình này.

public void CreateDatabase2()

{

MyDVDs db = new MyDVDs(@"c:\mydvds.mdf");

if (db.DatabaseExists())

{

Console.WriteLine("Deleting old database...");

db.DeleteDatabase();

}

db.CreateDatabase();

}

## Bạn có thể làm gì với LINQ to SQL:

LINQ to SQL cung cấp tất cả các từ khóa cần thiết dành cho một lập trình viên SQL. Bạn có thể truy vấn dữ liệu và thêm, sửa, xóa các dữ liệu trong bảng.

### Lựa chọn (Selecting):

Là viết một truy vấn LINQ trong ngôn ngữ lập trình của mình và sau đó xử lý truy vấn để lấy các kết quả mong muốn. LINQ to SQL tự động dịch tất cả các hoạt động cần thiết thành các truy vấn SQL cần thiết mà bạn có thể hiểu được.

Ví dụ sau đây sẽ thực hiện lấy tên công ty của một khách hàng đến từ London và hiển thị nó lên màn hình Console

// Northwnd inherits from System.Data.Linq.DataContext.

Northwnd nw = new Northwnd(@"northwnd.mdf");

// or, if you are not using SQL Server Express

// Northwnd nw = new Northwnd("Database=Northwind;Server=server\_name;Integrated Security=SSPI");

var companyNameQuery =

from cust in nw.Customers

where cust.City == "London"

select cust.CompanyName;

foreach (var customer in companyNameQuery)

{

Console.WriteLine(customer);

}

### Thêm (Inserting):

Để thực hiện một truy vấn Insert như trong SQL, chỉ cần thêm các đối tượng vào mô hình đối tượng bạn đã tạo, và gọi lệnh SubmitChanges trên DataContext.

Trong ví dụ sau, một khách hàng mới và các thông tin về các khách hàng sẽ được thêm vào bảng Customers bằng cách sử dụng lệnh InsertOnSubmit.

// Northwnd inherits from System.Data.Linq.DataContext.

Northwnd nw = new Northwnd(@"northwnd.mdf");

Customer cust = new Customer();

cust.CompanyName = "SomeCompany";

cust.City = "London";

cust.CustomerID = "98128";

cust.PostalCode = "55555";

cust.Phone = "555-555-5555";

nw.Customers.InsertOnSubmit(cust);

// At this point, the new Customer object is added in the object model.

// In LINQ to SQL, the change is not sent to the database until

// SubmitChanges is called.

nw.SubmitChanges();

### Cập nhật (Updating):

Để Update cơ sở dữ liệu, đầu tiên bạn cần truy xuất mục cần thay đổi và chỉnh sửa trực tiếp trong mô hình đối tượng. Sau khi bạn đã sửa đổi đối tượng, hãy gọi SendChanges trên DataContext để cập nhật vào CSDL.

Trong ví dụ sau, tất cả các khách hàng đến từ London đều được truy xuất. Sau đó, tên của thành phố được đổi từ "London" sang "London - Metro". Cuối cùng, SendChanges được gọi để gửi các thay đổi đến CSDL.

Northwnd nw = new Northwnd(@"northwnd.mdf");

var cityNameQuery =

from cust in nw.Customers

where cust.City.Contains("London")

select cust;

foreach (var customer in cityNameQuery)

{

if (customer.City == "London")

{

customer.City = "London - Metro";

}

}

nw.SubmitChanges();

### Xóa (Deleting):

Để Delete một mục, hãy xóa mục đó khỏi danh sách đối tượng và sau đó gọi SendChanges trên DataContext để thực hiện thay đổi.

Lưu ý: LINQ to SQL không thể nhận ra các ràng buộc liên quan khi xóa một mục. Nếu bạn muốn xóa một hàng trong bảng mà có các ràng buộc liên quan đến nó hãy xem lại phần “Chỉ định lưu trữ Thực hiện thủ tục Update, Insert, và delete”.

Trong ví dụ sau, khách hàng có “CustomerID” là “98128” được lấy từ CSDL. Sau đó, sau khi xác nhận rằng hàng của khách hàng đã được truy xuất vào đối tượng quan hệ, DeleteOnSubmit được gọi để xóa đối tượng đó khỏi danh sách. Cuối cùng, SendChanges được gọi để tạo yêu cầu xóa tại CSDL.

Northwnd nw = new Northwnd(@"northwnd.mdf");

var deleteIndivCust =

from cust in nw.Customers

where cust.CustomerID == "98128"

select cust;

if (deleteIndivCust.Count() > 0)

{

nw.Customers.DeleteOnSubmit(deleteIndivCust.First());

nw.SubmitChanges();

}

## Thủ tục lưu trữ (Stored Procedures):

LINQ to SQL sử dụng các phương thức trong mô hình đối tượng để thể hiện các thủ tục lưu trữ vào CSDL. Bạn sẽ chỉ định các phương thức và các thủ tục lưu trữ bằng cách áp dụng thuộc tính FunctionAttribution và thuộc tính ParameterAttribution nếu cần thiết.

Các Lập trình viên sử dụng Visual Studio thường sử dụng Object Relational Designer để ánh xạ các thủ tục lưu trữ.

### Trả về RowSets:

Khi bạn thực hiện một Stored Procedure trả về một rowset, bạn phải sử dụng một lớp kết quả lưu trữ giá trị trả về từ Stored Procedure đó.

Ví dụ sau trả về một rowset từ CSDL và có một tham số đầu vào để lọc kết quả, tham số đầu vào sẽ lọc ra những khách hàng có “London” là thành phố. Ví dụ có chứa lớp Enumerable CustomersByCityResult

CREATE PROCEDURE [dbo].[Customers By City]

(@param1 NVARCHAR(20))

AS

BEGIN

-- SET NOCOUNT ON added to prevent extra result sets from

-- interfering with SELECT statements.

SET NOCOUNT ON;

SELECT CustomerID, ContactName, CompanyName, City from Customers

as c where c.City=@param1

END

[Function(Name="dbo.Customers By City")]

public ISingleResult<CustomersByCityResult> CustomersByCity([Parameter(DbType="NVarChar(20)")] string param1)

{

IExecuteResult result = this.ExecuteMethodCall(this, ((MethodInfo)(MethodInfo.GetCurrentMethod())), param1);

return ((ISingleResult<CustomersByCityResult>)(result.ReturnValue));

}

// Call the stored procedure.

void ReturnRowset()

{

Northwnd db = new Northwnd(@"c:\northwnd.mdf");

ISingleResult<CustomersByCityResult> result =

db.CustomersByCity("London");

foreach (CustomersByCityResult cust in result)

{

Console.WriteLine("CustID={0}; City={1}", cust.CustomerID,

cust.City);

}

}

### Sử dụng Stored Procedures với tham số đầu vào:

LINQ to SQL ánh xạ các tham số đầu ra thành các tham số tham chiếu và khai báo kiểu dữ liệu của các tham số này là Nullable.

Ví dụ sau nhận một tham số đầu vào (ID khách hàng) và trả về một tham số (tổng doanh số cho khách hàng đó).

CREATE PROCEDURE [dbo].[CustOrderTotal]

@CustomerID nchar(5),

@TotalSales money OUTPUT

AS

SELECT @TotalSales = SUM(OD.UNITPRICE\*(1-OD.DISCOUNT) \* OD.QUANTITY)

FROM ORDERS O, "ORDER DETAILS" OD

where O.CUSTOMERID = @CustomerID AND O.ORDERID = OD.ORDERID

[Function(Name="dbo.CustOrderTotal")]

[return: Parameter(DbType="Int")]

public int CustOrderTotal([Parameter(Name="CustomerID", DbType="NChar(5)")] string customerID, [Parameter(Name="TotalSales", DbType="Money")] ref System.Nullable<decimal> totalSales)

{

IExecuteResult result = this.ExecuteMethodCall(this, ((MethodInfo)(MethodInfo.GetCurrentMethod())), customerID, totalSales);

totalSales = ((System.Nullable<decimal>)(result.GetParameterValue(1)));

return ((int)(result.ReturnValue));

}

// Call the stored procedure.

Northwnd db = new Northwnd(@"c:\northwnd.mdf");

decimal? totalSales = 0;

db.CustOrderTotal("alfki", ref totalSales);

Console.WriteLine(totalSales);

### Sử dụng Stored Procedure ánh xạ cho nhiều dạng kết quả:

Khi một Stored Procedures có thể trả về nhiều dạng kết quả, kiểu trả về không thể phụ thuộc vào một phép chiếu. Mặc dù LINQ to SQL có thể tạo ra tất cả các loại phép chiếu có thể, nhưng nó không thể biết thứ tự chúng sẽ được trả về. Mặc dù LINQ to SQL có thể giải quyết tất cả các phép chiếu có thể, nhưng nó cũng không thể biết được thứ tự trả về của chúng

Trong ví dụ sau, dạng kết qủa phụ thuộc vào đầu vào (shape = 1 hay shape = 2). Bạn sẽ không biết rằng phép chiếu nào sẽ hoàn thành trước.

CREATE PROCEDURE VariableResultShapes(@shape int)

AS

if(@shape = 1)

select CustomerID, ContactTitle, CompanyName from customers

else if(@shape = 2)

select OrderID, ShipName from orders

[Function(Name="dbo.VariableResultShapes")]

[ResultType(typeof(VariableResultShapesResult1))]

[ResultType(typeof(VariableResultShapesResult2))]

public IMultipleResults VariableResultShapes([Parameter(DbType="Int")] System.Nullable<int> shape)

{

IExecuteResult result = this.ExecuteMethodCall(this, ((MethodInfo)(MethodInfo.GetCurrentMethod())), shape);

return ((IMultipleResults)(result.ReturnValue));

}

Northwnd db = new Northwnd(@"c:\northwnd.mdf");

// Assign the results of the procedure with an argument

// of (1) to local variable 'result'.

IMultipleResults result = db.VariableResultShapes(1);

// Iterate through the list and write results (the company names)

// to the console.

foreach(VariableResultShapesResult1 compName in

result.GetResult<VariableResultShapesResult1>())

{

Console.WriteLine(compName.CompanyName);

}

// Pause to view company names; press Enter to continue.

Console.ReadLine();

// Assign the results of the procedure with an argument

// of (2) to local variable 'result'.

IMultipleResults result2 = db.VariableResultShapes(2);

// Iterate through the list and write results (the order IDs)

// to the console.

foreach (VariableResultShapesResult2 ord in

result2.GetResult<VariableResultShapesResult2>())

{

Console.WriteLine(ord.OrderID);

}

# LINQ to Object:

using **System.Linq**; **System.Collections.Generic**;

Thuật ngữ "LINQ to Objects" đề cập đến việc sử dụng các truy vấn LINQ với bất kỳ tập hợp IEnumerable hay IEnumerable (T),

## Các khái niệm trong LINQ to Object:

### Object:

Obj là…..

Sau đây là các đối tượng khác nhau mà chúng ta có thể sử dụng với LINQ.

* Linq to Strings
* Linq to String Array
* Linq to Int Array
* Linq to Files
* Linq to Lists

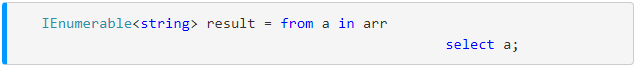
## Xây dựng chương trình với LINQ to Object:

### LINQ to String:

LINQ to String Array có nghĩa là viết các truy vấn LINQ trên mảng chuỗi để nhận dữ liệu cần thiết. Nếu chúng ta sử dụng truy vấn LINQ trên mảng chuỗi, chúng ta có thể dễ dàng nhận được các phần tử cần thiết mà không cần viết nhiều mã.

**Cú pháp**

Cú pháp truy vấn lấy ra phần tử của mảng string

****

**Ví dụ**

Lấy ra những phần tử từ mảng tuần tự string với tên bắt đầu bằng “s”

### :

## Tổng kết:

# Tổng quan về LinQ to XML:

|  |  |
| --- | --- |
| **Chủ đề** | **Miêu tả** |
| **Tổng quan LINQ cho XML (C#)** | Cung cấp tổng quan về công nghệ LINQ cho XML. |
| **LINQ cho XML so với DOM (C#)** | So sánh LINQ cho XML với Mô hình đối tượng tài liệu (DOM). |
| **LINQ cho XML so với Các công nghệ XML khác** | So sánh LINQ cho XML với các công nghệ phân tích và xử lý phân tích XML khác: XmlReader, XSLT, MSXML và XmlLite. |

XML đã được áp dụng rộng rãi như một cách để định dạng dữ liệu trong nhiều ngữ cảnh. Ví dụ: bạn có thể tìm thấy XML trên Web, trong các tệp cấu hình, trong các tệp Microsoft Office Word và trong cơ sở dữ liệu.

LINQ cho XML là một cách tiếp cận được cập nhật, được thiết kế lại để lập trình với XML. Nó cung cấp các khả năng sửa đổi tài liệu trong bộ nhớ của Mô hình đối tượng tài liệu (DOM) và hỗ trợ các biểu thức truy vấn LINQ. Mặc dù các biểu thức truy vấn này khác về mặt cú pháp so với XPath, chúng cung cấp chức năng tương tự.

**Các nhà phát triển LINQ to XML**

LINQ cho XML nhắm vào nhiều nhà phát triển. Đối với một nhà phát triển trung bình chỉ muốn hoàn thành công việc, LINQ cho XML giúp XML dễ dàng hơn bằng cách cung cấp trải nghiệm truy vấn tương tự như SQL. Chỉ cần một chút nghiên cứu, các lập trình viên có thể học cách viết các truy vấn ngắn gọn và mạnh mẽ bằng ngôn ngữ lập trình mà họ lựa chọn.

Các nhà phát triển chuyên nghiệp có thể sử dụng LINQ cho XML để tăng đáng kể năng suất của họ. Với LINQ cho XML, họ có thể viết ít mã hơn, biểu cảm hơn, gọn hơn và mạnh mẽ hơn. Họ có thể sử dụng các biểu thức truy vấn từ nhiều miền dữ liệu cùng một lúc.

**LINQ cho XML là gì?**

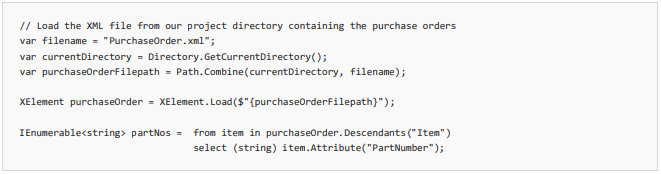
LINQ cho XML là giao diện lập trình XML trong bộ nhớ hỗ trợ LINQ cho phép bạn làm việc với XML từ bên trong các ngôn ngữ lập trình .NET Framework.

LINQ cho XML giống như Mô hình Đối tượng Tài liệu (DOM) ở chỗ nó đưa tài liệu XML vào bộ nhớ. Bạn có thể truy vấn và sửa đổi tài liệu, và sau khi bạn sửa đổi nó, bạn có thể lưu nó vào một tệp hoặc tuần tự hóa nó và gửi nó qua Internet. Tuy nhiên, LINQ cho XML khác với DOM: Nó cung cấp một mô hình đối tượng mới có trọng lượng nhẹ hơn và dễ làm việc hơn và tận dụng các tính năng ngôn ngữ trong C #.

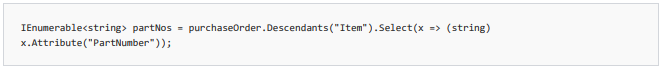
Ưu điểm quan trọng nhất của LINQ cho XML là tích hợp với Truy vấn tích hợp ngôn ngữ (LINQ). Việc tích hợp này cho phép bạn viết các truy vấn trên tài liệu XML trong bộ nhớ để truy xuất các bộ sưu tập các phần tử và thuộc tính. Khả năng truy vấn của LINQ cho XML có thể so sánh về chức năng (mặc dù không theo cú pháp) với XPath và XQuery. Việc tích hợp LINQ trong C # cung cấp khả năng gõ mạnh hơn, kiểm tra thời gian biên dịch và hỗ trợ trình gỡ lỗi được cải thiện.

Một ưu điểm khác của LINQ to XML là khả năng sử dụng các kết quả truy vấn làm tham số cho các hàm tạo đối tượng **XEuity** và **XAttribution** cho phép một cách tiếp cận mạnh mẽ để tạo cây XML. Cách tiếp cận này, được gọi là xây dựng chức năng, cho phép các nhà phát triển dễ dàng chuyển đổi các cây XML từ hình dạng này sang hình dạng khác.

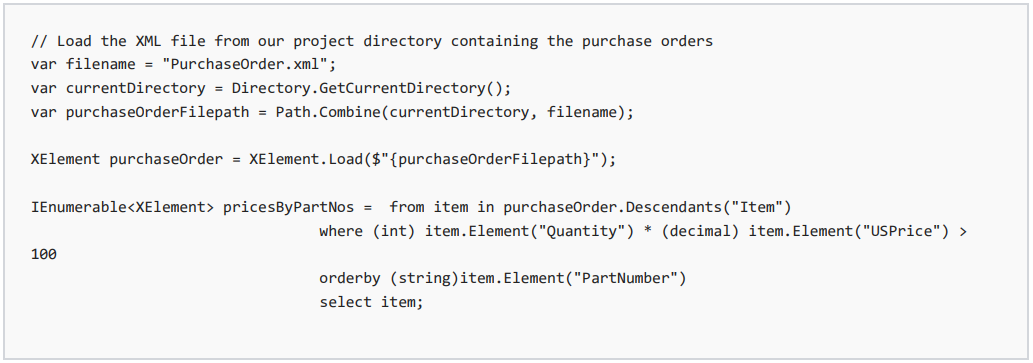
Ví dụ: bạn có thể có một đơn đặt hàng XML điển hình như được mô tả trong Tệp XML mẫu: Đơn đặt hàng tiêu biểu (LINQ to XML). Bằng cách sử dụng LINQ cho XML, bạn có thể chạy truy vấn sau để có được giá trị thuộc tính số phần cho mọi thành phần vật phẩm trong đơn đặt hàng:



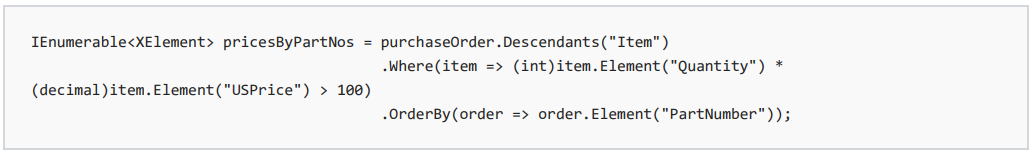
Viết lại dưới dạng cú pháp phương thức:

****

Một ví dụ khác, bạn có thể muốn một danh sách, được sắp xếp theo số phần, của các mục có giá trị lớn hơn $ 100. Để có được thông tin này, bạn có thể chạy truy vấn sau:

****

Một lần nữa, điều này có thể được viết lại dưới dạng cú pháp phương thức:

****

Ngoài các khả năng LINQ này, LINQ to XML cung cấp giao diện lập trình XML được cải tiến. Sử dụng LINQ to XML, bạn có thể:

* Tải XML từ **Files** và **streams.**
* Nối tiếp XML thành **Files** và **streams.**
* Tạo XML từ đầu bằng cách sử dụng chức năng xây dựng.
* Truy vấn XML bằng các trục giống như XPath.
* Thao tác với cây XML trong bộ nhớ bằng cách sử dụng các phương thức như **Add**, **Remove**, **ReplaceWith** và **SetValue**.
* Xác thực cây XML bằng XSD.
* Sử dụng kết hợp các tính năng này để chuyển đổi cây XML từ hình dạng này sang hình dạng khác.

**Tạo cây XML**

Một trong những lợi thế quan trọng nhất của việc lập trình với LINQ cho XML là dễ dàng tạo các cây XML. Ví dụ: để tạo một cây XML nhỏ, bạn có thể viết mã như sau:



# LINQ to XML so với DOM (C #):

**Những cách mới để xây dựng cây XML**

Trong DOM W3C, bạn xây dựng cây XML từ dưới lên; nghĩa là bạn tạo một tài liệu, bạn tạo các phần tử và sau đó bạn thêm các phần tử vào tài liệu.

Ví dụ, sau đây sẽ là một cách điển hình để tạo cây XML bằng cách sử dụng triển khai DOM, **XmlDocument** của Microsoft:



Kiểu mã hóa này không cung cấp trực quan nhiều thông tin về cấu trúc của cây XML. LINQ to XML hỗ trợ cách tiếp cận này để xây dựng cây XML, nhưng cũng hỗ trợ một cách tiếp cận khác, xây dựng chức năng. Xây dựng chức năng sử dụng các hàm tạo **XEuity** và **XAttribution** để xây dựng cây XML.

Đây là cách bạn sẽ xây dựng cùng một cây XML bằng cách sử dụng chức năng xây dựng chức năng LINQ to XML:



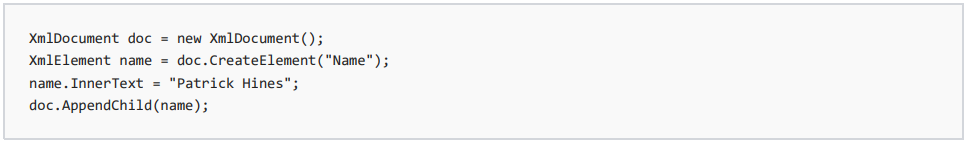
Lưu ý rằng việc thụt lề mã để xây dựng cây XML cho thấy cấu trúc của XML bên dưới. Để biết thêm thông tin, hãy xem **Tạo Cây XML (C #).**

**Làm việc trực tiếp với các phần tử XML**

Khi bạn lập trình với XML, trọng tâm chính của bạn thường là các phần tử XML và có lẽ là các thuộc tính. Trong LINQ to XML, bạn có thể làm việc trực tiếp với các thành phần và thuộc tính XML. Ví dụ: bạn có thể làm như sau:

* Tạo các phần tử XML mà không sử dụng một đối tượng tài liệu nào cả. Điều này đơn giản hóa việc lập trình khi bạn phải làm việc với các đoạn cây XML.
* Tải các đối tượng T: System.Xml.Linq.XElament trực tiếp từ tệp XML.
* Nối tiếp các đối tượng T: System.Xml.Linq.XElement vào một tệp hoặc một luồng.

So sánh điều này với W3C DOM, trong đó tài liệu XML được sử dụng làm vùng chứa logic cho cây XML.Trong DOM, các nút XML, bao gồm các thành phần và thuộc tính, phải được tạo trong ngữ cảnh của tài liệu XML. Đây là một đoạn mã để tạo thành phần tên trong DOM:

****

Nếu bạn muốn sử dụng một yếu tố trên nhiều tài liệu, bạn phải nhập các nút trên các tài liệu. LINQ to XML tránh lớp phức tạp này.

Khi sử dụng LINQ to XML, bạn chỉ sử dụng lớp **XDocument** nếu bạn muốn thêm một nhận xét hoặc hướng dẫn xử lý ở cấp gốc của tài liệu.

**Xử lý đơn giản tên và không gian tên**

Xử lý tên, không gian tên và tiền tố không gian tên nói chung là một phần phức tạp của lập trình XML. LINQ to XML đơn giản hóa các tên và không gian tên bằng cách loại bỏ yêu cầu xử lý các tiền tố không gian tên. Nếu bạn muốn kiểm soát tiền tố không gian tên, bạn có thể. Nhưng nếu bạn quyết định không kiểm soát rõ ràng các tiền tố không gian tên, LINQ to XML sẽ gán các tiền tố không gian tên trong quá trình tuần tự hóa nếu chúng được yêu cầu hoặc sẽ tuần tự hóa bằng cách sử dụng các không gian tên mặc định nếu không. Nếu không gian tên mặc định được sử dụng, sẽ không có tiền tố không gian tên trong tài liệu kết quả. Để biết thêm thông tin, hãy xem **Làm việc với Không gian tên XML (C #).**

Một vấn đề khác với DOM là nó không cho phép bạn thay đổi tên của một nút. Thay vào đó, bạn phải tạo một nút mới và sao chép tất cả các nút con vào nó, làm mất danh tính nút gốc. LINQ to XML tránh được vấn đề này bằng cách cho phép bạn đặt thuộc tính **XName** trên một nút.

**Hỗ trợ phương thức tĩnh để tải XML**

LINQ to XML cho phép bạn tải XML bằng cách sử dụng các phương thức tĩnh, thay vì các phương thức cá thể. Điều này đơn giản hóa việc tải và phân tích cú pháp. Để biết thêm thông tin, hãy xem Cách: **Tải XML từ một tệp (C #).**

**Xóa hỗ trợ cho các công trình xây dựng DTD**

LINQ to XML tiếp tục đơn giản hóa việc lập trình XML bằng cách loại bỏ hỗ trợ cho các thực thể và tham chiếu thực thể. Việc quản lý các thực thể rất phức tạp và hiếm khi được sử dụng. Loại bỏ hỗ trợ của họ làm tăng hiệu suất và đơn giản hóa giao diện lập trình. Khi cây LINQ to XML được điền, tất cả các thực thể DTD được mở rộng.

**Hỗ trợ cho các Fragment**

LINQ to XML không cung cấp tương đương cho lớp XmlDocumentFragment. Tuy nhiên, trong nhiều trường hợp, khái niệm XmlDocumentFragment có thể được xử lý bằng kết quả của một truy vấn được nhập cho **IEnumerable <T>** của **XNode** hoặc **IEnumerable <T>** của **XEuity**.

**Hỗ trợ cho XPathNavigator**

LINQ to XML cung cấp hỗ trợ cho **XPathNavigator** thông qua các phương thức mở rộng trong không gian tên **System.Xml.XPath**. Để biết thêm thông tin, hãy xem **System.Xml.XPath.Extensions**.

**Hỗ trợ cho không gian trắng và thụt lề**

LINQ to XML xử lý khoảng trắng đơn giản hơn DOM. Một kịch bản phổ biến là đọc XML thụt lề, tạo cây XML trong bộ nhớ mà không có bất kỳ nút văn bản khoảng trắng nào (nghĩa là không bảo toàn khoảng trắng), thực hiện một số thao tác trên XML và sau đó lưu XML bằng cách thụt lề. Khi bạn tuần tự hóa XML với định dạng, chỉ có khoảng trắng đáng kể trong cây XML được giữ nguyên. Đây là hành vi mặc định cho LINQ to XML.

Một kịch bản phổ biến khác là đọc và sửa đổi XML đã được thụt lề có chủ ý. Bạn có thể không muốn thay đổi vết lõm này theo bất kỳ cách nào. Trong LINQ to XML, bạn có thể thực hiện điều này bằng cách duy trì khoảng trắng khi bạn tải hoặc phân tích cú pháp XML và vô hiệu hóa định dạng khi bạn tuần tự hóa XML.

LINQ to XML lưu trữ khoảng trắng dưới dạng nút **XText**, thay vì có loại nút **Whitespace** chuyên dụng, như DOM thực hiện.

**Hỗ trợ chú thích**

Các phần tử LINQ to XML hỗ trợ một bộ chú thích mở rộng. Điều này hữu ích để theo dõi thông tin linh tinh về một phần tử, chẳng hạn như thông tin lược đồ, thông tin về việc phần tử đó có bị ràng buộc với UI hay bất kỳ loại thông tin cụ thể nào khác của ứng dụng hay không. Để biết thêm thông tin, hãy xem **Chú thích LINQ to XML.**

**Hỗ trợ thông tin lược đồ**

LINQ to XML cung cấp hỗ trợ xác thực XSD thông qua các phương thức mở rộng trong không gian tên **System.Xml.Schema**. Bạn có thể xác thực rằng một cây XML tuân thủ XSD. Bạn có thể điền vào cây XML bằng infoset xác thực sau lược đồ (PSVI). Để biết thêm thông tin, hãy xem Cách: **Xác thực bằng XSD** và **Tiện ích mở rộng**.

# LINQ to XML so với các công nghệ XML khác:

Chúng ta sẽ so sánh LINQ với XML với các công nghệ XML sau: **XmlReader**, XSLT, MSXML và XmlLite. Thông tin này có thể giúp bạn quyết định sử dụng công nghệ nào.

Để so sánh LINQ với XML với Mô hình đối tượng tài liệu (DOM), hãy xem **LINQ với XML so với DOM (C #).**

**LINQ to XML so với XmlReader**

**XmlReader** là một trình phân tích cú pháp nhanh, không chuyển tiếp, bộ đệm.

LINQ to XML được triển khai trên **XmlReader** và chúng được tích hợp chặt chẽ. Tuy nhiên, bạn cũng có thể sử dụng **XmlReader**.

Ví dụ: giả sử bạn đang xây dựng một dịch vụ Web sẽ phân tích hàng trăm tài liệu XML mỗi giây và các tài liệu có cùng cấu trúc, nghĩa là bạn chỉ phải viết một triển khai mã để phân tích cú pháp XML. Trong trường hợp này, có lẽ bạn sẽ muốn sử dụng **XmlReader**.

Ngược lại, nếu bạn đang xây dựng một hệ thống phân tích nhiều tài liệu XML nhỏ hơn và mỗi tài liệu khác nhau, bạn sẽ muốn tận dụng các cải tiến năng suất mà LINQ sang XML cung cấp.

**LINQ to XML so với XSLT**

Cả LINQ sang XML và XSLT đều cung cấp các khả năng chuyển đổi tài liệu XML mở rộng. XSLT là một cách tiếp cận khai báo dựa trên quy tắc. Các lập trình viên XSLT nâng cao viết XSLT theo kiểu lập trình chức năng nhấn mạnh cách tiếp cận không trạng thái. Các biến đổi có thể được viết bằng các hàm thuần túy được thực hiện mà không có tác dụng phụ. Cách tiếp cận dựa trên quy tắc hoặc chức năng này không quen thuộc với nhiều nhà phát triển và có thể khó học và tốn thời gian để tìm hiểu.

XSLT có thể là một hệ thống rất năng suất mang lại các ứng dụng hiệu suất cao. Ví dụ: một số công ty web lớn sử dụng XSLT như một cách để tạo HTML từ XML được lấy từ nhiều kho dữ liệu khác nhau. Công cụ XSLT được quản lý biên dịch mã XSLT thành CLR và thậm chí còn hoạt động tốt hơn trong một số trường hợp so với công cụ XSLT gốc. Tuy nhiên, XSLT không tận dụng kiến thức C # và Visual Basic mà nhiều nhà phát triển có. Nó yêu cầu các nhà phát triển viết mã bằng một ngôn ngữ lập trình phức tạp và khác biệt. Sử dụng hai hệ thống phát triển không tích hợp như C # (hoặc Visual Basic) và XSLT dẫn đến các hệ thống phần mềm khó phát triển và bảo trì hơn.

Sau khi bạn đã thành thạo các biểu thức truy vấn LINQ to XML, các phép biến đổi LINQ sang XML là một công nghệ mạnh mẽ, dễ sử dụng. Về cơ bản, bạn hình thành tài liệu XML của mình bằng cách sử dụng chức năng xây dựng, lấy dữ liệu từ nhiều nguồn khác nhau, xây dựng các đối tượng **Xlement** một cách linh hoạt và lắp ráp toàn bộ thành một cây XML mới. Việc chuyển đổi có thể tạo ra một tài liệu hoàn toàn mới. Xây dựng các phép biến đổi trong LINQ thành XML tương đối dễ dàng và trực quan và mã kết quả có thể đọc được. Điều này làm giảm chi phí phát triển và bảo trì.

LINQ to XML không nhằm thay thế XSLT. XSLT vẫn là công cụ được lựa chọn cho các phép biến đổi XML phức tạp và tập trung vào tài liệu, đặc biệt nếu cấu trúc của tài liệu không được xác định rõ.

XSLT có lợi thế là tiêu chuẩn của World Wide Web Consortium (W3C). Nếu bạn có một yêu cầu là bạn chỉ sử dụng các công nghệ là tiêu chuẩn, XSLT có thể phù hợp hơn.

XSLT là XML và do đó có thể được lập trình thao tác.

**LINQ to XML so với MSXML**

MSXML là công nghệ dựa trên COM để xử lý XML được bao gồm trong Microsoft Windows. MSXML cung cấp một triển khai nguyên gốc của DOM với sự hỗ trợ cho XPath và XSLT. Nó cũng chứa bộ phân tích cú pháp dựa trên sự kiện, bộ nhớ đệm SAX2.

MSXML hoạt động tốt, được bảo mật theo mặc định trong hầu hết các kịch bản và có thể được truy cập trong Internet Explorer để thực hiện xử lý XML phía máy khách trong các ứng dụng kiểu AJAX. MSXML có thể được sử dụng từ bất kỳ ngôn ngữ lập trình nào hỗ trợ COM, bao gồm C ++, JavaScript và Visual Basic 6.0.

MSXML không được khuyến nghị sử dụng trong mã được quản lý dựa trên thời gian chạy ngôn ngữ chung (CLR).

**LINQ to XML so với XmlLite**

XmlLite là một bộ đệm không lưu, chỉ chuyển tiếp, kéo trình phân tích cú pháp. Các nhà phát triển chủ yếu sử dụng XmlLite với C ++. Nhà phát triển không nên sử dụng XmlLite với mã được quản lý.

Ưu điểm chính của XmlLite là nó là một trình phân tích cú pháp XML nhẹ, nhanh, an toàn trong hầu hết các tình huống. Diện tích bề mặt đe dọa của nó là rất nhỏ. Nếu bạn phải phân tích các tài liệu không đáng tin cậy và bạn muốn bảo vệ chống lại các cuộc tấn công như từ chối dịch vụ hoặc tiếp xúc với dữ liệu, XmlLite có thể là một lựa chọn tốt.

XmlLite không được tích hợp với Truy vấn tích hợp ngôn ngữ (LINQ). Nó không mang lại sự cải thiện năng suất lập trình viên là động lực thúc đẩy đằng sau LINQ.

# Tài liệu tham khảo:

* Lịch sử của truy vấn tích hợp ngôn ngữ: ***Data Access API of the Day****,**Mike Pizzo* <https://blogs.msdn.microsoft.com/data/2006/12/05/data-access-api-of-the-day-part-i/>
* MSDN .Net Language-Integrated Query: <https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/concepts/linq/>
* Hướng dẫn khởi tạo và sử dụng Object Relational Designer(O/R Designer): <https://docs.microsoft.com/en-us/previous-versions/bb384429(v=vs.120)>
* LINQ to SQL cơ bản: <https://yinyangit.wordpress.com/2011/08/11/linq-to-sql-basic-concepts-object-relational-mapping-entity-class-association-and-datacontext/>
* MSDN LINQ to SQL: <https://docs.microsoft.com/en-us/dotnet/framework/data/adonet/sql/linq/>
* Mẫu truy vấn LINQ: <https://code.msdn.microsoft.com/101-LINQ-Samples-3fb9811b>
* ***C# in Depth Third Edition****, JohnSkeet*
* ***Practical Database Programming With Visual C#.NET***, *Ying Bai*
* ***Professional LINQ****, Scott Klein*