Introduction

LINQ = Language INtegrated Query

= new features that was added to C#3, VB9 and .NET Framework 3.5 for querying databases and local collections

Brings static type safety to database queries

Simple and composable

A universal querying language that can work across SQL, XML, local collections and third-party APIs such as SharePoint
Proliferation of Querying APIs

**SQL**
```
select * from customer where FirstName = 'Jim'
```

**XPath**
```
customers/customer[FirstName='Jim']
```

**C# 2.0**
```
Array.Find (customers, delegate (Customer c) { return c.FirstName == "Jim"; })
```

**CAML**
```
<Query>
  <Where>
    <Eq>
      <FieldRef Name="FirstName"/>
      <Value Type="Text">Jim</Value>
    </Eq>
  </Where>
</Query>
```

What’s wrong with SQL?

- Lack of static type checking in embedded SQL queries
  ```
  new SqlCommand ("select * from Customer where Name=@p0");
  ```
- Awkward to dynamically compose queries
- Plumbing code in parameterization & marshalling data
- Difficulty in working with hierarchical data
- Has not been redesigned in decades
Pagination

SELECT TOP 20 UPPER(Customer.Name)
FROM Customer
WHERE (NOT (EXISTS (
    SELECT NULL
    FROM ( 
        SELECT TOP 40 ID
        FROM Customer c1
        WHERE c1.Name LIKE 'A%' 
        ORDER BY c1.Name 
    ) AS c2
    WHERE Customer.ID = c2.ID 
))) AND (Customer.Name LIKE 'A%')
ORDER BY Customer.Name

How does LINQ do better?

```csharp
var query =
    from c in db.Customers
    where c.Name.StartsWith ("A")
    orderby c.Name
    select c.Name.ToUpper();
var thirdPage = query.Skip(40).Take(20);
```

- Simplicity
- Static type safety
- Composability (thanks to deferred execution)

*Query syntax is syntactic sugar.*
Compiler Translation

```csharp
var query = db.Customers
            .Where (c => c.Name.StartsWith ("A"))
            .OrderBy (c => c.Name)
            .Select (c => c.Name.ToUpper());
var thirdPage = query.Skip (40).Take (20);
```

The `db` variable is a window into an object relational mapper.
Creating a DataContext

```csharp
db = new MyDB ("connection string");

var query = db.Customers
    .Where (c => c.Name.StartsWith ("A"))
    .OrderBy (c => c.Name)
    .Select (c => c.Name.ToUpper());

var thirdPage = query
    .Skip (40)
    .Take (20);
```

Typed DataContext

```csharp
public class MyDB : DataContext
{
    public Table<Customer> Customers
    {
        get { return GetTable<Customer>(); } }
}

[Table]
public class Customer
{
    [Column(IsPrimaryKey=true)]
    public int ID;

    [Column]
    public string Name;

    [Association (OtherKey="CustomerID")]
    public EntitySet<Purchase> Purchases = new EntitySet<Purchase>();
}
```
Object Relational Mappers allow Associations

```csharp
[Table]
public class Purchase
{
    [Column(IsPrimaryKey=true)]
    public int ID;

    [Column]
    public int CustomerID;

    [Column]
    public string Description;

    [Column]
    public decimal Price;

    int EntityRef<Customer> custRef;

    [Association (Storage="custRef", ThisKey="CustomerID", IsForeignKey=true)]
    public Customer Customer
    {
        get { return custRef.Entity; } set { custRef.Entity = value; }
    }
}
```

Querying through Associations

```csharp
from c in db.Customers
where c.Purchases.Count() >= 2
select new
{
    c.Name,
    TotalSpend = c.Purchases.Sum (p => p.Price)
}
```
Previous Query, in One Step

```csharp
var thirdPage = db.Customers
    .Where (c => c.Name.StartsWith ("A"))
    .OrderBy (c => c.Name)
    .Select  (c => c.Name.ToUpper())
    .Skip  (40)
    .Take    (20);
```

`thirdPage` evaluates to an expression tree.

LINQ to SQL Queries are Expression Trees

![Expression Tree Diagram](Image)
var thirdPage = db.Customers
    .Where (c => c.Name.StartsWith("A"))
    .OrderBy (c => c.Name)
    .Select (c => c.Name)
    .Skip (40)
    .Take (20);

Set Operators

Concat, Union, Intersect, Except

db.Customers.Select (c => c.Name)
    .Union (db.Purchases.Select (p => p.Description))
The Join Operator

```
from c in db.Customers
join p in db.Purchases on c.ID equals p.CustomerID
```

The Group Operator

```
var grouped =
    from p in db.Purchases
group p by p.Date.Year;
```
The SelectMany Operator

```csharp
var flattened =
    from g in grouped
    from p in g
    select p;
```

```csharp
from c in db.Customers
from p in c.Purchases
select c.Name +
    " -- " +
    p.Description;
```
Element/Quantifiers/Aggregation Operations

Element Operators
First, Single

```
db.Customers.First (c => c.ID == 123);
```

Quantifiers
All, Any, Contains

```
bool anyInDebt =
    db.Customers.Any (c => c.Balance < 0);
```

Aggregation Operators
Aggregate, Average, Count, Sum, Max, Min

```
decimal totalBalance =
    db.Customers.Sum (c => c.Balance);
```

Lambda Expressions

```
from c in db.Customers
where c.Name.StartsWith("a")
select c
```

```
db.Customers.Where (c => c.Name.StartsWith("a"))
```
Lambda Expressions

Lambda expression

Sequence (or nested sequence) → Query Operator → Sequence (or nested sequence)

```
db.Customers.Where (c => c.Name.StartsWith("a"))
```

Subqueries

Sequence → Query Operator → Sequence → Lambda expression

Sequence (or nested sequence) → Query Operator → Sequence (or nested sequence)

```
```
Projecting Subsequences

Subqueries - Select

```csharp
from c in db.Customers
select new
{
    c.Name,
    HighValuePurchases =
        from p in c.Purchases
        where p.Price > 1000
        orderby p.Date
        select new { p.Description, p.Price }
}
```
from c in db.Customers
select new
{ c.Name,
  Purchases =
    from p in db.Purchases
    orderby p.Date
    select new { p.Description, p.Price }
}

Sample Queries

Preloaded in LINQPad:

www.linqpad.net
Collateral Damage

- Losses in translation
  - certain kinds of SQL query hard to achieve
    - workaround = table-value functions
  - locking and optimization hints impossible
- C# expressions with no SQL translation
- Limits in expression composability
  - workaround: [www.albahari.com/nutshell/extras.html](http://www.albahari.com/nutshell/extras.html)
- Mistaking local for interpreted queries
- Leaks in abstraction
  - local & LINQ to SQL queries may need to be formulated differently for maximum efficiency
- Performance cost
  - Conversion time
    - workaround = compiled queries & metamodel sharing
  - Non-optimal SQL
    - workaround = use SQL or SPs for those cases
- Updates that don’t involve retrieving data first

Verdict

- LINQ to SQL has more than halved the middle tier development time, in my own experience
- A LINQ to SQL middle tier is smaller, tidier and safer
- Mix and match where necessary: sometimes old-fashioned SQL is best
- The technology has further promise
  - Provider independence
  - LINQ to Entities
  - Third party Object Relational Mappers
Resources

MS LINQ Forum:
http://tinyurl.com/4y93ta

PredicateBuilder & LINQKit:
www.albahari.com/nutshell/extras.html

LINQPad:
www.linqpad.net

C# 3.0 in a Nutshell
• C# 3.0 Language
• CLR
• Core .NET Framework
• LINQ to Objects
• LINQ to SQL
• LINQ to XML

C# 3.0 Pocket Reference
• C# 3.0 Language
• LINQ: distilled summary

LINQ Pocket Reference
• Learn LINQ in 170 pages
• LINQ to Objects
• LINQ to SQL
• LINQ to XML

Joseph Albahari  www.albahari.com