LINQ

F. Marguerie et al, LINQ in Action, Manning, 2008
J. C. Rattz, Pro LINQ: Language Integrated Query in C# 2008, Apress, 2007
P. Kimmel, LINQ Unleashed for C#, SAMS, 2008
MSDN, 101 LINQ Samples
MSDN, LINQ to SQL: .NET Language-Integrated Query for Relational Data

LINQ – Language Integrated Query

- It unifies data access
- It allows for query and set operations, similar to what SQL statements offer for databases
- It integrates queries directly within .NET languages
- It was designed to be used against any type of object or data source and to provide a consistent programming model for doing so
- It allows to check queries at compile-time

LINQ Design Goals

- Integrate objects, relational data, and XML
- SQL and XQuery-like power in C# and VB
- Extensibility model for languages
- Extensibility model for multiple data sources
- Type safety
- Extensive IntelliSense support (enabled by strong-typing)
- Debugger support
- Build on the foundation laid in C# 1.0 and 2.0
- Run on the .NET 2.0 CLR
  - .NET 2.0 Service Pack 1 is required for LINQ to SQL
- Remain 100% backward compatible

LINQ Providers

- 3 main providers:
  - LINQ to Objects
  - LINQ to SQL
    (SQL Server 2000 or later)
  - LINQ to XML
- Other providers supported by Microsoft:
  - LINQ to DataSet
  - LINQ to Entities
    (Entity Framework)
  - Parallel LINQ
- 3rd party providers:
  - LINQ to Amazon
  - LINQ to Google
  - LINQ to Excel
  - LINQ to WMI
  - LINQ to Oracle
  - LINQ to NHibernate
  - DbLinq (open source)
    (MySQL, Oracle, PostreSQL, SQLite, Ingres, Firebird, SQL Server)
  - ALinq (commercial)
    (MySQL, Oracle, SQLite, Firebird, SQL Server, Access)
  - …

LINQ Namespaces and Assemblies

- System.Core.dll
  - System.Linq – standard query operators provided by the System.Linq.Enumerable class
  - System.Linq.Expressions – for working with expression trees and creating custom IQueryable implementation
- System.Data.Linq – LINQ to SQL
- System.Data.DataSetExtensions.dll – LINQ to DataSet
- System.Xml.Linq – LINQ to XML

C# 3.0 Features Introduced for LINQ

Implicit Typing C# 3.0

```
var stringVariable = "Something";
var list = new List<int>();
var starter = (ThreadStart)delegate() { Console.WriteLine(); };
```

- Limitations:
  - only local variables
  - the variable must be initialized in the declaration
  - the initialization expression must not be an anonymous function without casting
  - the variable must not be initialized to null
  - only one variable can be declared in the statement
  - the type must be compile-time type

Simplified Initialization C# 3.0

```
public class Person {
    public int Age { get; set; }
    public string Name { get; set; }

    List<Person> friends = new List<Person>();
    public List<Person> Friends { get { return friends; }}

    Location home = new Location();
    public Location Home {
        get { return home;
    }

    public Person(string na
        { Name = name; }
    public Person()
        { }
}

public class Location {
    public string Country
        { get; set; }
    public string Town
        { get; set; }
}
```

```
Person tom = new Person
{
    Name = "Tom",
    Age = 4,
    Home = { Town = "Reading", Country = "UK" },
    Friends =
    {
        new Person { Name = "Phoebe" },
        new Person("Abi"),
        new Person { Name = "Ethan", Age = 4 },
        new Person("Ben")
        {
            Age = 4,
            Home = { Town = "Purley", Country="UK" }
        }
    }
};
```

Anonymous Types C# 3.0

```
var family = new[]
{
    new { Name = "Holly", Age = 31 },
    new { Name = "Jon", Age = 31 },
    new { Name = "Tom", Age = 4 },
    new { Name = "Robin", Age = 1 },
    new { Name = "William", Age = 1 }
};

int totalAge = 0;
foreach (var person in family)
{
    totalAge += person.Age;
}
Console.WriteLine("Total age: {0}", totalAge);
```

Projection C# 3.0

```
List<Person> family = new List<Person>
{
    new Person { Name = "Holly", Age = 31},
    new Person { Name = "Jon", Age = 31},
    new Person { Name = "Tom", Age = 4},
    new Person { Name = "Robin", Age = 1},
    new Person { Name = "William", Age = 1}
};

var converted = family.ConvertAll(delegate(Person person)
    { return new { person.Name, IsAdult = (person.Age >= 18) }; }
);

foreach (var person in converted)
{
    Console.WriteLine("{0} is an adult? {1}",
                      person.Name, person.IsAdult);
}
```

Lambda Expressions C# 3.0

```
Func<string, int> returnLength;
returnLength = delegate(string text) { return text.Length; };
Console.WriteLine(returnLength("Hello"));
```

```
Func<string, int> returnLength;
returnLength = (string text) => { return text.Length; };
Console.WriteLine(returnLength("Hello"));
```

```
Func<string, int> returnLength;
returnLength = (text => text.Length);
Console.WriteLine(returnLength("Hello"));
```

```
class Person : IComparable {
    public string FirstName { get; set; }
    public string LastName { get; set; }
    public int Age { get; set; }
    public override string ToString() {
        return string.Format("{0} {1} [{2}]", LastName, FirstName, Age);
    }
    int IComparable.CompareTo(object obj) {
        Person p2 = obj as Person;
        if (p2 == null) {
            return 1;
        } else {
            int cmp = LastName.CompareTo(p2.LastName);
            if (cmp == 0) {
                cmp = FirstName.CompareTo(p2.FirstName);
            }
            return cmp;
        }
    }
}
```

```
var persons = new List<Person>
{
    new Person { FirstName = "Adrian", LastName = "Abacki", Age = 14 },
    new Person { FirstName = "Damian", LastName = "Dabacki", Age = 20 },
    new Person { FirstName = "Barbara", LastName = "Babacka", Age = 16 },
    new Person { FirstName = "Adam", LastName = "Abacki", Age = 26 }
};
Action<Person> show = person => Console.WriteLine(person);
// adults
persons.FindAll(person => person.Age >= 18).ForEach(show);
// sorted by default
persons.Sort();
persons.ForEach(show);
// sorted by age
persons.Sort((p1, p2) => p1.Age.CompareTo(p2.Age));
persons.ForEach(show);
```

Extension Methods C# 3.0

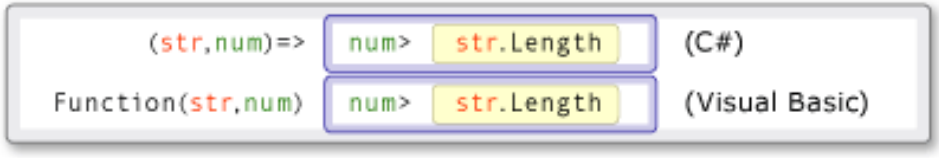
```
public static class StringExtension {
    public static int WordCount(this string str) {
        return str.Split(new char[] { ' ', '.', '?' },
            StringSplitOptions.RemoveEmptyEntries).Length;
    }
    public static int CharCount(this string str, char ch) {
        int count = 0;
        for (int i = 0; i < str.Length; i++) {
            if (str[i] == ch) {
                count++;
            }
        }
        return count;
    }
}
```

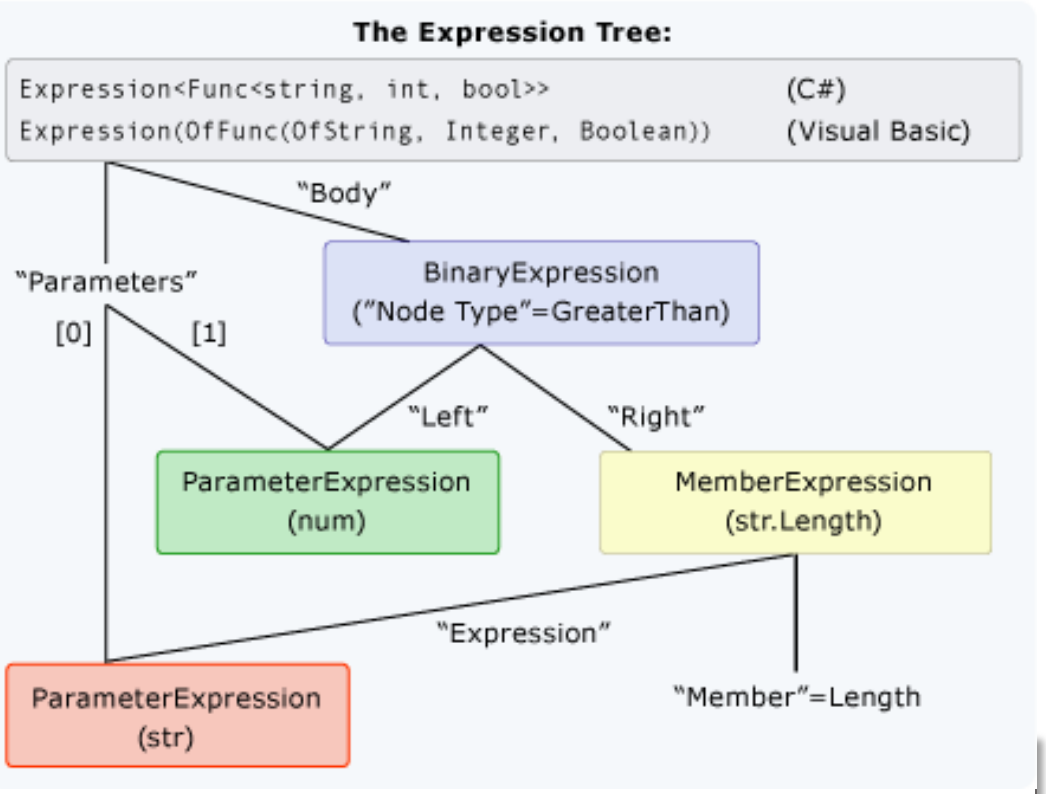
```
string s = "Sample text.";
int wordCount = s.WordCount();
int eCount = s.CharCount('e');
```

- Do not overuse it
- Can be called on a null reference

Expression Trees

- Expression trees represent language-level code in the form of data. The data is stored in a tree-shaped structure.






```
// Create an expression tree.
Expression<Func<int, bool>> exprTree = num => num < 5;
// Decompose the expression tree.
ParameterExpression param = (ParameterExpression)exprTree.Parameters[0];
BinaryExpression operation = (BinaryExpression)exprTree.Body;
ParameterExpression left = (ParameterExpression)operation.Left;
ConstantExpression right = (ConstantExpression)operation.Right;
Console.WriteLine("Decomposed expression: {0} => {1} {2} {3}",
    param.Name, left.Name, operation.NodeType, right.Value);
```

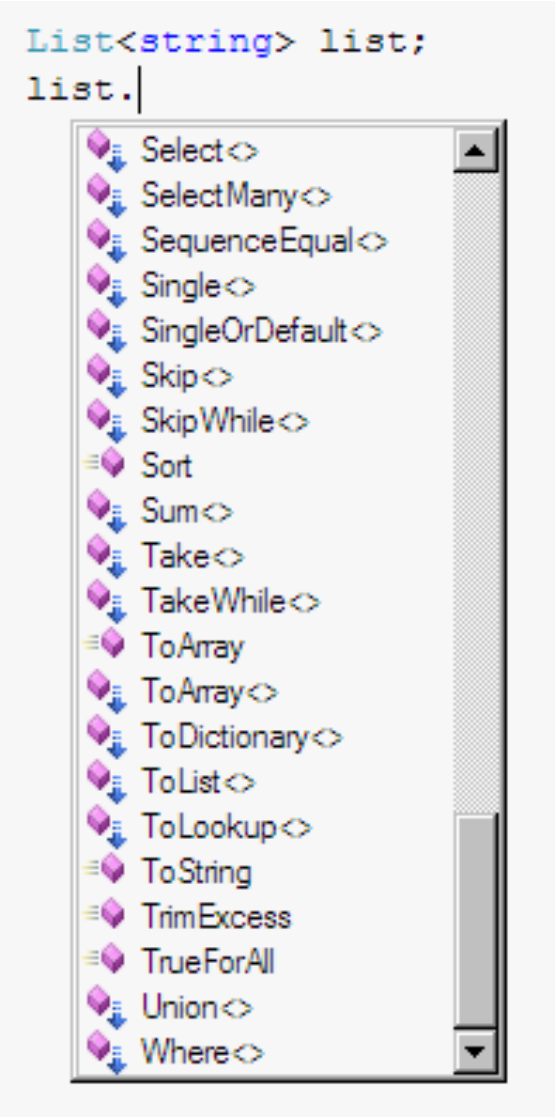
LINQ to Objects

LINQ to Objects

- **IEnumerable<T>**
  - Most of Standard Query Operators are extension methods for the IEnumerable<T> type
  - The rest of operators are static methods and must be called on the System.Linq.Enumerable class

Query Operators

- They are a set of extension methods that perform operations in the context of LINQ queries
- They are not language extensions, but extensions to the .NET Framework Class Library
- They work on enumerations
- They allow pipelined data processing
- They rely on delayed execution



C# Query Expression Syntax

```
from [ type ] id in source
[ join [ type ] id in source on expr equals expr [ into id ] ]
{ from [ type ] id in source | let id = expr | where condition }
[ orderby ordering, ordering, ... ]
select expr | group expr by key
[ into id query ]
```

- starts with **from**
- zero or more **join**
- zero or more **from**, **let** or **where**
- optional **orderby**
- ends with **select** or **group by**
- optional **into** continuation

Translating Expressions

```
var authors =
    SampleData.Books
    .Where(book => book.Title.Contains("LINQ"))
    .SelectMany(book => book.Authors.Take(1))
    .Distinct()
    .Select(author =>
        new { author.FirstName, author.LastName });
```

```
var authors =
    from distinctAuthor in (
        from book in SampleData.Books
        where book.Title.Contains("LINQ")
        from author in book.Authors.Take(1)
        select author)
        .Distinct()
    select new
        { distinctAuthor.FirstName, distinctAuthor.LastName };
```

Where

```
int[] numbers = { 5, 4, 1, 3, 9, 8, 6, 7, 2, 0 };
var lowNums =
    from n in numbers
    where n < 5
    select n;
```

```
4 1 3 2 0
```

```
string[] digits = { "zero", "one", "two", "three", "four",
    "five", "six", "seven", "eight", "nine" };
var shortDigits =
    digits.Where((digit, index) => digit.Length < index);
```

```
five six seven eight nine
```

```
var expensiveInStockProducts =
    from p in products
    where p.UnitsInStock > 0 && p.UnitPrice > 3.00M
    select p;
```

Select

```
int[] numbers = { 5, 4, 1, 3, 9, 8, 6, 7, 2, 0 };
var lowNums =
    from n in numbers
    select n + 1;
```

```
6 5 2 4 10 9 7 8 3 1
```

```
int[] numbers = { 5, 4, 1, 3, 9, 8, 6, 7, 2, 0 };
string[] strings = { "zero", "one", "two", "three", "four",
    "five", "six", "seven", "eight", "nine" };
var textNums =
    from n in numbers
    where n < 5
    select strings[n];
```

```
four one three two zero
```

Select cont.

```
int[] numbersA = { 0, 2, 4 };
int[] numbersB = { 1, 3, 5, 7 };
var pairs =
    from a in numbersA
    from b in numbersB
    where a < b
    select new {a, b};
```

```
(0,1) (0,3) (0,5) (0,7) (2,3) (2,5) (2,7) (4,5) (4,7)
```

```
int[] numbersA = { 0, 1, 2, 3, 4 };
int[] numbersB = { 1, 2, 3, 4, 5 };
var pairs =
    from a in numbersA
    where a % 2 == 0
    from b in numbersB
    where b % 2 == 1 && a < b
    select new { a, b };
```

```
(0,1) (0,3) (0,5) (2,3) (2,5) (4,5)
```

Take, TakeWhile

```
int[] numbers = { 5, 4, 1, 3, 9, 8, 6, 7, 2, 0 };
var first3even = (
    from n in numbers
    where n % 2 == 0
    select n)
    .Take(3);
```

```
4 8 6
```

```
int[] numbers = { 5, 4, 1, 3, 9, 8, 6, 7, 2, 0 };
var sequence = (
    from n in numbers
    where n % 2 == 0
    select n)
    .TakeWhile(n => n <= 4);
```

```
4
```

Skip, SkipWhile

```
int[] numbers = { 5, 4, 1, 3, 9, 8, 6, 7, 2, 0 };
var allButFirst4Numbers = numbers.Skip(4);
```

```
9 8 6 7 2 0
```

```
int[] numbers = { 5, 4, 1, 3, 9, 8, 6, 7, 2, 0 };
var allButFirst3Numbers = numbers.SkipWhile(n => n % 3 != 0);
```

```
3 9 6 0
```

```
int[] numbers = { 5, 4, 1, 3, 9, 8, 6, 7, 2, 0 };
var laterNumbers =
    numbers.SkipWhile((n, index) => n >= index);
```

```
1 3 9 8 6 7 2 0
```

OrderBy

```
int[] numbers = { 1, 3, 2 };
var sortedPairs =
    from n1 in numbers
    orderby n1
    from n2 in numbers
    orderby n2 descending
    select new { n1, n2 };
```

```
(1,3) (2,3) (3,3) (1,2) (2,2) (3,2) (1,1) (2,1) (3,1)
```

OrderBy – Custom Comparer

```
public class SpecialNumbersComparer : IComparer<int>
{
    public int Compare(int x, int y)
    {
        if (x % 2 == y % 2) {
            return x.CompareTo(y);
        } else {
            return ((x % 2 == 0) ? -1 : 1);
        }
    }
}
```

```
int[] numbers = { 5, 4, 1, 3, 9, 8, 6, 7, 2, 0 };
var sortedNumbers =
    numbers.OrderBy(n => n, new SpecialNumbersComparer());
```

```
0 2 4 6 8 1 3 5 7 9
```

ThenBy

```
string[] digits = { "zero", "one", "two", "three", "four",
    "five", "six", "seven", "eight", "nine" };
var sortedDigits =
    from d in digits
    orderby d.Length, d
    select d;
```

```
one six two five four nine zero eight seven three
```

Reverse

```
string[] digits = { "zero", "one", "two", "three", "four",
"five", "six", "seven", "eight", "nine" };

var reversedIDigits =
    (from d in digits
     where d[1] == 'i'
     select d)
    .Reverse();
```

```
nine eight six five
```

GroupBy

```
int[] numbers = { 5, 4, 1, 3, 9, 8, 6, 7, 2, 0 };

var numberGroups =
    from n in numbers
    group n by n % 5 into g
    orderby g.Key
    select new { Remainder = g.Key, Numbers = g };

foreach (var g in numberGroups) {
    Console.Write("| {0}: ", g.Remainder);
    foreach (var n in g.Numbers) {
        Console.Write("{0} ", n);
    }
}
```

```
| 0: 5 0 | 1: 1 6 | 2: 7 2 | 3: 3 8 | 4: 4 9
```

Union, Intersect, Except, Concat, EqualAll

```
int[] numbersA = { 0, 2, 4, 5, 6, 8, 9 };
int[] numbersB = { 1, 3, 5, 7, 8 };
```

```
numbersA.Union(numbersB);
```

```
0 2 4 5 6 8 9 1 3 7
```

```
numbersA.Intersect(numbersB);
```

```
5 8
```

```
numbersA.Except(numbersB);
```

```
0 2 4 6 9
```

```
numbersA.Concat(numbersB);
```

```
0 2 4 5 6 8 9 1 3 5 7 8
```

```
numbersA.SequenceEqual(numbersB);
```

```
false
```

OfType

```
object[] numbers =
    { null, 1.0, "two", 3, 4.0f, 5, "six", 7.0 };
var doubles = numbers.OfType<double>();
```

```
1 7
```

Other Available Methods

- Distinct
- First, FirstOrDefault, ElementAt
- Count, Sum, Min, Max, Average
- Any, All

Conversion Operators

- **AsEnumerable** – IEnumerable<T>
- **ToQueryable** – IQueryable<T>
- **ToArray** – array
- **ToList** – IList<T>
- **ToDictionary** – IDictionary<K, T>
- **ToLookup** – ILookup<K, T>
- **Cast** – converts a non-generic IEnumerable collection to one of IEnumerable<T> by casting each element to type T. Throws an exception for incompatible types.
- **OfType** – converts a non-generic IEnumerable collection to one of IEnumerable<T>. Only elements of type T are included.

C# Contextual Keywords for LINQ

- **from** – specifies the data source on which the query or sub-query will be run
- **group** – returns a sequence of IGrouping<TKey, TElement> objects
- **into** – used to create a temporary identifier to store the results of a group, join or select clause into a new identifier
- **join** – useful for associating elements from different source sequences that have no direct relationship
- **let** – creates a new range variable and initializes it with the result of the expression
- **orderby** – causes the returned sequence to be sorted
- **select** – specifies the type of values that will be produced when the query is executed
- **where** – specify which elements from the data source will be returned in the query expression

Deferred Query Execution

```
double Square(double n) {
    Console.WriteLine(string.Format("Computing Square ({0})...", n));
    return Math.Pow(n, 2);
}
```

```
public void DeferredQueryExecution()
{
    int[] numbers = { 1, 2, 3 };
    var query =
        from n in numbers
        select Square(n);
    foreach (var n in query) {
        Console.WriteLine(n);
    }
}
```

```
public void ImmediateQueryExecution()
{
    int[] numbers = { 1, 2, 3 };
    var query =
        from n in numbers
        select Square(n);
    foreach (var n in query.ToList()) {
        Console.WriteLine(n);
    }
}
```

```
Computing Square (1)...
1
Computing Square (2)...
4
Computing Square (3)...
9
```

```
Computing Square (1)...
Computing Square (2)...
Computing Square (3)...
1
4
9
```

Deferred Query Example

```
int[] numbers = new int[] { 1, 1, 1 };
int i = 0;
```

```
var q = from n in numbers
        select ++i;
```

```
var q = (from n in numbers
         select ++i)
        .ToList();
```

```
foreach (var v in q) {
    Console.WriteLine("(v={0}, i={1}) ", v, i);
}
```

```
(v=1, i=1)
(v=2, i=2)
(v=3, i=3)
```

```
(v=1, i=3)
(v=2, i=3)
(v=3, i=3)
```

Deferred Query Operators

- Filtering: **OfType, Where**
- Projection: **Select, SelectMany**
- Partitioning: **Skip, SkipWhile, Take, TakeWhile**
- Join: **GroupJoin, Join**
- Concatenation: **Concat**
- Ordering: **OrderBy, OrderByDescending, Reverse, ThenBy, ThenByDescending**
- Grouping: **GroupBy**
- Set: **Distinct, Except, Intersect, Union**
- Conversion: **AsEnumerable, Cast**
- Generation: **DefaultIfEmpty, Empty, Range, Repeat**

Not Deferred Query Operators

- Conversion: **ToArray, ToDictionary, ToList, ToLookup**
- Equality: **SequenceEqual**
- Element: **ElementAt, ElementAtOrDefault, First, FirstOrDefault, Last, LastOrDefault, Single, SingleOrDefault**
- Quantifiers: **All, Any, Contains**
- Aggregation: **Aggregate, Average, Count, LongCount, Max, Min, Sum**

Query Reuse

```
private void ShowValues<T>(IEnumerable<T> tValues)
{
    foreach (T t in tValues) {
        Console.Write("{0} ", t);
    }
    Console.WriteLine();
}
```

```
int []numbers = { -5, -4, -1, -3, -9, 8, 6, 7, 2 };
var positive = from n in numbers
               where n > 0
               select n;
ShowValues<int>(positive);
for (int i = 0; i < numbers.Length; i++) {
    numbers[i] = -numbers[i];
}
ShowValues<int>(positive);
```

```
8 6 7 2
5 4 1 3 9
```

LINQ to XML

LINQ to XML

- LINQ to XML allows to use the powerful query capabilities offered by LINQ with XML data
- LINQ to XML also provides developers with a new XML programming API:
  - lightweight
  - in-memory
  - designed to take advantage of the latest .NET Framework
  - similar to the DOM, but more intuitive
- It supports reading, modifying, transforming, and saving changes to XML documents

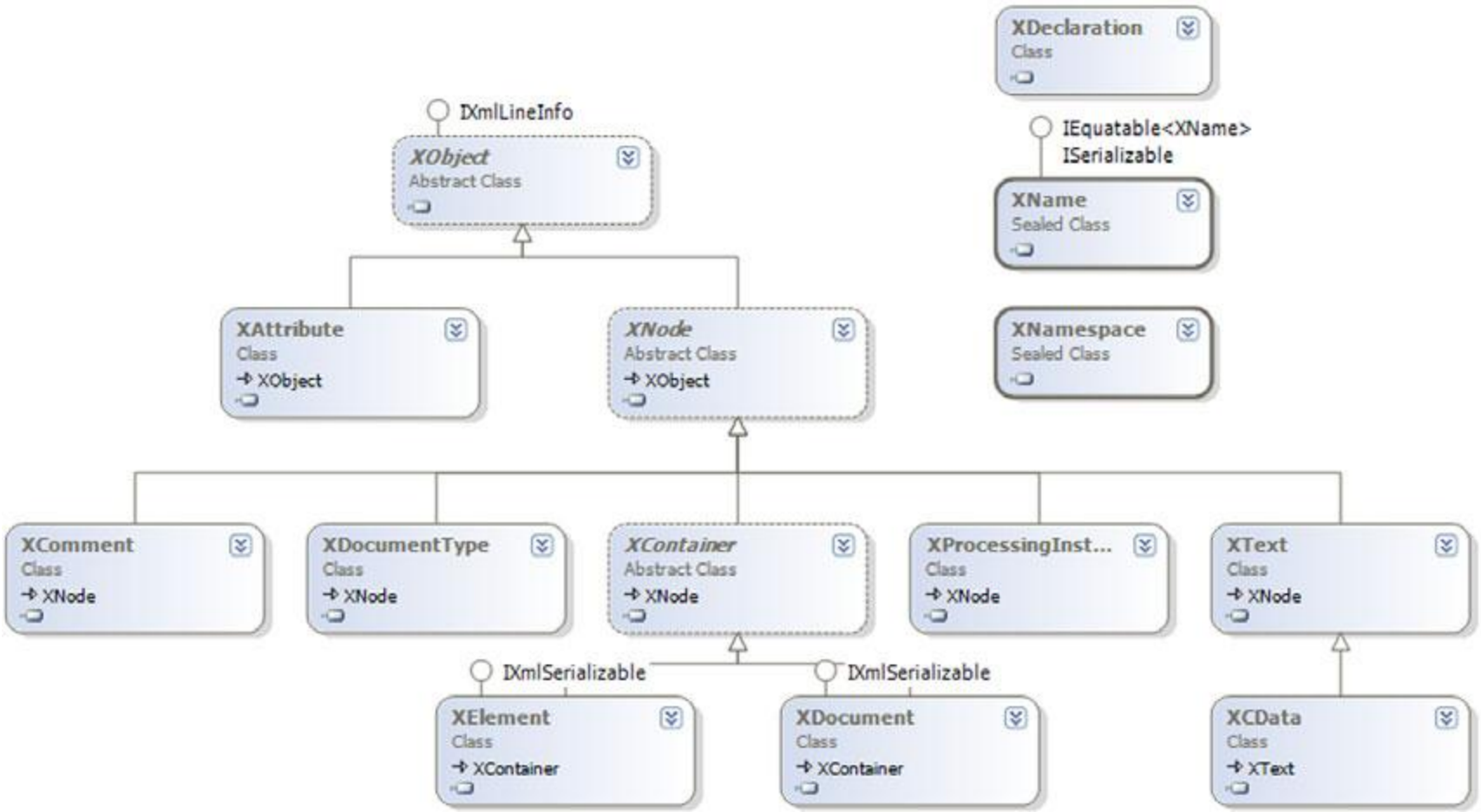
LINQ to XML Possibilities

- Load XML from files or streams
- Serialize XML to files or streams
- Create XML from scratch by using functional construction
- Query XML using XPath-like axes
- Manipulate the in-memory XML tree by using methods such as Add, Remove, ReplaceWith, and SetValue
- Validate XML trees using XSD
- Use a combination of these features to transform XML trees from one shape into another

Existing .NET Framework XML APIs

- **XmlTextReader** – low-level parsing of XML
- **XmlReader** – streaming API for large documents
- **XmlNode** – DOM – traversing the XML
- **XPathNavigator** – traversal of XML nodes via XPath expressions

LINQ to XML Class Hierarchy



LINQ to XML classes

- **XElement** is the most fundamental class within LINQ to XML
  - it represents XML element nodes that contain other elements as child nodes
  - it provides the Load, Parse, Save, and WriteTo methods
- **XDocument** represents a complete XML document
  - it can contain one root XElement, one XML declaration, one XML document type and XML processing instructions
- **XAttribute** represents attributes within LINQ to XML
  - XAttribute objects are name/value pairs that are associated with XElement objects
- **XName** represents a fully expanded name for an XElement or XAttribute
- **XNamespace** represents the namespace portion of an XName

LINQ to XML classes cont.

- **XDeclaration** – declaration of the version, encoding etc.
- **XComment** – XML comment
- **XDocumentType** – XML DTD
- **XProcessingInstruction** – a processing instruction intended to convey information to an application that processes the XML
- **XStreamingElement** – allows elements to be streamed on input and output
- **XText** and **XCData** – text node classes, used for CData sections or for mixed content

LINQ to DataSet

System.Data.TypedTableBase<T>

- The **System.Data.TypedTableBase<T>** class is new in .NET Framework 3.5
  - tables contained in a typed DataSets created by the 3.5 version are represented using classes inherited from this class
- It extends the System.Data.DataTable class by implementing the IEnumerable<T> and IEnumerable interfaces
- It allows to query the tables contained in a typed DataSet using LINQ

Querying Untyped DataSets

```
var orders1998 =
    from order in ds.Tables["Orders"].AsEnumerable()
    join customer in ds.Tables["Customers"].AsEnumerable()
        on order["CustomerID"] equals customer["CustomerID"]
    where !order.IsNull("OrderDate") &&
        ((DateTime)order["OrderDate"]).Year == 1998
    select new {
        CompanyName = customer["CompanyName"],
        OrderID = order["OrderID"],
        OrderDate = order["OrderDate"]
    };
```

- The AsEnumerable method must be called to get the IEnumerable<DataRow> interface
  - fortunately, the DataTable's Rows collection is IEnumerable

```
    from order in ds.Tables["Orders"].Rows
    join customer in ds.Tables["Customers"].Rows
```

Querying Untyped DataSets Using Relations

```
ds.Relations.Add(new DataRelation("Customers2Orders",
    ds.Tables["Customers"].Columns["CustomerID"],
    ds.Tables["Orders"].Columns["CustomerID"]));

var orders1998 =
    from customer in ds.Tables["Customers"].AsEnumerable()
    from order in customer.GetChildRows("Customers2Orders")
    where order.Field<DateTime?>("OrderDate") >=
        new DateTime(1998, 1, 1)
    select new
    {
        CompanyName = customer["CompanyName"],
        OrderID = order["OrderID"],
        OrderDate = order["OrderDate"]
    };
```

Querying Typed DataSets

```
var orders1998 =
    from order in ds.Orders
    join customer in ds.Customers
        on order.CustomerID equals customer.CustomerID
    where !order.IsOrderDateNull() &&
        order.OrderDate.Year == 1998
    select new
    {
        CompanyName = customer.CompanyName,
        OrderID = order.OrderID,
        OrderDate = order.OrderDate
    };
```

LINQ to SQL

1. Defining Entity Classes

```
[Table(Name = "dbo.Customers")]
public class Customer {
    [Column(IsPrimaryKey = true)]
    public string CustomerID { get; set; }
    [Column]
    public string CompanyName { get; set; }
}

[Table(Name = "dbo.Orders")]
public class Order {
    [Column(IsPrimaryKey = true)]
    public int OrderID { get; set; }
    [Column]
    public string CustomerID { get; set; }
}
```

- **TableAttribute** – to designate a class as an entity class that is associated with a database table or view
- **ColumnAttribute** – used to designate a member of an entity class to represent a column in a database table

2. Creating the DataContext

```
ConnectionStringSettings css =
  ConfigurationManager.ConnectionStrings["NorthwindConnectionString"];
DataContext dataContext = new DataContext(css.ConnectionString);

Table<Customer> customers = dataContext.GetTable<Customer>();
Table<Order> orders = dataContext.GetTable<Order>();

Console.WriteLine("Number of customers: {0}", customers.Count());
Console.WriteLine("Number of orders: {0}", orders.Count());
```

- The **DataContext** class represents the main entry point for the LINQ to SQL framework
- It is the source of all entities mapped over a database connection
- It tracks changes made to all retrieved entities and maintains an "identity cache" that guarantees that entities retrieved more than one time are represented by using the same object instance

3. Defining Relationships

```
[Table(Name = "dbo.Customers")]
public class Customer
{
    [Column(IsPrimaryKey = true)]
    public string CustomerID { get; set;
    [Column]
    public string CompanyName { get; set; }

    private EntitySet<Order> _Orders;
    [Association(Storage = "_Orders", OtherKey = "CustomerID")]
    public EntitySet<Order> Orders {
        get { return this._Orders; }
        set { this._Orders.Assign(value); }
    }
}
```

```
var q =
    from c in customers
    from o in c.Orders
    where c.CompanyName.StartsWith("Al")
    select new { c.CustomerID, o.OrderID };
```

```
[Table(Name = "dbo.Orders")]
public class Order
{
    [Column(IsPrimaryKey = true)]
    public int OrderID { get; set; }
    [Column]
    public string CustomerID { get; set; }

    private EntityRef<Customer> _Customer;
    [Association(Storage = "_Customer", ThisKey = "CustomerID")]
    public Customer Customer {
        get { return this._Customer.Entity; }
        set { this._Customer.Entity = value; }
    }
}
```

4. Modifying and Saving Entities

```
Table<Customer> customers = dataContext.GetTable<Customer>();
Customer customer = customers.Single(c => c.CustomerID == "ALFKI");

// Modify a property
customer.CompanyName += "--modified--";
// Create and add a new Order to Orders collection
Order ord = new Order { OrderDate = DateTime.Now };
customer.Orders.Add(ord);

// Ask the DataContext to save
dataContext.SubmitChanges();
```

```
[Table(Name = "dbo.Orders")]
public class Order
{
    [Column(IsPrimaryKey = true,
            IsDbGenerated = true)]
    public int OrderID { get; set; }
    //...
```

- Saving is always wrapped in a transaction
  - if no transaction is already in scope, the DataContext object will automatically start a database transaction

System.Data.Linq.Mapping Attributes

- **Association** – primary-key and foreign-key relationship
- **Column** – mapping for a column in the database table
- **Database** – the database name
- **Function** – used to map user-defined functions or stored procedures to a method
- **InheritanceMapping** – used for polymorphic objects
- **Parameter** – parameters for a stored procedure or function
- **Provider** – the type used to perform the querying (version of SQL Server)
- **ResultType** – the type of object that is returned as the result of a stored procedure or function
- **Table** – the name of the table

Deferred and Immediate Loading

- By default, LINQ to SQL supports deferred loading
  - only asked objects for are retrieved
  - the related objects are not automatically fetched at the same time, they will be loaded when required
- To force immediate loading:
  - set **DataContext.DeferredLoadingEnabled** to false
    - instructs the framework not to delay-load one-to-many or one-to-one relationships
  - use the **DataLoadOptions.LoadWith** method, to immediately load data related to the main target
- Use the **DataLoadOptions.AssociateWith** method to filter objects retrieved for a particular relationship

Example of Immediate Loading

```
dataContext.DeferredLoadingEnabled = false;
DataLoadOptions dlo = new DataLoadOptions();
dlo.LoadWith<Customer>(c => c.Orders);
dataContext.LoadOptions = dlo;
```

```
var custQuery = from c in customers
                where c.CompanyName.StartsWith("A")
                select c;
foreach (var c in custQuery) {
    Console.WriteLine("{0}:", c.CustomerID);
    foreach (Order o in c.Orders) {
        Console.WriteLine("    {0}", o.OrderID);
    }
}
```

Compiled Queries

- Possibility to increase performance by compiling the query once and executing it several times with different parameters

```
public static Func<NorthwindDataContext, string,
                   IQueryable<Customer>>
CustomersByCity = CompiledQuery.Compile(
    (NorthwindDataContext dc, string city) =>
        from c in dc.Customers where c.City == city select c);
```

```
NorthwindDataContext northwind = new NorthwindDataContext();

var londonCustomers = CustomersByCity(northwind, "London");
var newYorkCustomers = CustomersByCity(northwind, "Madrid");
```

Read-Only Data Context

- Set the **ObjectTrackingEnabled** property to false
- Use when there is no need of updates
- It improves performance
  - the DataContext will not track changes to the entities

Transactions

- The default transaction used by the SubmitChanges method uses the ReadCommitted isolation level

```
dataContext.Connection.Open();
dataContext.Transaction = dataContext.Connection.BeginTransaction();
try {
    // ... e.g. dataContext.ExecuteCommand("exec sp_BeforeSubmit");
    dataContext.SubmitChanges();
    // ...
    dataContext.Transaction.Commit();
}
catch {
    dataContext.Transaction.Rollback();
    throw;
}
finally {
    dataContext.Transaction = null;
}
dataContext.Connection.Close();
```

```
using (TransactionScope scope =
    new TransactionScope())
{
    // ...
    dataContext.SubmitChanges();
    // ...
    scope.Complete();
}
```

- Promotable transactions are only available when using the .NET Framework Data Provider for SQL Server (SqlClient) with SQL Server 2005

Inheritance

- LINQ to SQL supports single-table mapping, when an entire inheritance hierarchy is stored in a single database table

```
[Table]
[InheritanceMapping(Code = "C", Type = typeof(Car))]
[InheritanceMapping(Code = "T", Type = typeof(Truck))]
[InheritanceMapping(Code = "V", Type = typeof(Vehicle), IsDefault = true)]
public class Vehicle {
    [Column(IsDiscriminator = true)]
    public string Key;
    [Column(IsPrimaryKey = true)]
    public string VIN;
    [Column]
    public string MfgPlant;
}
public class Car : Vehicle {
    [Column]
    public int TrimCode;
    [Column]
    public string ModelName;
}
public class Truck : Vehicle {
    [Column]
    public int Tonnage;
    [Column]
    public int Axles;
}
```

Creating Databases

- The **DataContext.CreateDatabase** method will create a replica of the database using information from entity classes
  - attributes of entity classes describing the structure of the relational database tables and columns
  - user-defined functions, stored procedures, triggers, and check constraints are not represented by the attributes
  - it allows to create an application that automatically installs itself on a customer system, or a client application that needs a local database to save its offline state

Optimistic Concurrency

- The DataContext has built-in support for optimistic concurrency by automatically detecting change conflicts
  - individual updates only succeed if the database's current state matches the original state
- The **ColumnAttribute.UpdateCheck** property allows to control detecting change conflicts:
  - Always, Never, WhenChanged
- As an alternative, the **ColumnAttribute.IsVersion** property can be used to pick a column responsible for checking for conflicts
  - usually, version numbers or timestamps are used

Change Conflicts

- When an object is refreshed, the change tracker has the old original values and the new database values
  - it allows to determine whether the object is in conflict or not
- The **ConflictMode** parameter for the **SubmitChanges** method:
  - **FailOnFirstConflict** (the default mode)
  - **ContinueOnConflict** - conflicts should be accumulated and returned at the end of the process
- **ChangeConflictException** is thrown in case of a conflict and the **DataContext.ChangeConflicts** collection contains information about all conflicts

Change Conflicts Example

```
try
{
    dataContext.SubmitChanges(ConflictMode.ContinueOnConflict);
}
catch (ChangeConflictException e)
{
    Console.WriteLine("Optimistic concurrency error.");
    Console.WriteLine(e.Message);
    foreach (ObjectChangeConflict occ in
        dataContext.ChangeConflicts)
    {
        MetaTable metatable = dataContext.Mapping.GetTable(
            occ.Object.GetType());
        Customer entityInConflict = (Customer)occ.Object;
        Console.WriteLine("Table name: {0}",
            metatable.TableName);
        Console.Write("Customer ID: ");
        Console.WriteLine(entityInConflict.CustomerID);
        Console.ReadLine();
    }
}
```

Automatic Resolving Conflicts

```
try {
    dataContext.SubmitChanges(ConflictMode.ContinueOnConflict);
}
catch (ChangeConflictException e) {
    Console.WriteLine(e.Message);
    foreach (ObjectChangeConflict occ in dataContext.ChangeConflicts)
    {
        occ.Resolve(RefreshMode.KeepChanges);
    }
}
// Submit succeeds on second try.
dataContext.SubmitChanges(ConflictMode.FailOnFirstConflict);
```

```
dataContext.ChangeConflicts.ResolveAll(RefreshMode.KeepChanges);
```

- Available options for resolving conflicts:
  - **KeepCurrentValues**
  - **KeepChanges** (default)
  - **OverwriteCurrentValues**

Example of Using a Stored Procedure

```
CREATE FUNCTION ProductsCostingMoreThan(@cost money)
RETURNS TABLE AS
RETURN
    SELECT ProductID, UnitPrice
    FROM Products
    WHERE UnitPrice > @cost
```

```
[Function(Name = "[dbo].[ProductsCostingMoreThan]")]
public IQueryable<Product> ProductsCostingMoreThan(
    decimal? cost)
{
    return this.CreateMethodCallQuery<Product>(this,
        (MethodInfo)MethodInfo.GetCurrentMethod(),
        cost);
}
```

```
var q =
    from p in dc.ProductsCostingMoreThan(80.50m)
    join s in dc.Products on p.ProductID equals s.ProductID
    select new { p.ProductID, s.UnitPrice };
```

SQLMetal

- SQLMetal.exe can be used to extract SQL metadata from a database and generate a source file containing entity class declarations

```
SqlMetal.exe /server:.\SQLExpress /database:Northwind
/pluralize /namespace:nwind /code:Northwind.cs
```

  - the same can be done visually using Visual Studio
- The DBML (Database Mapping Language) file is foremost a description of the SQL metadata for a given database
  - it is an XML file

Previewing SQL Queries

- Use the **DataContext.Log** for debug output
- Query Visualizer for Visual Studio 2008
  http://weblogs.asp.net/scottgu/archive/2007/07/31/linq-to-sql-debug-visualizer.aspx