



TECHNOLOGY : MEET BUSINESS

# Xport C# 3.0 via Linq

William Kent

- ||| The latest and greatest version of the language we all love.
- ||| C# 3.0 NOT part of .Net Framework 3.0
- ||| C# 3.0 IS part of .Net Framework 3.5
- ||| C# 3.0 will ship with Visual Studio 2008 (Currently Beta2)

This talk was prepared with the Beta 1 of Visual Studio  
2008.

Beta 2 is now out.

Certain things could have changed

**No WARRENTY, No Money Back**

# C# 3.0 – A brief history



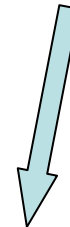
TECHNOLOGY > MEET BUSINESS

- ≡ The specifications came out in September 2005
- ≡ Various releases for a long time

Allows the type of local variables to be inferred from the declaration.

```
var aString = "Hello";  
var aDouble = 1.0;  
var anArray = new int[] { 10, 20, 30 };
```

Don't use this color  
in presentations



This is identical to

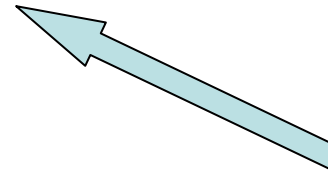
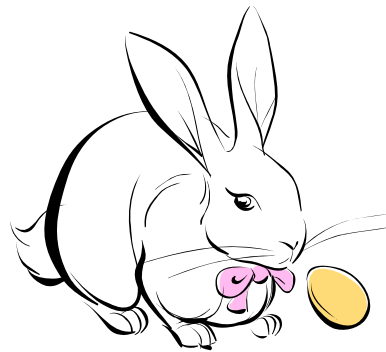
```
string aString = "Hello";  
double aDouble = 1.0;  
int[] anArray = new int[] { 10, 20, 30 };
```

**Don't think of Variants from VB**

It works in both for statements and for – each statements

- ⇒ Extension methods allow you to declare methods that add functionality to existing instance types.
- ⇒ They should only be used when modifying the instance type is not feasible. For instance if the base instance type is sealed.

```
Namespace Something.Demo {  
    public static class Extensions {  
        public static int ToInt32(this string s)  
        {  
            return Int32.Parse(s);  
        }  
    }  
}
```

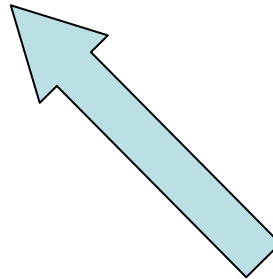


Cute rabbit

Compelling presentations should have pictures in addition to text

The previous code example when imported will apply to all instances of string

```
Using Something.Demo;  
String myString = "100";  
Int myInt = s.ToInt32();
```



**Our extension method**

- ≃ Instance methods take precedence over extension methods
- ≃ Extension methods imported in inner namespace declarations take precedence over extension methods imported in outer namespace declarations

Class representing a point:

```
public class Point {  
    private int x, y;  
  
    public int X { get { return x; } set { x = value; } }  
    public int Y { get { return y; } set { y = value; } }  
}
```

⇒ New instance can be created using object initializer:

```
var a = new Point { X = 0, Y = 1 };
```

⇒ Which is equivalent to:

```
var a = new Point();  
a.X = 0;  
a.Y = 1;
```

- ≡ Class representing a contact with name and list of phone numbers:

```
public class Contact {  
    private string name;  
    private List<string> phoneNumbers = new List<string>();  
    public string Name { get { return name; } set { name = value; } }  
    public List<string> PhoneNumbers { get { return phoneNumbers; } }  
}
```

- ≡ List of contacts can be created and initialized with:

```
var contacts = new List<Contact> {  
    new Contact {  
        Name = "Chris Smith",  
        PhoneNumbers = { "206-555-0101", "425-882-8080" }  
    },  
    new Contact {  
        Name = "Bob Harris",  
        PhoneNumbers = { "650-555-0199" }  
    }  
};
```

- ≡ Which is equivalent to:

```
var contacts = new List<Contact>();  
var __c1 = new Contact();  
__c1.Name = "Chris Smith";  
__c1.PhoneNumbers.Add("206-555-0101");  
__c1.PhoneNumbers.Add("425-882-8080");  
contacts.Add(__c1);  
var __c2 = new Contact();  
__c2.Name = "Bob Harris";  
__c2.PhoneNumbers.Add("650-555-0199");  
contacts.Add(__c2);
```

- ||| An anonymous type is described as a tuple type that is automatically inferred and created from its object initializer. An object initializer specifies the values from one or more fields or properties of an object.
  
- ||| HUH?
  
- ||| The object initializer specifies the named parameters to be passed to an object.
  
- ||| This happens at compile time, so anonymous types are still strongly typed. In reality the compiler automatically creates a class that represents the anonymous type.
  
- ||| Clear as muddy mud

Following expression:

```
new { p1 = e1 , p2 = e2 , ... pn = en }
```

Can be used to define an anonymous type :

```
class __Anonymous1 {  
    private T1 f1 ;  
    private T2 f2 ;  
    ...  
    private Tn fn ;  
  
    public T1 p1 { get { return f1 ; } set { f1 = value ; } }  
    public T2 p2 { get { return f2 ; } set { f2 = value ; } }  
    ...  
    public T1 p1 { get { return f1 ; } set { f1 = value ; } }  
}
```

And create its instance using object initializer

Different anonymous object initializers that define properties with same names in the same order generate the same anonymous type:

```
var p1 = new { Name = "Lawnmower", Price = 495.00 };  
var p2 = new { Name = "Shovel", Price = 26.95 };  
p1 = p2;
```

- || Functional programming and F#
- || C# 3.0 brings a little bit of F# goodness to everyone
- || Perhaps the first edition of C# to bring something new to the table



- ≡ Familiar to anyone with a background in functional programming
- ≡ Familiar to anyone who is a math nerd
- ≡ Somewhat similar to anonymous methods in C# 2.0
- ≡ Has a weird syntax for people who havent seen them before
- ≡ I will only cover the basics ( I am not a math nerd)

```
x => x +1
```

```
x => {return x+1;}
```

```
int x => x+1;
```

Read it like

for the variable  $x$  apply the following function to it.  $F(x) = x+1$  from math 😊

Lambda expressions are a functional superset of anonymous methods in C# 2.0 providing the following additions

- ⇒ Lambda expressions permit parameter types to be omitted and inferred
- ⇒ The body of a Lambda expression can be an expression block or a statement block
- ⇒ Lambda expressions passed as arguments participate in type argument inference and in method overload resolutions
- ⇒ Lambda expressions with an expression body can be converted to expression trees. (A run-time representation of the syntax tree.)

- ||| Expression or statement body
- ||| Implicitly or explicitly typed parameters
- ||| Examples:

```
x => x + 1 // Implicitly typed, expression body
x => { return x + 1; } // Implicitly typed, statement body
(int x) => x + 1 // Explicitly typed, expression body
(int x) => { return x + 1; } // Explicitly typed, statement body
(x, y) => x * y // Multiple parameters
() => Console.WriteLine() // No parameters
```

- ||| A lambda expression is a value, that does not have a type but can be implicitly converted to a compatible delegate type

```
delegate R Func<A,R>(A arg);
Func<int,int> f1 = x => x + 1; // Ok
Func<int,double> f2 = x => x + 1; // Ok
Func<double,int> f3 = x => x + 1; // Error – double cannot be
// implicitly converted to int
```

## What is an ORM?

≡ Orm means snake in Norwegian

≡ O-R-M

- Object
- Relational
- Mapping

- ||| Saying it is fun
- ||| It saves time, and cuts implementation cost, and allows YOU to focus on more important aspects
- ||| Creates an abstraction of the database
- ||| Creates at least a basic DAL, and objects to interact with the database.
- ||| 

```
Customer mark =new Customer("Mark", "Smith", "710-555-1212");  
Mark.Save();
```

## ≡ Common OR/M tools

You might have used:

- » Entity Spaces
- » LLBLGen Pro
- » NetTiers in CodeSmith
- » D00dads in MyGeneration
- » NHibernate
- » RapTier
- » Gentle.Net
- » Wilson OR/M
- » Jr. Programmer or Intern

Microsoft is giving us a basic ORM in C# 3.0



≅ Actually they were supposed to give us two

# This presentation is really about Linq



TECHNOLOGY > MEET BUSINESS

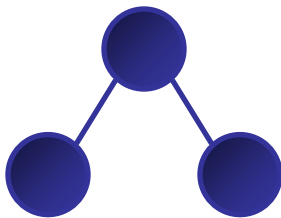
- ≡ Linq (Language INtegrated Query)
- ≡ In order to talk about Linq we must talk about C# 3.0

C#

VB

Others...

## .NET Language Integrated Query

Standard  
Query  
OperatorsDLinq  
(ADO.NET)XLinq  
(System.Xml)

Objects



SQL

WinFS

```
<book>  
<title/>  
<author/>  
<year/>  
<price/>  
</book>
```

XML

```
SqlConnection c = new SqlConnection(...);
c.Open();
SqlCommand cmd = new SqlCommand(
    @"SELECT c.Name, c.Phone
    FROM Customers c
    WHERE c.City = @p0"
);
cmd.Parameters["@p0"] = "London";
DataReader dr = c.Execute(cmd);
while (dr.Read()) {
    string name = r.GetString(0);
    string phone = r.GetString(1);
    DateTime date = r.GetDateTime(2);
}
r.Close();
```

Queries in quotes

Arguments loosely bound

Results loosely typed

Compiler cannot help catch mistakes

```
public class Customer
{
    public int Id;
    public string Name;
    public string Phone;
    ...
}
```

```
Table<Customer> customers = db.Customers;
```

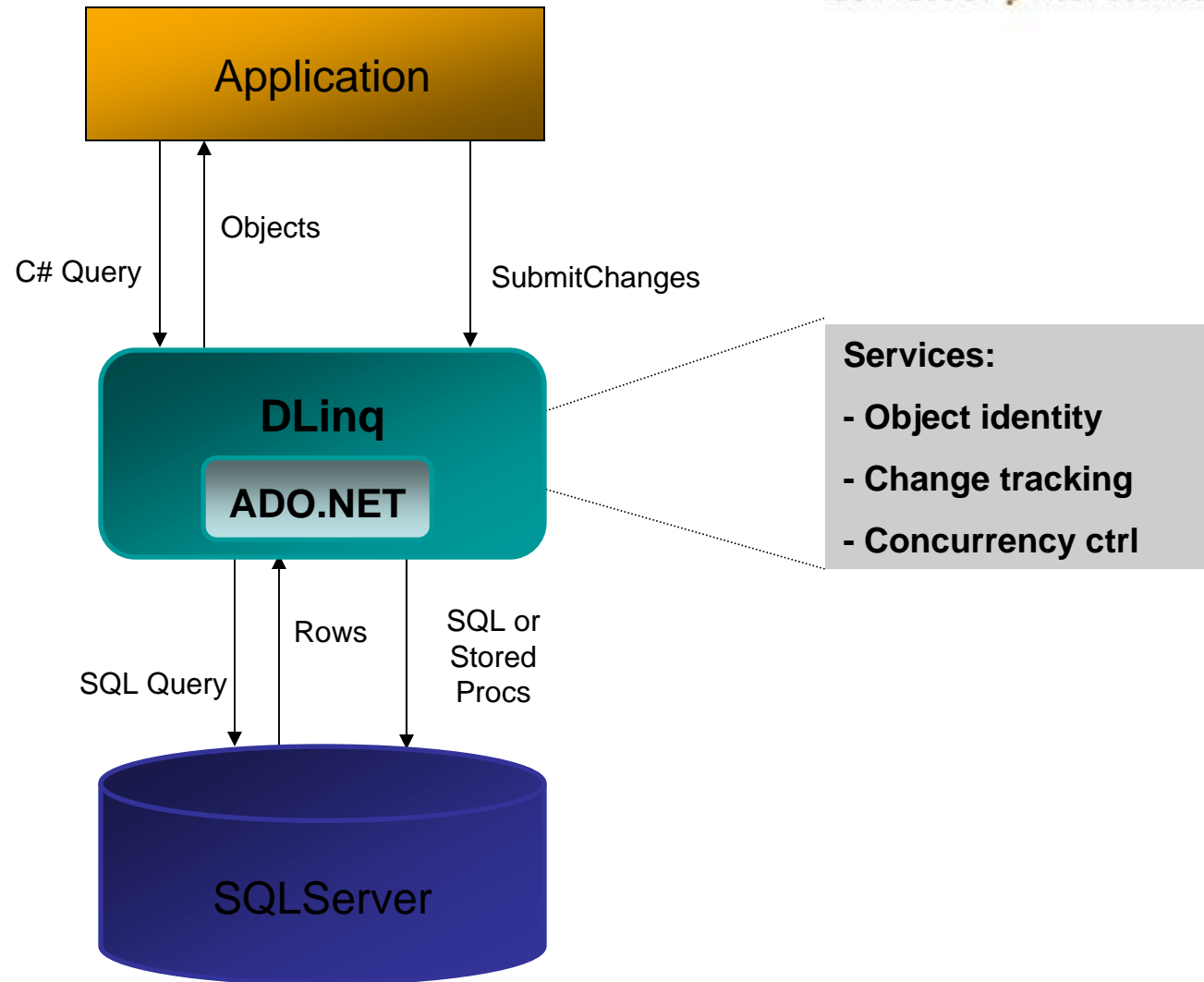
```
var contacts =
    from c in customers
    where c.City == "London"
    select new { c.Name, c.Phone };
```

Classes describe data

Tables are real objects

Query is natural part of the language

Results are strongly typed



```
public class Customer { ... }
```

Classes describe data

```
public class Northwind : DataContext  
{  
    public Table<Customer> Customers;  
    ...  
}
```

Tables are like collections

```
Northwind db = new Northwind(...);  
var contacts =  
    from c in db.Customers  
    where c.City == "London"  
    select new { c.Name, c.Phone };
```

Strongly typed connections

Integrated query syntax

Strongly typed results

```
var query = from c in customers where c.State == "WA" select c.Name;
```

```
var query = customers.Where(c => c.State == "WA").Select(c => c.Name);
```

Source implements  
IEnumerable<T>

Source implements  
IQueryable<T>

System.Query.Enumerable  
Based on Delegates

System.Query.Queryable  
Based on Expression Trees

Objects

SQL

DataSets

Others...

- ⇒ Unified query and transform of objects, relational, XML
- ⇒ SQL and XQuery-like power natively in C# and VB
- ⇒ Type checking, IntelliSense, refactoring for queries
- ⇒ Extensibility model for languages and APIs

So what in the world was the first part of the presentation about and why did we have to sit through it

```
var CommonWords =  
  from w in wordOccurrences  
  where w.Count > 2  
  select new { f.Name, w.Word, W.Count };
```

Query expressions

Local variable type inference

Lambda expressions

```
var CommonWords =  
wordOccurrences  
  .Where(w => w.Count > 2)  
  .Select(w => new { f.Name, w.Word, W.Count });
```

Extension methods

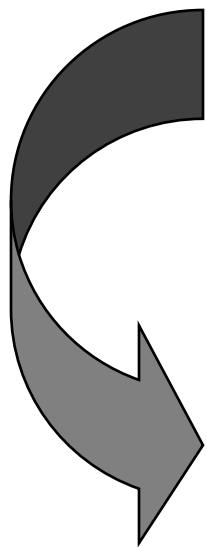
Anonymous types

Object initializers

- ||| OMG Its all coming together nwo
- ||| Every change in C# 3.0 was made to support Linq

# Son of a $\lambda$

- ≡ Queries translate to method invocations
  - » Where, Select, SelectMany, OrderBy, GroupBy



```
from c in customers
where c.State == "WA"
select new { c.Name, c.Phone };
```

```
customers
.Where(c => c.State == "WA")
.Select(c => new { c.Name, c.Phone });
```

What is this?

```
customers.Where(City == "London")
```

Expression?

Data Structure?

Code fragment?

## Local Query

Anonymous method?

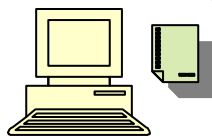
```
vertices.Where(X > Y)
```

```
vertices.Where(delegate(Point p)  
    { return p.X > p.Y; })
```

Beautiful?

## Remote Query

```
customers.Where(delegate(Customer c) {  
    return c.City == "London";  
})
```



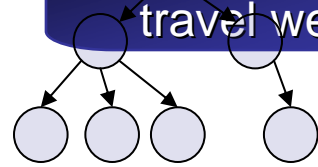
```
customers.Where(Expr.Delegate(  
    Expr.Param("c"),  
    Expr.EQ(  
        Expr.Property(  
            Expr.Param("c"),  
            "City"),  
        Expr.Literal("London"))
```

Remotable

Inspectable



AST  
(L doesn't travel well)



Good for API's

Still no type checking?

But please don't make me write this

Better if something like this ...

```
customers.Where(City == "London")
```

```
delegate(Customer c) {  
    return c.City == "London"  
}
```

could become this

```
Expr.Delegate(  
    Expr.Param("c"),  
    Expr.EQ(  
        Expr.Property(  
            Expr.Param("c"),  
            "City"),  
        Expr.Literal("London")  
    ))
```

or this

Parameterized  
fragments of code

```
public delegate Func<T,bool>(T t)  
Func<Customer,bool> f = c => c.City == "London";
```

Coercible to delegates

```
Expression<Func<Customer,bool>> e = c => c.City == "London";
```

or Expression types

Syntax is the same

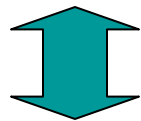
**And that is what a Lambda Expression is.**

Namespaces bring extensions into scope

```
using MyNamespace;
```

```
...
```

```
var q = vertices.Where(v => v.X > v.Y).Select(v => v.X * v.Y);
```



Discovered with Intellisense

```
var q = MyExtensions.Select(  
    MyExtensions.Where(vertices, v => v.X > v.Y),  
    v => v.X * v.Y  
);
```

Compiler translates into static invocations

```
public class Customer
{
    public string Name;
    public Address Address;
    public string Phone;
    ...
}
```

```
public class Contact
{
    public string Name;
    public string Phone;
}
```

class ???

```
{
    public string Name;
    public string Phone;
}
```

```
Customer c = GetCustomer(...);
Contact x = new Contact { Name = c.Name, Phone = c.Phone };
```

```
Customer c = GetCustomer(...);
var x = new { Name = c.Name, Phone = c.Phone };
```

```
Customer c = GetCustomer(...);
var x = new { c.Name, c.Phone };
```

Projection style  
initializer

≅ Begin with a simple array of, say, Customers.

```
Customer[] customers = new Customer[30];  
customers[0] = new Customer(...);  
...  
customers[29] = new Customer(...);
```

# Searching in Collections: The Old Way



TECHNOLOGY > MEET BUSINESS

≅ Find the names of all London customers:

```
List<string> londoners = new List<string>();  
  
foreach (Customer c in customers) {  
    if (c.City == "London") {  
        londoners.add(c.Name);  
    }  
}
```

# Linq works across your objects



TECHNOLOGY > MEET BUSINESS

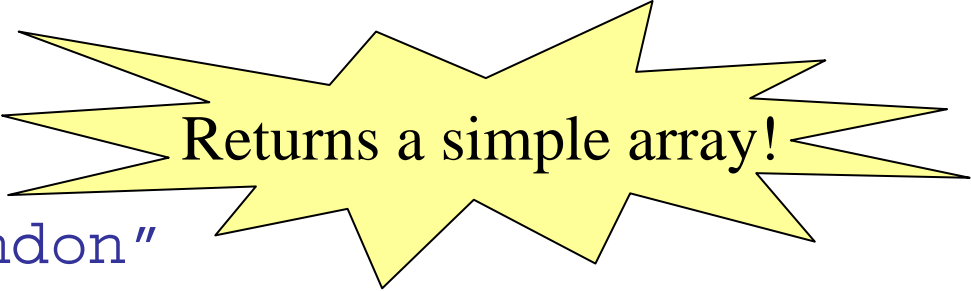
There is Linq 2 Sql (sometimes referred to as Dlinq) there is Linq to Xml (Xlinq) and there is Linq for regular objects.

Queries can span the different types.

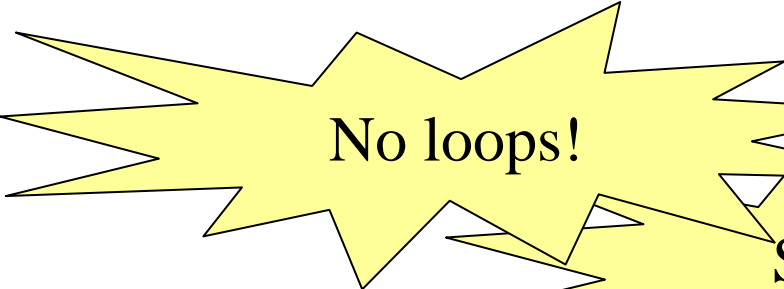
DEMO

# Searching in Collections: The LINQ Way

```
string[] londoners =  
    from c in customers  
    where c.City == "London"  
    select c.Name;
```



Returns a simple array!



No loops!



SQL-like!



Declarative!



•Example queries:

```
var q = db.Vehicle.Where(p => p is Truck);
```

```
var q = db.Vehicle.OfType<Truck>();
```

```
var q = db.Vehicle.Select(p => p as Truck).Where(p => p != null);
foreach (Truck p in q)
    Console.WriteLine(p.Axles);
```



Server Explorer

dbo.Vehicle: Table(des...B.MDF) Vehicle: Query(despen5...B.I

Column Name	Data Type	Allow Nulls
[Key]	nvarchar(4000)	<input type="checkbox"/>
VIN	nvarchar(4000)	<input type="checkbox"/>
MfgPlant	nvarchar(4000)	<input type="checkbox"/>
TrimCode	int	<input type="checkbox"/>
ModelName	nvarchar(4000)	<input type="checkbox"/>
Tonnage	int	<input type="checkbox"/>
Axles	int	<input type="checkbox"/>
		<input type="checkbox"/>

- ⇒ XLinq or Linq for Xml has been mentioned but its not covered by this presentation.
- ⇒ It allows full manipulation of xml documents, nodes and entities through the same type of interface we use for databases and objects.
- ⇒ Queries can be written in Linq, don't have to know XPath.

- ||| Linq is the biggest thing in C# 3.0 (yeah its in VB 9 too)
- ||| Extensions methods, Lamba expressions, anonymous types, object intilizers were all added to C# to support Linq
- ||| Linq is a uniform way to make queries.
- ||| Linq is extensible. You can write your own query predicates
- ||| Linq is fast.
- ||| Linq has limititions
- ||| Linq is not the ultimate OR/M.

?



TECHNOLOGY  MEET BUSINESS

## Corporate Headquarters

6501 E. Belleview Avenue,  
Suite 300  
Englewood, CO 80111-6022  
Ph. 720.346.0070  
Fx. 720.346.0080  
[www.statera.com](http://www.statera.com)