LEARNING
LIVES
HERE

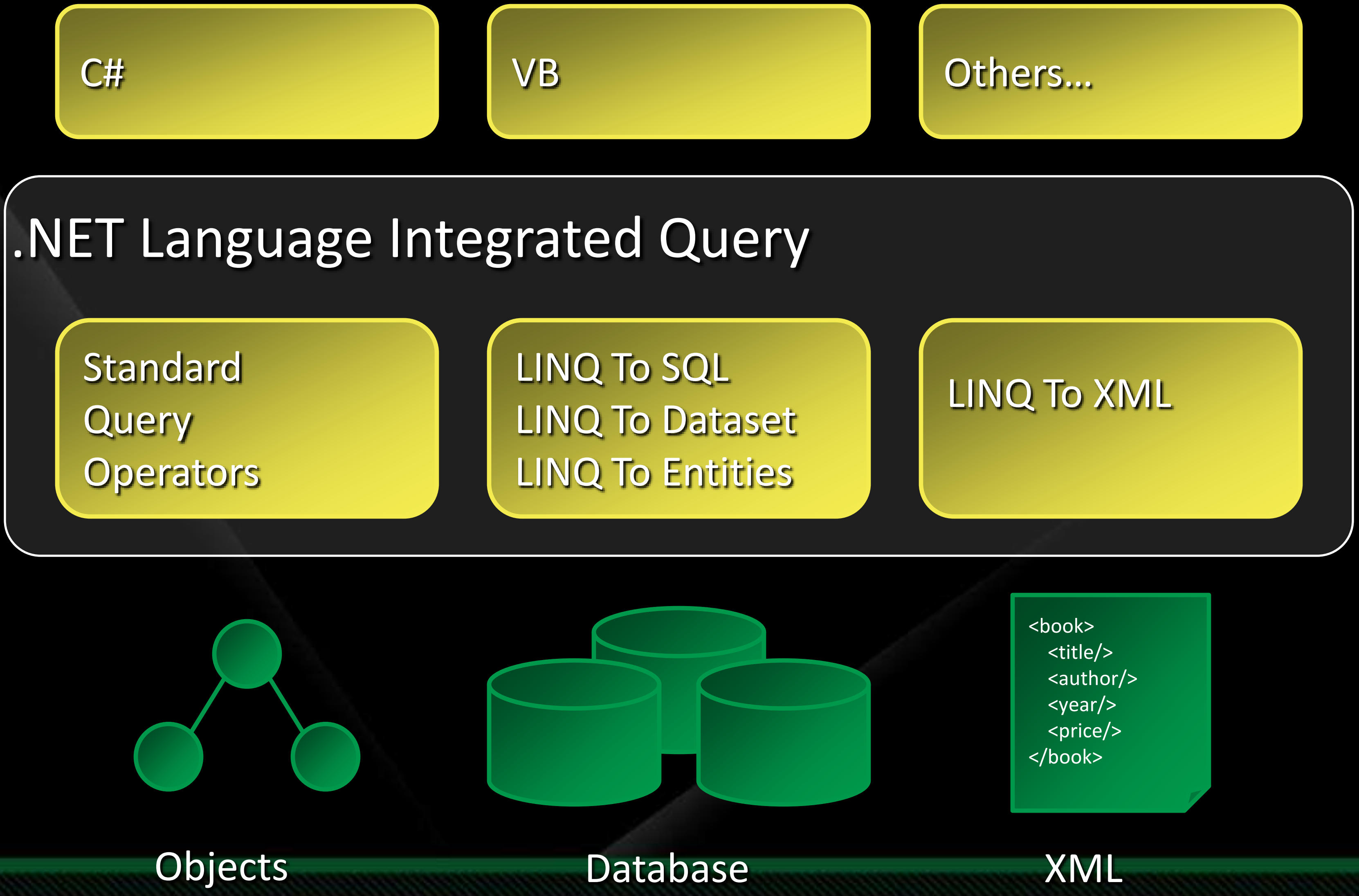
LINQ

Sanjay Vyas

msdn
Microsoft® Developer Network

Microsoft
TechNet

The LINQ Project



C#

VB

Others…

.NET Language Integrated Query

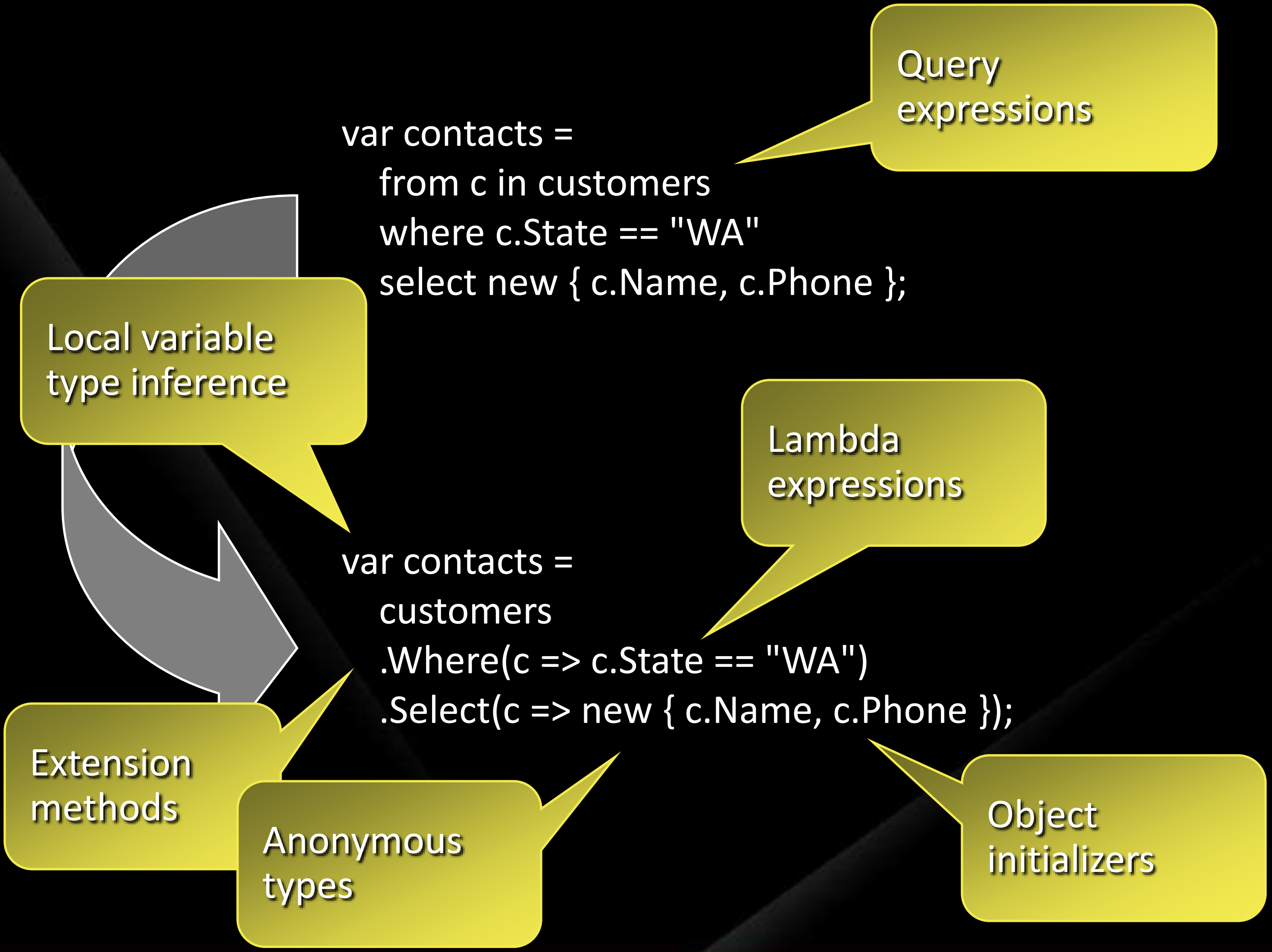
Standard Query Operators

LINQ To SQL
LINQ To Dataset
LINQ To Entities

LINQ To XML

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

Language Innovations



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

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

Query expressions

Local variable type inference

Lambda expressions

Extension methods

Anonymous types

Object initializers

Lambda Expressions

```
public delegate bool Predicate<T>(T obj);

public class List<T>
{
                FindAll(Predicate<T> test) { … }
```

```
List<C      omer> customers = GetCustomerLi   ();
```

```
List<Customer> x = customers.FindAll(
    delegate(Customer c) { return c.State == "WA"; }
);
```

```
List<Customer> x = customers.FindAll(c => c.State == "WA");
```

Explicitly typed

Statement context

Implicitly typed

Expression context

Lambda Expressions

```
public delegate T Func<T>();
public delegate T Func<A0, T>(A0 arg0);
public delegate T Func<A0, A1, T>(A0 arg0, A1 arg1);
…
```

```
Func<Customer, bool> test = c => c.State == "WA";
```

```
double factor = 2.0;
Func<double, double> f = x => x * factor;
```

```
Func<int, int, int> f = (x, y) => x * y;
```

```
Func<int, int, int> comparer =
    (int x, int y) => {
        if (x > y) return 1;
        if (x < y) return -1;
        return 0;
    };
```

Queries Through APIs

```
public class List<T>
{
    public List<T> Where(Func<T, bool> predicate) { … }
    public List<S> Select<S>(Func<T, S> selector) { … }
    …
}
```

Query operators are just methods

```
List<Customer> customers = GetCustomerList();
```

```
List<string> contacts =
    customers.Where(c => c.State == "WA").Select(c => c.Name);
```

Methods compose to form queries

Type inference figures out <S>

But what about other types?

Declare operators in all collections?

What about arrays?

Queries Through APIs

```
public static class Sequence
{
    public static IEnumerable<T> Where<T>(IEnumerable<T> source,
        Func<T, bool> predicate) { … }

    public static IEnumerable<S> Select<T, S>(IEnumerable<T> source,
        Func<T, S> selector) { … }
    …
}
```

Query operators are static methods

```
Customer[] customers = GetCustomerArray();
```

Huh?

```
IEnumerable<string> contacts = Sequence.Select(
        Sequence.Where(customers, c => c.State == "WA"),
        c => c.Name);
```

Want methods on IEnumerable<T>

Extension Methods

Extension methods

```
namespace System.Query
{
  public static class Sequence
  {
    public static IEnumerable<T> Where<T>(this IEnumerable<T> source,
      Func<T, bool> predicate) { … }

    public static IEnumerable<S> Select<T, S>(this IEnumera
      Func<T, S> selector) { … }
    …
  }
}
```

Brings extensions into scope

```
using System.Query;
```

obj.Foo(x, y)

↓

XXX.Foo(obj, x, y)

```
IEnumerable<string> contacts =
  customers.Where(c => c.State == "WA").Select(c => c.Name);
```

IntelliSense!

Object Initializers

```
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; } }
}
```

Field or property assignments

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

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

Object Initializers

```
public class Rectangle
{
   private Point p1 = new Point();
   private Point p2 = new Point();

   public Point P1 { get { return p1; } }
   public Point P2 { get { return p2; } }
}
```

Embedded objects

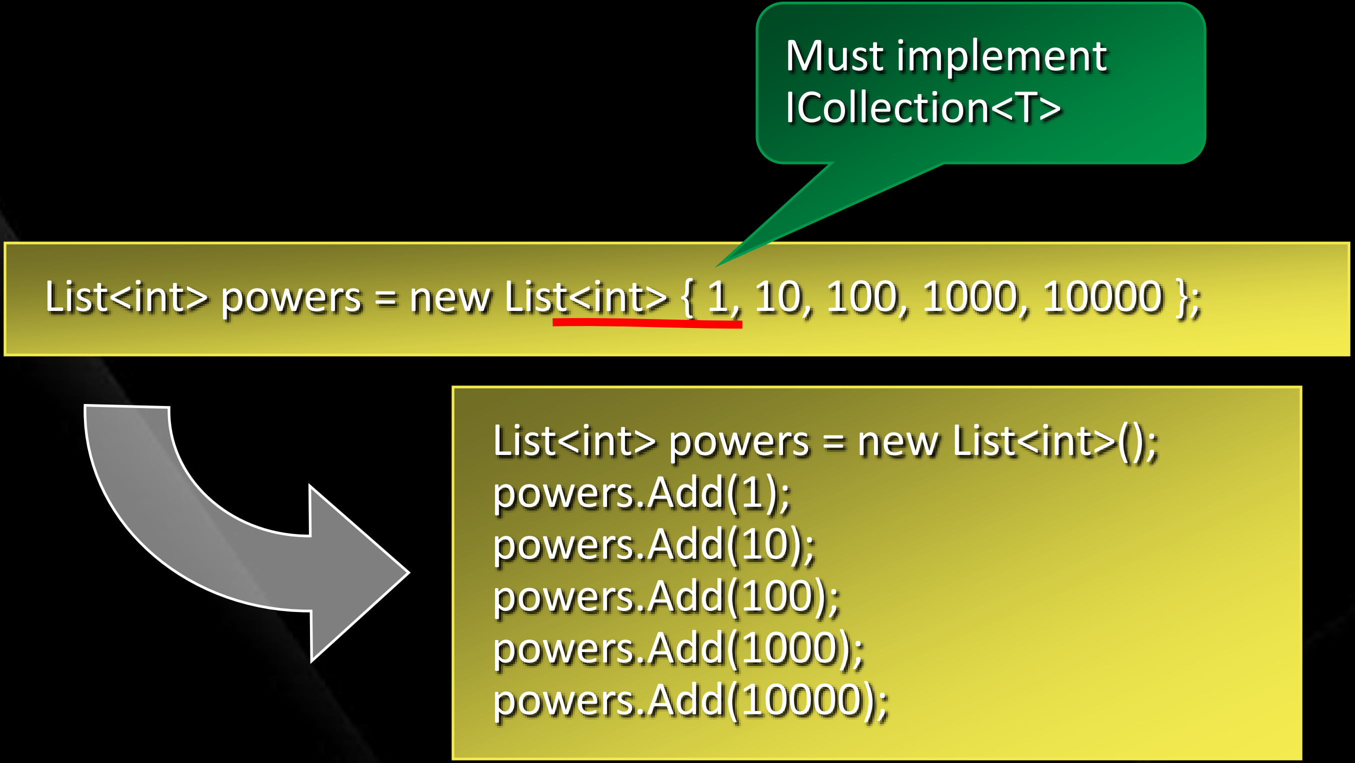
Read-only properties

```
Rectangle r = new Rectangle {
   P1 = { X = 0, Y = 1 },
   P2 = { X = 2, Y = 3 }
};
```

No “new Point”

```
Rectangle r = new Rectangle();
r.P1.X = 0;
r.P1.Y = 1;
r.P2.X = 2;
r.P2.Y = 3;
```

Collection Initializers



Must implement ICollection<T>

```
List<int> powers = new List<int> { 1, 10, 100, 1000, 10000 };
```

```
List<int> powers = new List<int>();
powers.Add(1);
powers.Add(10);
powers.Add(100);
powers.Add(1000);
powers.Add(10000);
```

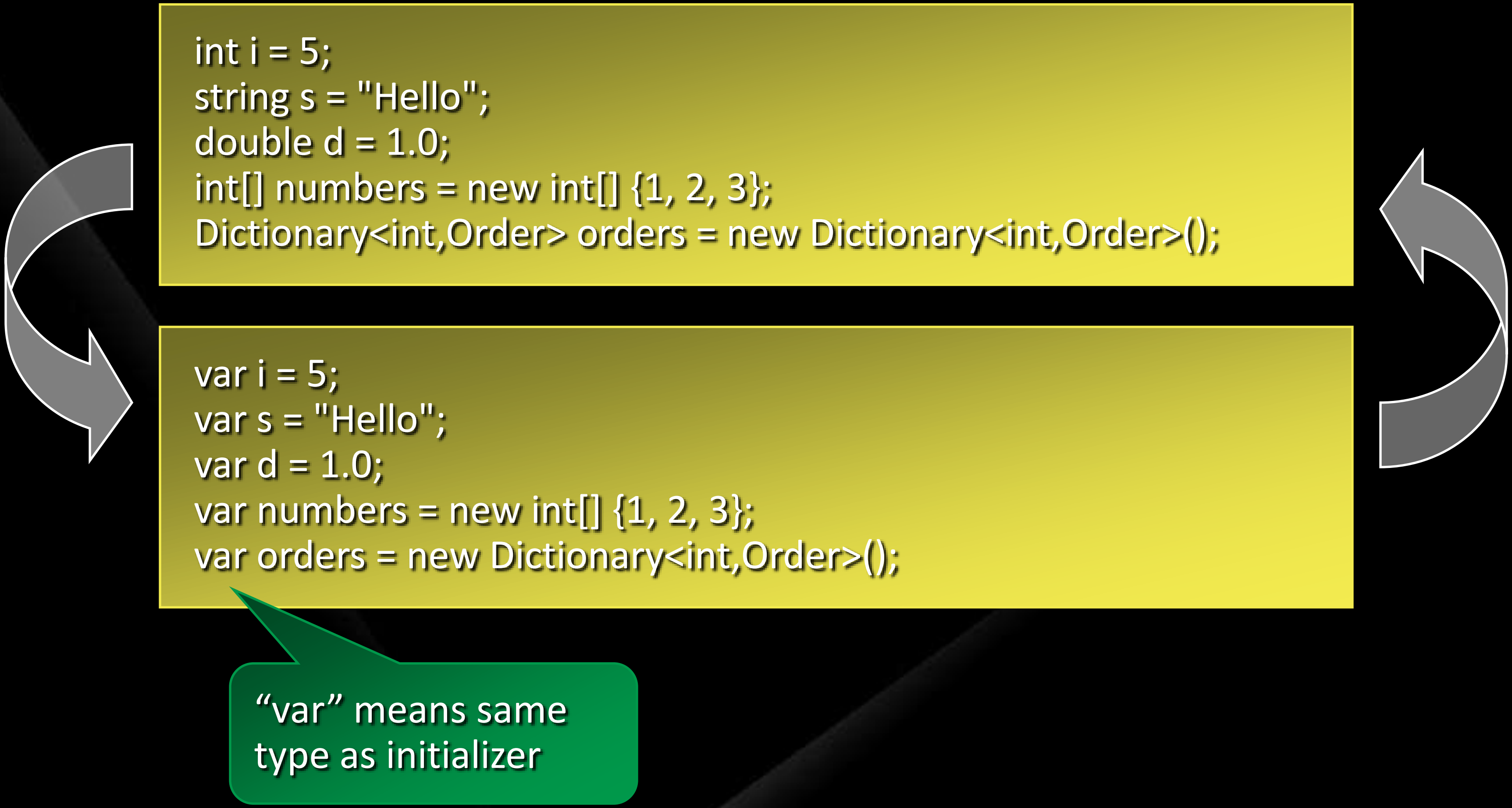
Collection Initializers

```
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<Contact> 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" }
    }
};
```

Local Variable Type Inference



```
int i = 5;
string s = "Hello";
double d = 1.0;
int[] numbers = new int[] {1, 2, 3};
Dictionary<int,Order> orders = new Dictionary<int,Order>();
```

```
var i = 5;
var s = "Hello";
var d = 1.0;
var numbers = new int[] {1, 2, 3};
var orders = new Dictionary<int,Order>();
```

“var” means same type as initializer

Anonymous Types

```
public class Customer
{
    public string Name;
    public Address Address;
    public string Phone;
    public List<Order> Orders;
    …
}
```

```
public class Contact
{
    publi
    publi
}
```

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

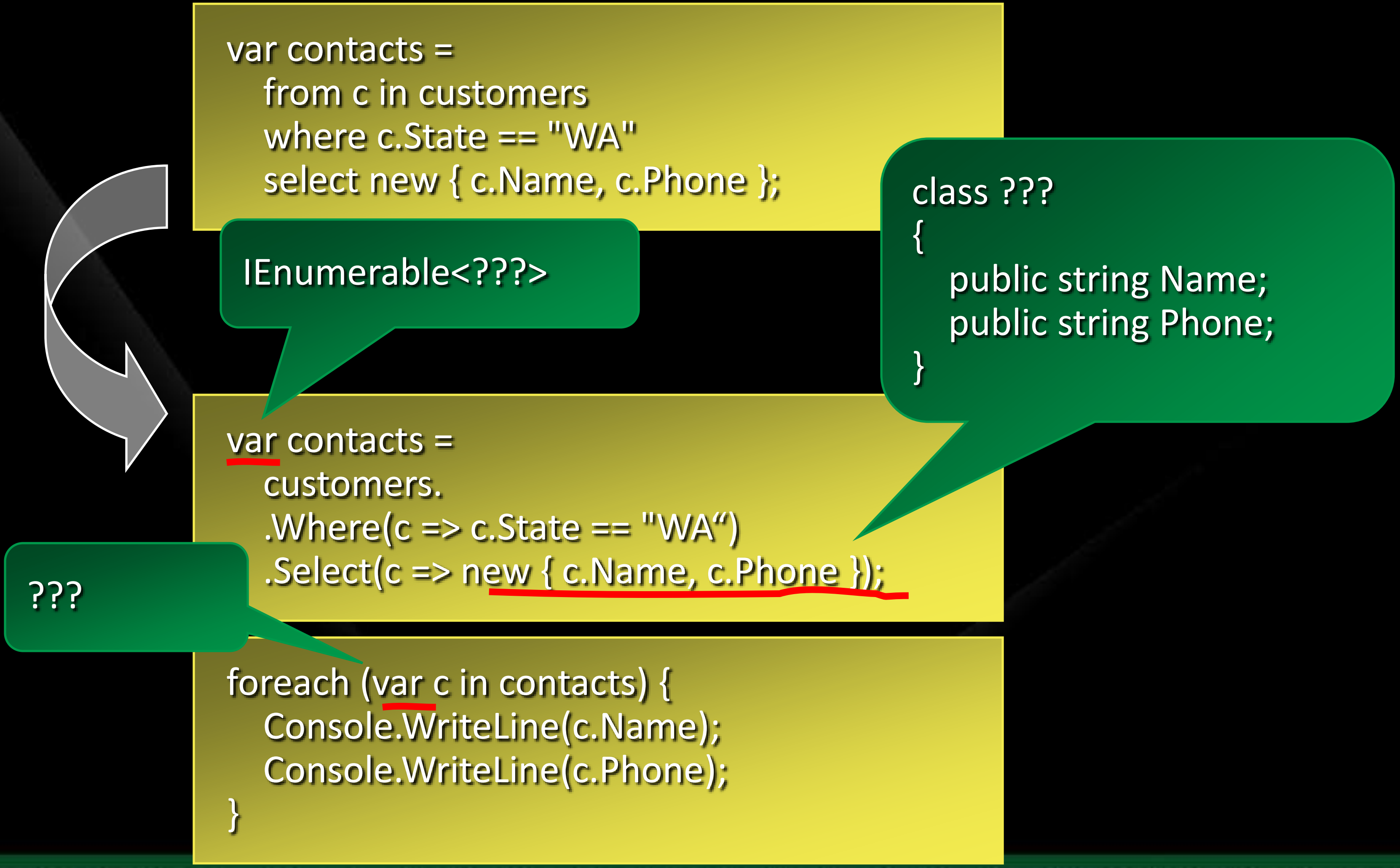
```
Customer c = GetCustomer(…);
Contact x = new Contact { Name = c.Name, Phone =        ne };
```

```
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

Anonymous Types



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

IEnumerable<???>

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

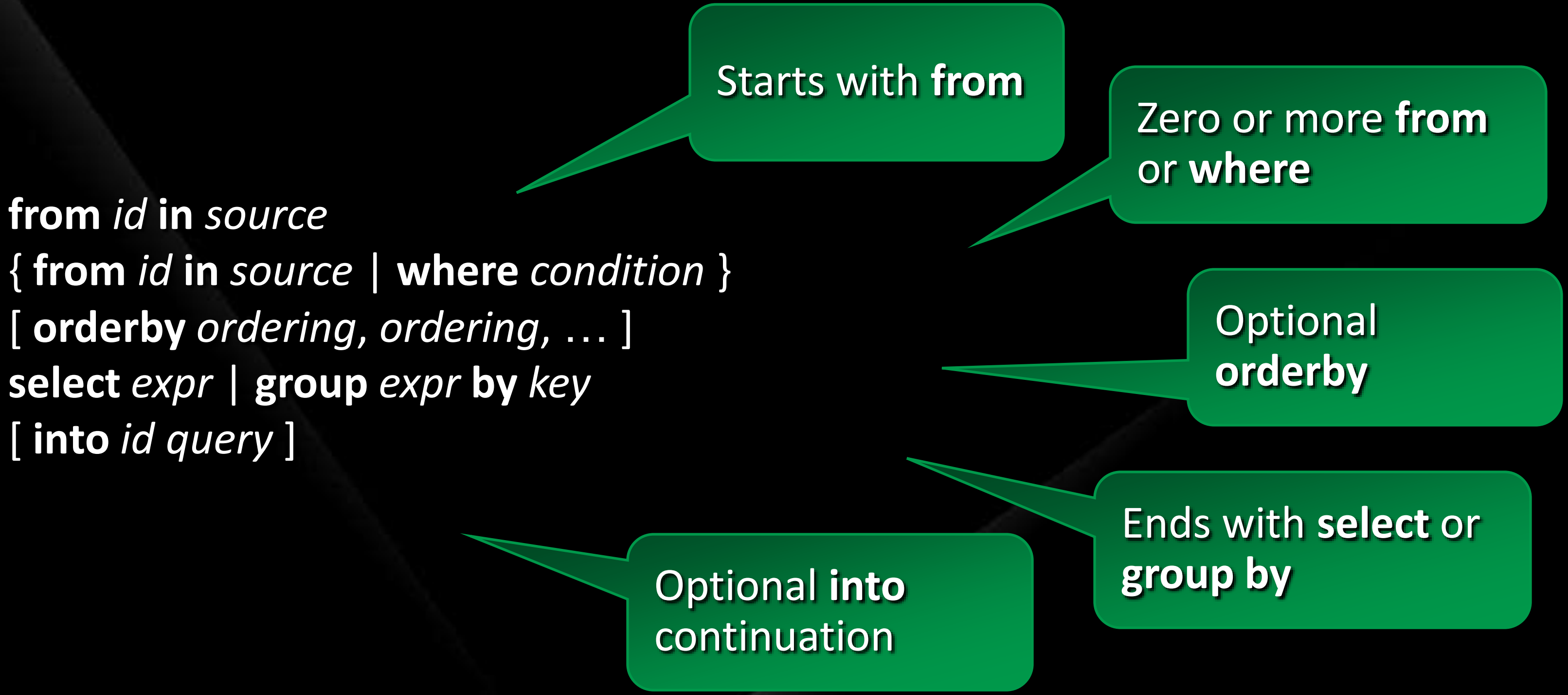
```
var contacts =
    customers.
    .Where(c => c.State == "WA“)
    .Select(c => new { c.Name, c.Phone });
```

???

```
foreach (var c in contacts) {
    Console.WriteLine(c.Name);
    Console.WriteLine(c.Phone);
}
```

Query Expressions

- Language integrated query syntax



from *id* **in** *source*

{ **from** *id* **in** *source* | **where** *condition* }

[**orderby** *ordering*, *ordering*, …]

select *expr* | **group** *expr* **by** *key*

[**into** *id query*]

Starts with **from**

Zero or more **from** or **where**

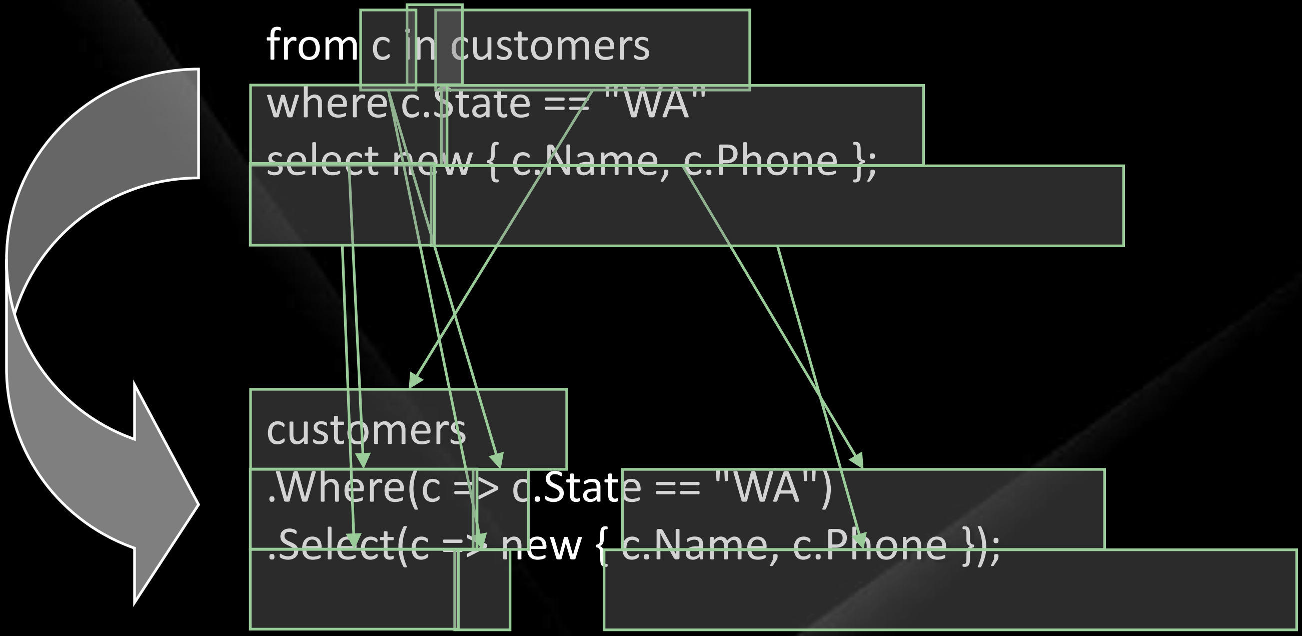
Optional **orderby**

Ends with **select** or **group by**

Optional **into** continuation

Query Expressions

- 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 });
```

Expression Trees

```
public class Northwind: DataContext
{
    public Table<Customer> Customers;
    public Table<Order> Orders;
    …
}
```

```
Northwind db = new Northwind(…);
var query = from c in db.Customers where c.State == "WA" select c;
```

```
Northwind db = new Northwind(…);
var query = db.Customers.Where(c => c.State == "WA");
```

How does this get remoted?

```
public class Table<T>: IEnumerable<T>
{
    public Table<T> Where(Expression<Func<T, bool>> predicate);
    …
}
```

Method asks for expression tree

System.Expressions.
Expression<T>

Expression Trees

- Code as Data

```
Func<Customer, bool> test = c => c.State == "WA";
```

```
Expression<Func<Customer, bool>> test = c => c.State == "WA";
```

```
ParameterExpression c =
    Expression.Parameter(typeof(Customer), "c");
Expression expr =
    Expression.EQ(
        Expression.Property(c, typeof(Customer).GetProperty("State")),
        Expression.Constant("WA")
    );
Expression<Func<Customer, bool>> test =
    Expression.Lambda<Func<Customer, bool>>(expr, c);
```

Language Innovations

- Lambda expressions
- Extension methods
- Local variable type inference
- Object initializers
- Anonymous types
- Query expressions
- Expression trees

c => c.Name

static void Dump(this object o);

var x = 5;

new Point { x = 1, y = 2 }

new { c.Name, c.Phone }

from … where … select

Expression<T>

Why Do We Need LINQ

Object-Relation mismatch

- O-R Mapping tools – EJB, Hibernate

Object-Hierarchical mismatch

- OODBMS, Code generators

Object-XML mismatch

- SAX, DOM Model

Complex Call Level Interfaces

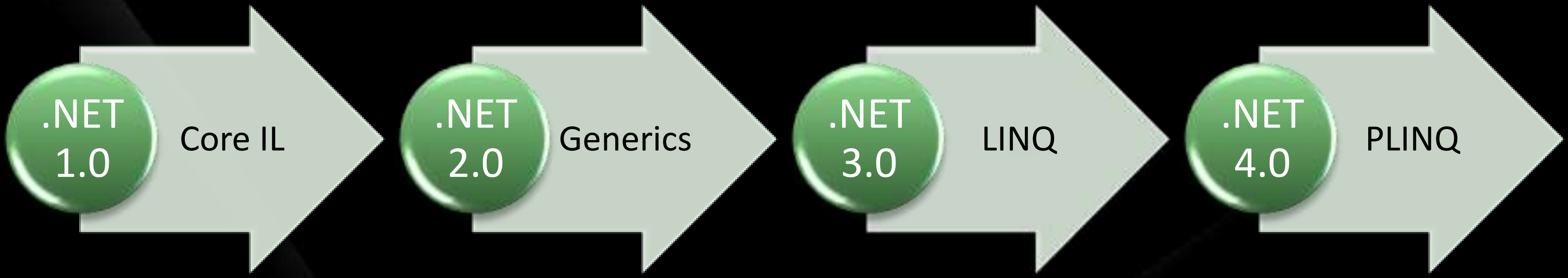
- ODBJ, JDBC, ADO.NET

Code readability & maintenance issue

Flavours of LINQ

- LINQ To Objects
- LINQ To XML
- LINQ To SQL
- LINQ To DataSets
- LINQ To Entities
- PLINQ
- LINQ To Amazon
- LINQ To Flickr
- LINQ To Nhibertnate
- LINQ To LDAP
- LINQ To Google
- LINQ To SharePoint

Evolution



.NET 1.0

Core IL

.NET 2.0

Generics

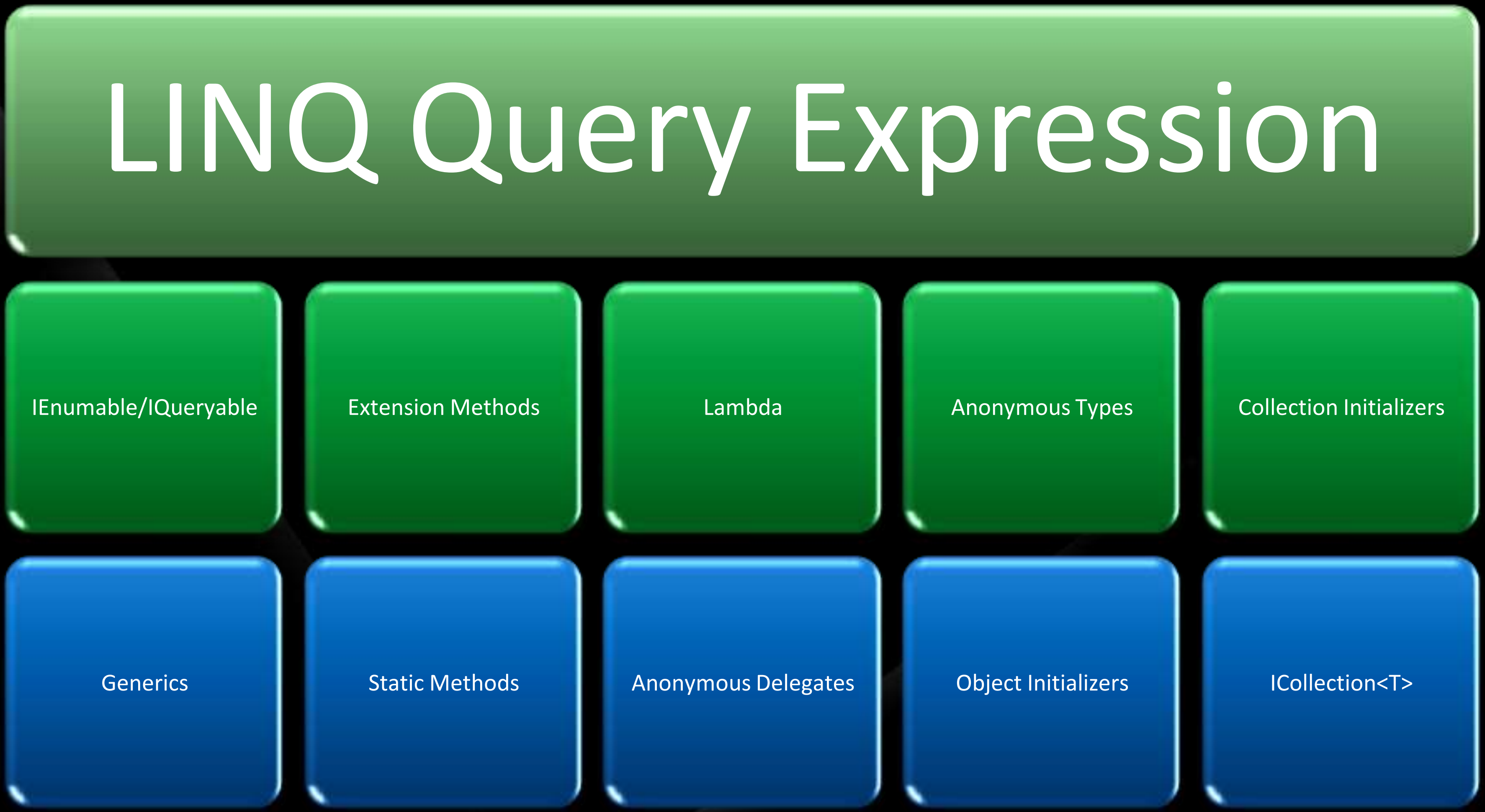
.NET 3.0

LINQ

.NET 4.0

PLINQ

Foundation



LINQ Query Expression

IEnumable/IQueryable

Extension Methods

Lambda

Anonymous Types

Collection Initializers

Generics

Static Methods

Anonymous Delegates

Object Initializers

ICollection<T>

LINQ to SQL

Accessing data today

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

Queries in quotes

Loosely bound arguments

Loosely typed result sets

No compile time checks

LINQ to SQL

Accessing data with LINQ

```
public class Customer { … }

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

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

Classes describe data

Tables are like collections

Strongly typed connections

Integrated query syntax

Strongly typed results

LINQ to SQL

- Language integrated data access
  - Maps tables and rows to classes and objects
  - Builds on ADO.NET and .NET Transactions
- Mapping
  - Encoded in attributes or external XML file
  - Relationships map to properties
- Persistence
  - Automatic change tracking
  - Updates through SQL or stored procedures

DataContext

- A **DataContext** is used to scope changes made to classes defined by LINQ to SQL
- A **DataContext** is responsible for keeping references to all LINQ to SQL classes, their properties, and foreign key relationships.
- A **DataContext** is not meant to be kept around; we want to create a new context for every “unit of work” to avoid concurrency issues. There are multiple ways to approach this.
- A **DataContext** is the API to the database, but at this stage it does not contain any business logic that is not implied by the database schema.

Defining DataContext

- Inherit from DataContext
- Override Constructor(s)

```
[Database(Name = “MyDB")]
public class MyDataContext : DataContext
{
        public MyDataContext(string connString)
                                : base(connString)
        {
        }
}
```

Creating DataContext

Similar to SqlConnection()

```
public static void Main()
{
     string connString = “server=MyServer; database=MyDb”;
    MyDataContext context = new MyDataContext(connString);
    :
    :
    :
}
```

LINQ Queries

- SQL “like” Syntax
- Not a hack/kludge
- Built upon
  - Generics
  - Extension methods
  - Lamdas

```
var result = from cust in context.Customers
                    where cust.Location = “Pune”
                    select cust;

foreach (Customer c in result)
{
    Console.WriteLine(c.CustomerName);
}
```

LINQ Queries

- LINQ To SQL fetches data from database
- Populates the Table Object/EntitySet
- Basic LINQ semantics allows iteration

join Query

- SQL “Like” join
- Inner join implemented as natural syntax
- Outer joins thru “DataShapes”

```
var  result = from c in Customers
                    join o in Order on c.CustomerID equals o.CustomerID
                    select new { c.CustomerName, o.OrderID }

foreach (var v in result)
{
    Console.WriteLine(v);
}
```

Attribute Mapping

- Declarative mapping
- No code required
- Map Relational to Objects

```
[Table(Name=“prod”)]
public class Product
{
     [Column(Name=“ProdId”, IsPrimaryKey=true)]
      public string ProductID;

      [Column]
      public string ProductName;
}
```

XML Mapping

- Externalized mapping
- Can be modified without rebuild
- Can be generated dynamically

Sample xml mapping file

```
<?xml version="1.0" encoding="utf-8"?>
<Database Name="northwind" xmlns="http://schemas.microsoft.com/linqtosql/mapping/2007">
 <Table Name="dbo.Customers" Member="Customers">
  <Type Name="Customer">
   <Column Name="CustomerID"
       Member="CustomerID"
       Storage="_CustomerID"
       DbType="NChar(5) NOT NULL"
       CanBeNull="false"
       IsPrimaryKey="true" />
   <Column Name="CompanyName"
       Member="CompanyName"
       Storage="_CompanyName"
       DbType="NVarChar(40) NOT NULL"
       CanBeNull="false" />
  </Type>
 </Table>
</Database>
```

Code Generation Tools

- Attribute and XML can be manually generated
- CodeGen Tools
  - VS Designer Tool
    - Link to SQL class item
    - Server Explorer Drag and Drop
  - SQLMetal.exe
    - Can generate DBML (Database Markup Language)
    - XML Mapping File
    - Attribute mapped code file (.cs|.vb)
  - VLinq
    - Visual design LINQ Queries

LINQ Associations

- Mirror database relation in object collection
- Master-Detail mapping
- Data available thru Object Collections

```
[Table(Name=“Customers”]
Class Customer
{
    [Column] public string CustomerID;
    [Column]public string CompanyName;

    [Association(ThisKey=“CustomerID”, OtherKey=“CustomerID”]
    public EntitySet<Order> orders;
}

[Table(Name=“Orders”)]
public class Order
{
    [Column] public string CustomerID;
    [Column] public stringOrderID;
}
```

Association Thru XMLMapping

- Similar to attribute

```
<?xml version="1.0" encoding="utf-8"?>
<Database Name="northwind" xmlns="http://schemas.microsoft.com/linqtosql/mapping/2007">
  <Table Name="dbo.Customers" Member="Customers">
    <Type Name="Customers">
      <Column Name="CustomerID" Member="CustomerID" Storage="_CustomerID" DbType="NChar(5) NOT NULL" CanBeNull="false" IsPrimaryKey="true" />
      <Column Name="CompanyName" Member="CompanyName" Storage="_CompanyName" DbType="NVarChar(40) NOT NULL" CanBeNull="false" />
      <Association Name="FK_Orders_Customers" Member="Orders" Storage="_Orders" ThisKey="CustomerID" OtherKey="CustomerID"/>
    </Type>
  </Table>

  <Table Name="dbo.Orders" Member="Orders">
    <Type Name="Orders">
      <Column Name="OrderID" Member="OrderID" Storage="_OrderID" DbType="Int NOT NULL IDENTITY" IsPrimaryKey="true" IsDbGenerated="true" 
AutoSync="OnInsert" />
      <Column Name="CustomerID" Member="CustomerID" Storage="_CustomerID" DbType="NChar(5)" />
      <Column Name="OrderDate" Member="OrderDate" Storage="_OrderDate" DbType="DateTime" />
    </Type>
  </Table>
</Database>
```

Call StoreProcedures

- SPs can be mapped thru attributes or XML
- Call semantics similar to tables
- Supports parameter passing (in/out)
- Existing Entity behaviour can be changed to use SPs instead of SQL

LINQ to Entities

```
using(AdventureWorksDB aw = new
AdventureWorksDB(Settings.Default.AdventureWorks)) {
   Query<SalesPerson> newSalesPeople = aw.GetQuery<SalesPerson>(
      "SELECT VALUE sp " +
      "FROM AdventureWorks.AdventureWorksDB.SalesPeople AS sp " +
      "WHERE sp.HireDate > @date",
      new QueryParameter("@date", hireDate));

   foreach(SalesPerson p in newSalesPeople) {
      Console.WriteLine("{0}\t{1}", p.FirstName, p.LastName);
   }
}
```

```
using(AdventureWorksDB aw = new
AdventureWorksDB(Settings.Default.AdventureWorks)) {
   var newSalesPeople = from p in aw.SalesPeople
                        where p.HireDate > hireDate
                        select p;

   foreach(SalesPerson p in newSalesPeople) {
      Console.WriteLine("{0}\t{1}", p.FirstName, p.LastName);
   }
}
```

PLINQ

Parallel LINQ

Declarative Data Parallelism

Parallel LINQ-to-Objects (PLINQ)

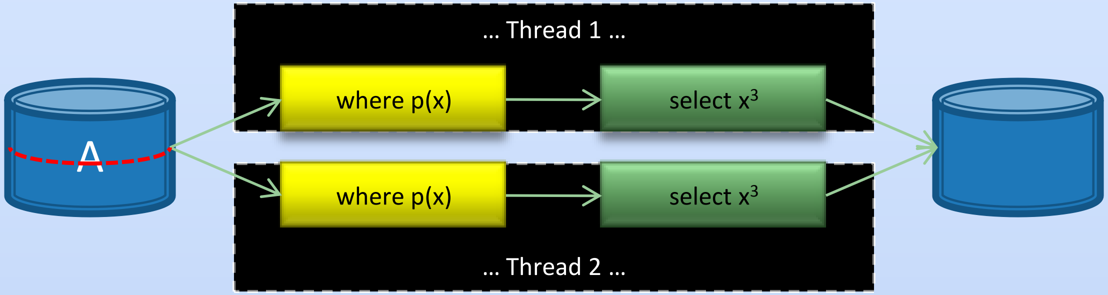
- Enables LINQ devs to leverage multiple cores
- Fully supports all .NET standard query operators
- Minimal impact to existing LINQ model

```
var q = from p in people.AsParallel()
        where p.Name == queryInfo.Name &&
              p.State == queryInfo.State &&
              p.Year >= yearStart &&
              p.Year <= yearEnd
        orderby p.Year ascending
        select p;
```

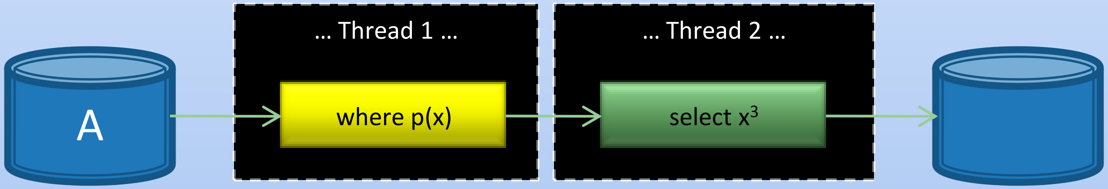
Parallelism Illustrations

q = from x in A where p(x) select x^3;

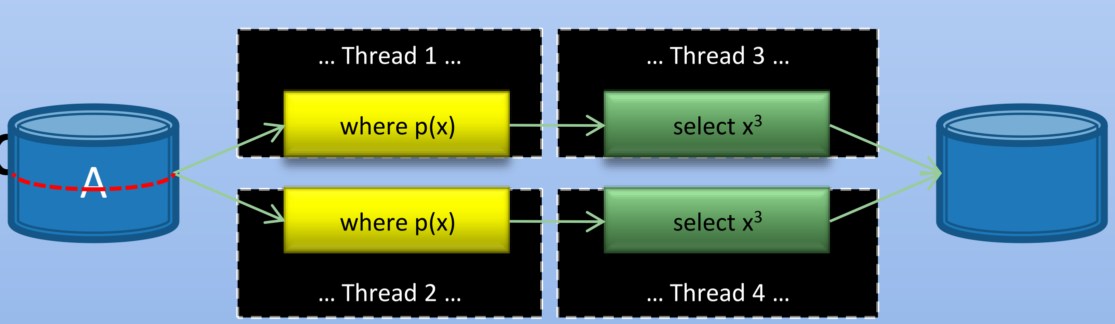
- Intra-operator:



- Inter-operator:



- Both composed



Operator Parallelism

- *Intra-operator*, i.e. *partitioning*:
  - Input to a single operator is “split” into *p* pieces and run in parallel
  - Adjacent and nested operators can enjoy *fusion*
  - Good temporal locality of data – each datum “belongs” to a partition
- *Inter-operator*, i.e. *pipelining*
  - Operators run concurrently with respect to one another
  - Can avoid “data skew”, i.e. imbalanced partitions, as can occur w/ partitioning
  - Typically incurs more synchronization overhead and yields considerably worse locality than intra-operator parallelism, so is less attractive
- Partitioning is preferred unless there is no other choice
  - For example, sometimes the programmer wants a single-CPU view, e.g.:

    ```
    foreach (x in q) a(x)
    ```
  - Consumption action *a* for might be written to assume no parallelism
  - Bad if a(x) costs more than the element production latency
    - Otherwise, parallel tasks just eat up memory, eventually stopping when the bounded buffer fills
    - But a(x) can be parallel too

Deciding Parallel Execution Strategy

- Tree analysis informs decision making:
  - Where to introduce parallelism?
  - And what kind? (partition vs. pipeline)
  - Based on intrinsic query properties and operator costs
    - Data sizes, selectivity (for filter *f*, what % satisfies the predicate?)
    - Intelligent “guesses”, code analysis, adaptive feedback over time
- But not just parallelism, higher level optimizations too, e.g.
  - Common sub-expression elimination, e.g.

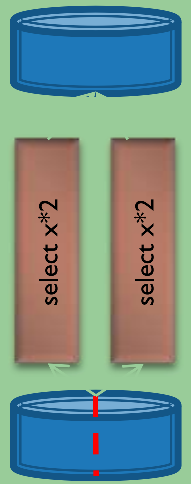
    ```
    from x in X where p(f(x)) select f(x);
    ```
  - Reordering operations to:
    - Decrease cost of query execution, e.g. put a *filter* before the *sort*, even if the user wrote it the other way around
    - Achieve better operator *fusion*, reducing synchronization cost

Partitioning Techniques

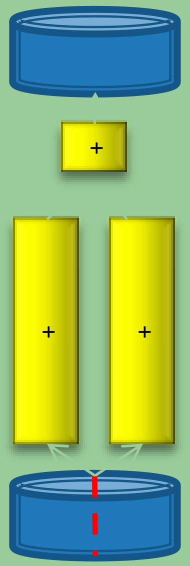
- Partitioning can be data-source sensitive
  - If a nested query, can fuse existing partitions
  - If an array, calculate strides and contiguous ranges (+spatial locality)
  - If a (possibly infinite) stream, lazily hand out chunks
- Partitioning can be operator sensitive
  - E.g. equi-joins employ a hashtable to turn an O(*nm*) "nested join" into O(*n*+*m*)
    - *Build* hash table out of one data source; then *probe* it for matches
    - Only works if all data elements in data source *A* with key *k* are in the same partition as those elements in data source *B* also with key *k*
    - We can use "hash partitioning" to accomplish this: for *p* partitions, calculate *k* for each element *e* in *A* and in *B*, and then assign to partition based on key, e.g. *k.GetHashCode() % p*
  - Output of sort: we can fuse, but restrict ordering, ordinal and key based
- Existing partitions might be repartitioned
  - Can't "push down" key partitioning information to leaves: types changed during stream data flow, e.g. *select* operator
  - Nesting: join processing output of another join operator
  - Or just to combat partition skew

Example: Query Nesting and Fusion

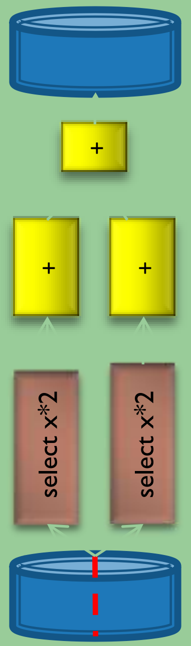
- Nesting queries inside of others is common
  - We can fuse partitions
    - `var q1 = from x in A select x*2;`
    - `var q2 = q1.Sum();`



1. Select (alone)



2. Sum (alone)



3. Select + Sum