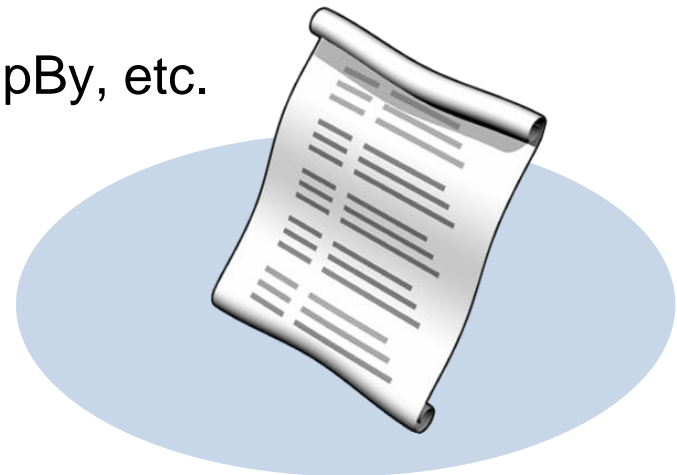


# LINQ to Objects

Introducing Language Integrated Query

# Objectives

- **LINQ architecture**
  - supporting language features (C# 3.0)
  - IEnumerable<T> iteration model
- **Query syntax**
  - from c in customers  
where City == "London"  
select c
- **Standard query operators**
  - Where, Select, OrderBy, GroupBy, etc.



# Problem [objects != data]

- **Difficult to treat objects as data**
  - code tends to be imperative (versus declarative)
  - developer's intention not always apparent
  - no standard query language for in-memory objects

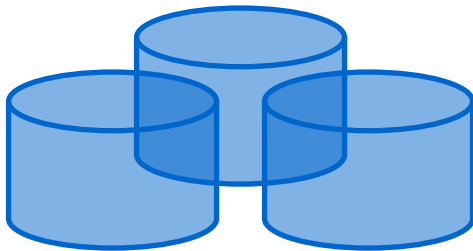
Query operations  
**mixed in** with the  
rest of your code

```
List<Customer> CustomersInLondon
(List<Customer> customers)
{
    List<Customer> result =
        new List<Customer>;
    foreach (Customer c in customers)
    {
        > if (City == "London")
            result.Add(c);
    }
    return result;
}
```

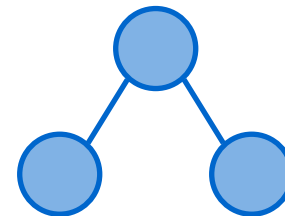
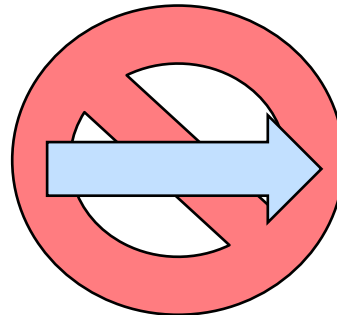


# Problem [data != objects]

- **Difficult to treat data as objects**
  - classic “object-relational” impedance mismatch
  - different type systems
  - different ways to model relationships



**Relational**



**Objects**

# LINQ Solution [language-integrated query]

- LINQ makes query **first-class** in the programming language
  - “sql-like” query syntax
  - supported initially by C# 3.0 and VB 9.0
  - enables type-safety, syntax-checking, intellisense

Declaratively  
expresses  
developer's  
intent

```
IEnumerable<Customer> custQuery =  
    from c in customers  
    where City == "London"  
    select c
```



# LINQ Solution [external data]

- **LINQ to SQL**
  - provides lightweight **object-relational mapper**
  - helps reduce (but not eliminate) object-relational impedance mismatch
- **LINQ to XML**
  - **expressive power** of XPath / XQuery
  - integrated into the programming language
- **Extensible architecture allows other query providers**
  - LINQ to **Dataset**
  - LINQ to **Entities** (next version of ADO.Net)
  - LINQ to X (SharePoint, LDAP, WMI, etc)



# LINQ Architecture [overview]

C# 3.0

VB 9.0

Others ...

## Language Integrated Query

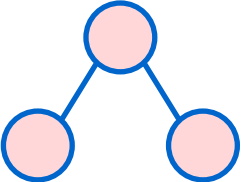
LINQ to  
Objects

LINQ to  
DataSets

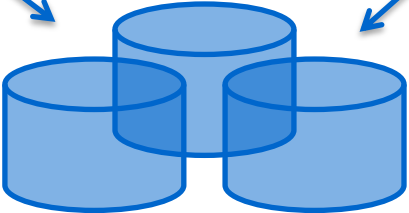
LINQ to  
SQL

LINQ to  
Entities

LINQ to  
XML



Objects



Relational



XML



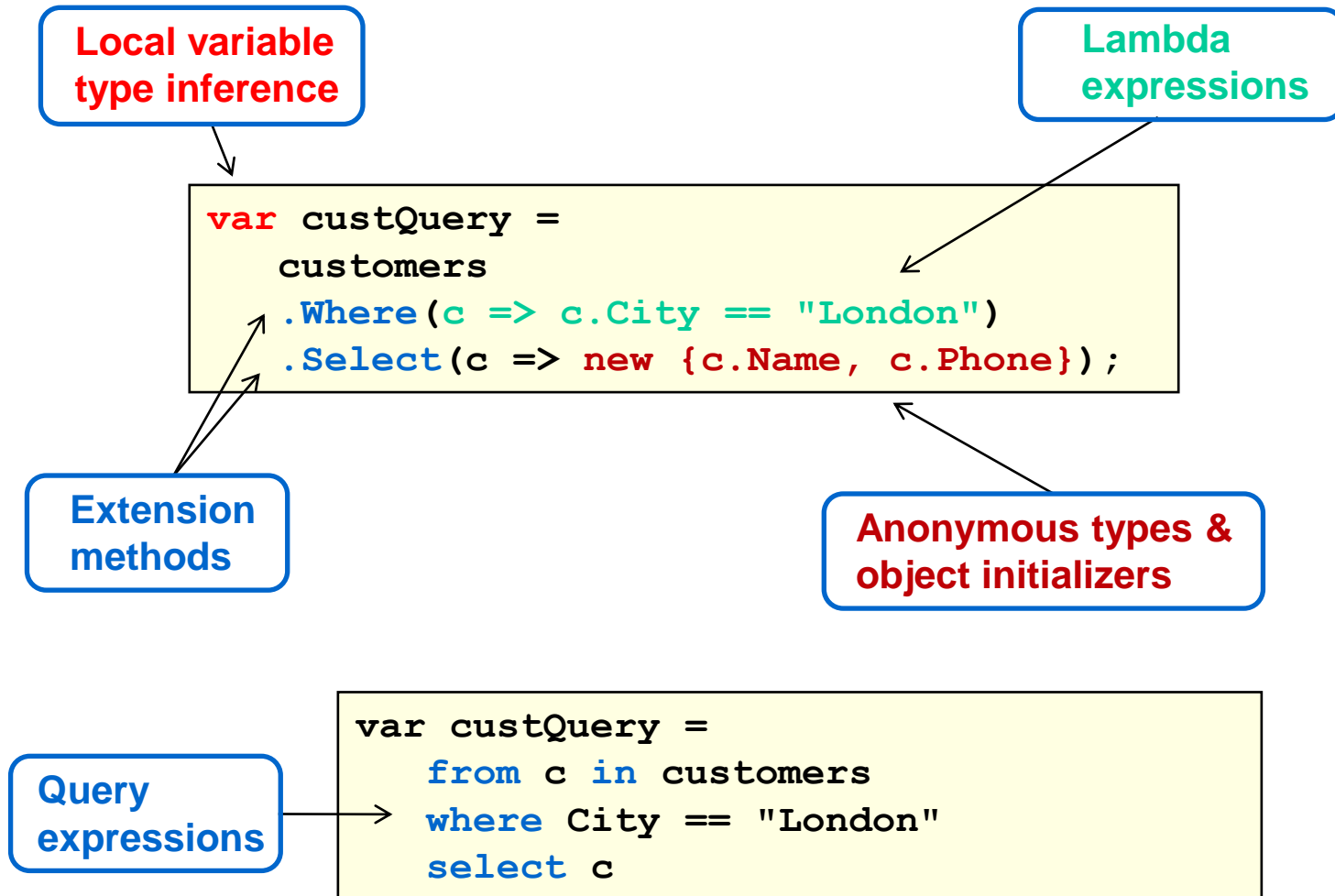
# LINQ Architecture [namespaces]

- Linq **classes** are contained in several **namespaces**

Assembly	Namespaces	Contents
System.Core.dll	System	Func generic delegate
	System.Linq	Enumerable, Queryable, companion Interfaces
	System.Linq.Expressions	classes that enable expression trees
System.Data.Linq.dll	System.Data.Linq	LINQ to SQL
System.Xml.Linq.dll	System.Xml.Linq	LINQ to XML
System.Data.DataSetExtensions.dll	System.Data	LINQ to DataSets

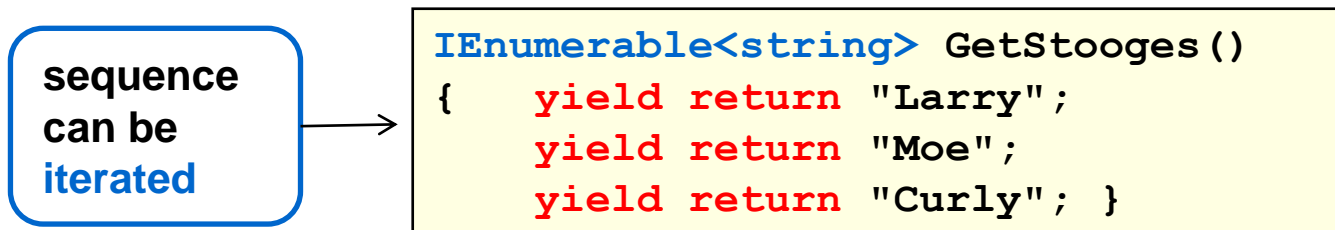


# LINQ Architecture [language enhancements]



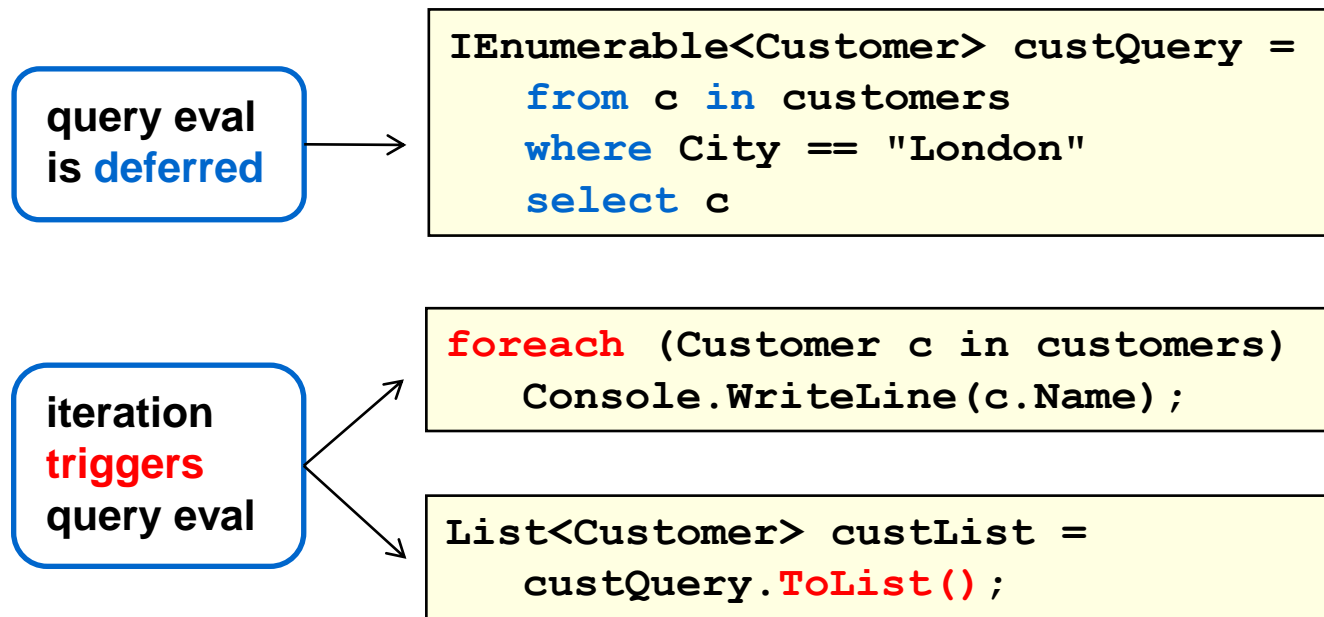
# IEnumerable<T> [iterators]

- LINQ works with any type implementing **IEnumerable<T>**
  - anything that can be **iterated** can be queried via LINQ
  - object-based collections (for ex, ArrayList) can also be queried
- LINQ uses extension methods that function as **iterators**
  - each method returns **IEnumerable<T>**
  - using **yield return** prompts the compiler to generate an **Enumerator** class



# Deferred query evaluation [iterators]

- Query operators simply return sequence that **can be** iterated
  - query execution is **deferred** until sequence **is** iterated
  - **foreach** triggers query evaluation: `IEnumerator.MoveNext`
  - **ToArray, ToList, ToDictionary** trigger query evaluation



# Func generic delegates

- Query operators accept a **Func** generic delegate
  - defined in the System namespace in `System.Core.dll`
  - last type argument used for **return value**

```
delegate TR Func<TR>()  
delegate TR Func<T, TR>(T arg)  
delegate TR Func<T1, T2, TR>(T1 arg1, T2 arg2)  
...
```

- For example, **Where** operator expects **Func<T, bool>**

**Func** delegate  
returns **bool**

```
Func<int, bool> predicate = i => i > 2;  
  
int[] ints = { 1, 2, 3, 4, 5 };  
IEnumerable<int> intQuery = ints  
    .Where(predicate);
```



# Query syntax [introduction]

- Working with **extension methods** and **lambda expressions** can be clumsy

```
string[] names = { "Larry", "Moe", "Curly", "Shemp" };  
IEnumerable<string> query = names  
    .Where(s => s.Length == 5)  
    .OrderBy(s => s)  
    .Select(s => s.ToUpper());
```

- **Query expressions** simplify syntax for common operators
  - Where, Select, OrderBy, GroupBy, Join (a few others too)

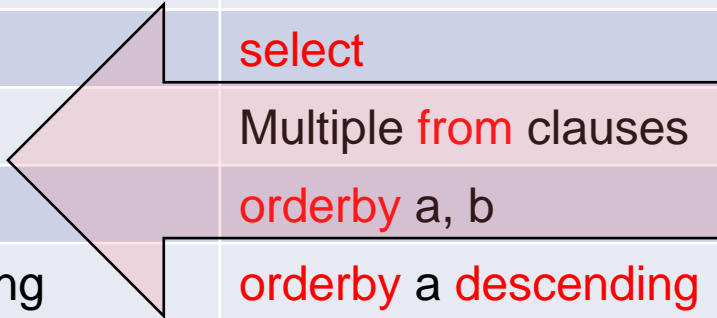
```
IEnumerable<string> query =  
    from s in names  
    where s.Length == 5  
    orderby s  
    select s.ToUpper();
```



# Query syntax [equivalent query operators]

- Compiler converts **query expressions** to **extension methods**

Query Operator	Query Syntax
Where	<code>where</code>
Cast	from <code>string</code> a in b [explicitly typed]
Select	<code>select</code>
SelectMany	Multiple <code>from</code> clauses
OrderBy, ThenBy	<code>orderby</code> a, b
OrderByDescending	<code>orderby</code> a <code>descending</code>
GroupBy	<code>group</code> a <code>by</code> b <code>group</code> a <code>by</code> b <code>into</code> c
Join	<code>join</code> a in b <code>on</code> c <code>equals</code> d
GroupJoin	<code>join</code> a in b <code>on</code> c <code>equals</code> d <code>into</code> e



# Query syntax [rules]

- **Must begin with from, end with select or group**
  - **in between:** from, let, where, join, orderby, into

```
var query = from s in names where s.Length > 3
            group s by s.Length;
```

- **Sometimes you have to mix and match**
  - most **query operators** have no equivalent query syntax
  - combine the two approaches using **parentheses**

```
int count = ( from s in names
              where s.Length == 5
              orderby s
              select s.ToUpper() ).Count();
```



## Query syntax [let expression]

- The **let** keyword allows you to declare an inline **variable**
  - you can then **reference** the let variable later in the query

```
var booksQuery = from b in books
    let tax = b.Price * .0825M
    let total = b.Price + tax
    where total > 20
    select new { b.Title, b.Price, Tax=tax, Total=total };
```

- You can also use **let** to reference an enumerable **sequence**
  - follow with another from clause to query the sequence

```
var booksQuery = from b in books
    let selAuths = from a in b.Authors
        where a.LastName.Length > 4 select a
    select new { b.Title, SelectedAuthors=selAuths };
```



# Standard query operators [deferred]

## Main standard query operators returning `IEnumerable<T>`

- brought into scope with `using System.Linq`
- query execution is **deferred** until iterated
- **Restriction: `Where`, `OfType`**
  - filter a sequence; filter non-generic sequences
- **Projection: `Select`, `SelectMany`**
  - partition results based on a key
- **Ordering: `OrderBy`, `ThenBy`** (also descending)
  - chain ordering operators to sort on any number of items
- **Grouping: `GroupBy`**
  - partition results based on a common element
- **Joins: `Join`, `GroupJoin`**
  - combine two sequences based on matching keys



# Standard query operators [immediate]

Main standard query operators **NOT** returning **IEnumerable<T>**

- query execution is **immediate**
- **Aggregation: Aggregate, Count, Sum, Min, Max, Average**
  - perform calculations over a result or a group
- **Conversion: ToArray, ToList, ToDictionary**
  - perform iteration to populate a collection
- **Many more ...**



# Where [restriction]

- **Where** query operator performs restriction (filtering)
  - `Func<T, bool>` performs a test on each element

Where  
operator

```
Func<Book, bool> predicate =  
    b => b.Title[0] > 'B';  
  
IEnumerable<Book> booksQuery =  
    books.Where(predicate);
```

where  
expression

```
IEnumerable<Book> booksQuery =  
    from b in books  
    where b.Title[0] > 'B'  
    select b;
```



# Select [projection]

- **Select** query operator performs projection
  - `Func<T, TResult>` accepts one type and returns another

Select  
operator

```
Func<Book, int> selector =  
    b => b.Title.Length;  
  
IEnumerable<int> booksQuery =  
    books.Select(selector);
```

select  
expression

```
IEnumerable<int> booksQuery =  
    from b in books  
    select b.Title.Length;
```



# Select with var [projection]

- **Select** can return an **anonymous type**
  - using an object initializer without type name
  - must use **var** to reference compiler-generated type

Select  
operator  
+ anon type

```
var booksQuery =  
    books.Select( b => new  
        { Title = b.Title,  
          TitleLength = b.Title.Length } );
```

select  
expression  
+ anon type

```
var booksQuery =  
    from b in books  
    select new  
        { Title = b.Title,  
          TitleLength = b.Title.Length } ;
```



# SelectMany [projection]

- **SelectMany** used to create **one-to-many** output
  - `Func<T, IEnumerable<S>>` returns children of the source
  - chain **from** expressions to use SelectMany with **query syntax**

SelectMany  
operator

```
Func<Book, IEnumerable<Author>>  
    selector = b => b.Authors;  
  
IEnumerable<Author> booksQuery =  
    books.SelectMany(selector);
```

2<sup>nd</sup> from  
returns  
children of 1<sup>st</sup>

```
IEnumerable<Author> booksQuery =  
    from b in books  
    from a in b.Authors  
    select a;
```



# OrderBy, ThenBy [sorting]

- **OrderBy** sorts based on a **key selector**
  - `Func<T, K>` returns the **key** used to sort the sequence
- **ThenBy** used to add additional sort criteria
  - can specify **descending** sorts to reverse sort order

OrderBy  
operator

```
Func<Book, string> keySelector =  
    b => b.Title;  
  
IEnumerable<Book> booksQuery =  
    books.OrderBy(keySelector);
```

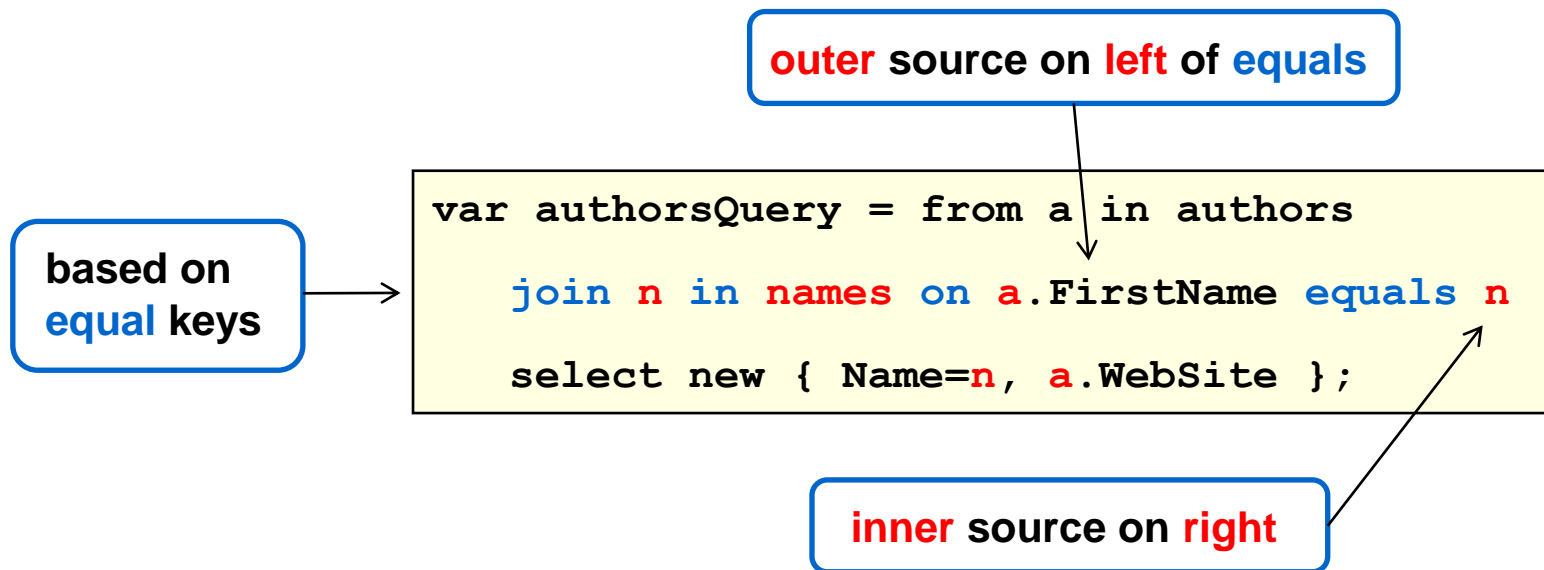
orderby  
expression

```
IEnumerable<Book> booksQuery =  
    from b in books  
    orderby b.Price descending, b.Title  
    select b;
```



# Join [joining]

- **Join** creates a sequence from **two sources** based on **keys**
  - only needed if classes not “related” (for ex, book.Authors)
  - result can combine fields from both sources
- Performs an **“inner”** join where keys **match** on each source
  - only key equality supported (not > or <) – hence **equals** \*



# GroupBy [grouping]

- **GroupBy** groups based on a **key selector**
  - `Func<T, K>` returns the **key** used to group the sequence
- Returns sequence of **IGrouping<TKey, T>**
  - derives from `IEnumerable<T>`, each group has **children** items

GroupBy  
operator

```
Func<Book, decimal> keySelector =  
    b => b.Price;  
  
IEnumerable<IGrouping<decimal, Book>>  
booksQuery =  
    books.GroupBy(keySelector);
```

group by  
expression

```
IEnumerable<IGrouping<decimal, Book>>  
booksQuery =  
    from b in books  
    group b by b.Price;
```



# GroupBy with into [grouping]

- Sometimes you want to do something with the group
  - the **into** keyword creates a **variable** representing the group
  - for example, you can **order** the groups or perform a **calculation**

create  
group  
variable

```
IEnumerable<IGrouping<decimal, Book>>  
booksQuery =  
from b in books  
group b by b.Price into g  
orderby g.Key descending  
select g;
```

iterate  
group

```
foreach (var priceGroup in booksQuery)  
{ Console.WriteLine(priceGroup.Key);  
  foreach (Book book in priceGroup)  
    Console.WriteLine(book); }
```



# GroupJoin [joining and grouping]

- Use **GroupJoin** to join on a sequence and project as **children**
  - place **into** after a **join** to create a **variable** for the group
  - place the variable in select to create a **hierarchical** result

reference  
joined  
group

```
var authorsQuery = from a in authors
    join p in personHobbies
        on a.FirstName equals p.Name
    into hobbiesGroup
select new { a.Name,
    Hobbies = hobbiesGroup };
```

iterate  
children

```
foreach (var a in authorsQuery)
{
    Console.WriteLine(a.Name);
    foreach (PersonHobby h in a.Hobbies)
        Console.WriteLine("\t{0}",
            h.Hobby);
}
```



# Count, Sum, Min, Max, Average [aggregation]

- Aggregation methods perform a **calculation** over a **range**
  - standard operators are **Count, Sum, Min, Max, Average**

operators  
return a  
single value

```
int[] ints = { 1, 3, 5, 7, 9};  
int c = ints.Count();  
int s = ints.Sum(i => i);  
int a = ints.Average(i => i);
```

- Custom aggregation performed with **Aggregate** method
  - returns value set on each iteration

aggregate  
items in  
string array

```
string[] names = {"Burke", "Connor",  
"Frank", "Everett", "Albert" };  
string result = names  
    .Aggregate( (n,s) => s += ", " + n );
```



# ToArray, ToList, ToDictionary [conversion]

- Conversion operators convert sequence to a **collection**
  - Invoking operator causes **immediate** query evaluation

query eval  
**deferred**  
until iterated

```
IEnumerable<string> query =  
    from n in names  
    where n.Length == 5  
    select n;
```

query **eval'd**  
and results  
**stored**

```
List<string> result = query.ToList();  
  
string[] result = query.ToArray();  
  
Dictionary<string, string>  
    result = query  
        .ToDictionary( s => s );
```



# Summary

- **LINQ helps bridge the gap between objects and data**
  - treat **objects as data** (LINQ to Objects)
  - treat **data as objects** (LINQ to SQL, XML, etc.)
- **LINQ to Objects based on delegates**
  - operators are **extension methods** in System.Linq.Enumerable
  - can replace with your own custom implementation
- **IEnumerable<T> iteration pattern**
  - results in **deferred query execution**
- **Query Expressions**
  - allow more **compact syntax** that is easier to read (sometimes)
- **Standard Query Operators**
  - provide **rich query engine**: restriction, projection, sorting, joining, grouping, aggregation, conversion (many more ...)

