LINQ

Test your understanding

- What is a query?
- What query language do you use for RDBMS?
- What query language do you use for XML?

What is LINQ

- Language-Integrated Query
- New from .NET 3.5 and VS 2008
- Enables using same query language for disparate data sources- SQL, XML or, web services, .NET objects.
- Also enables usage of queries against collections.
- Object –oriented query language
- VS 2008 offers IntelliSense support.
- Namespace **`System.Linq`** provides the LINQ support.

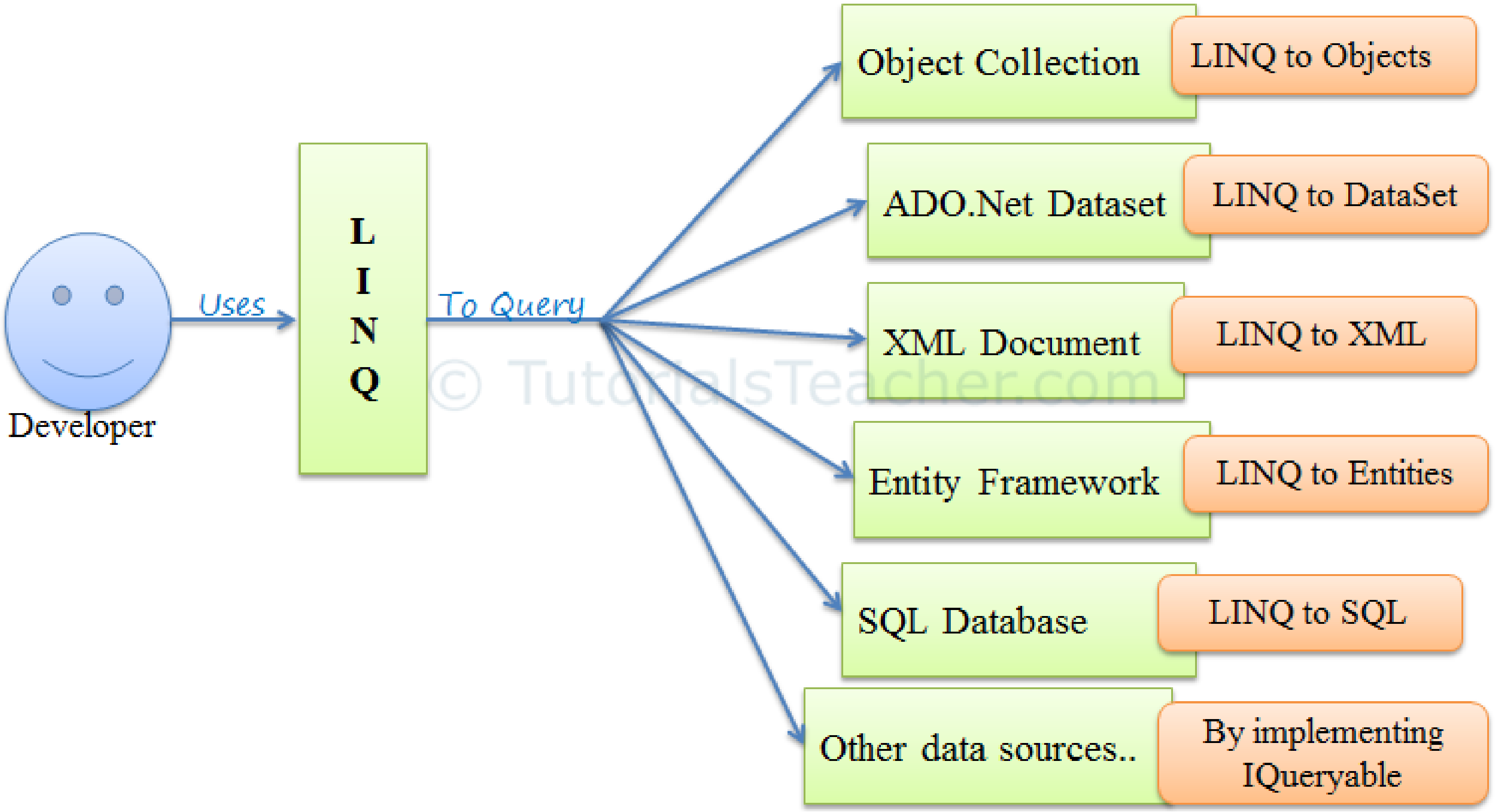
General form of the query

- Declarative query syntax :

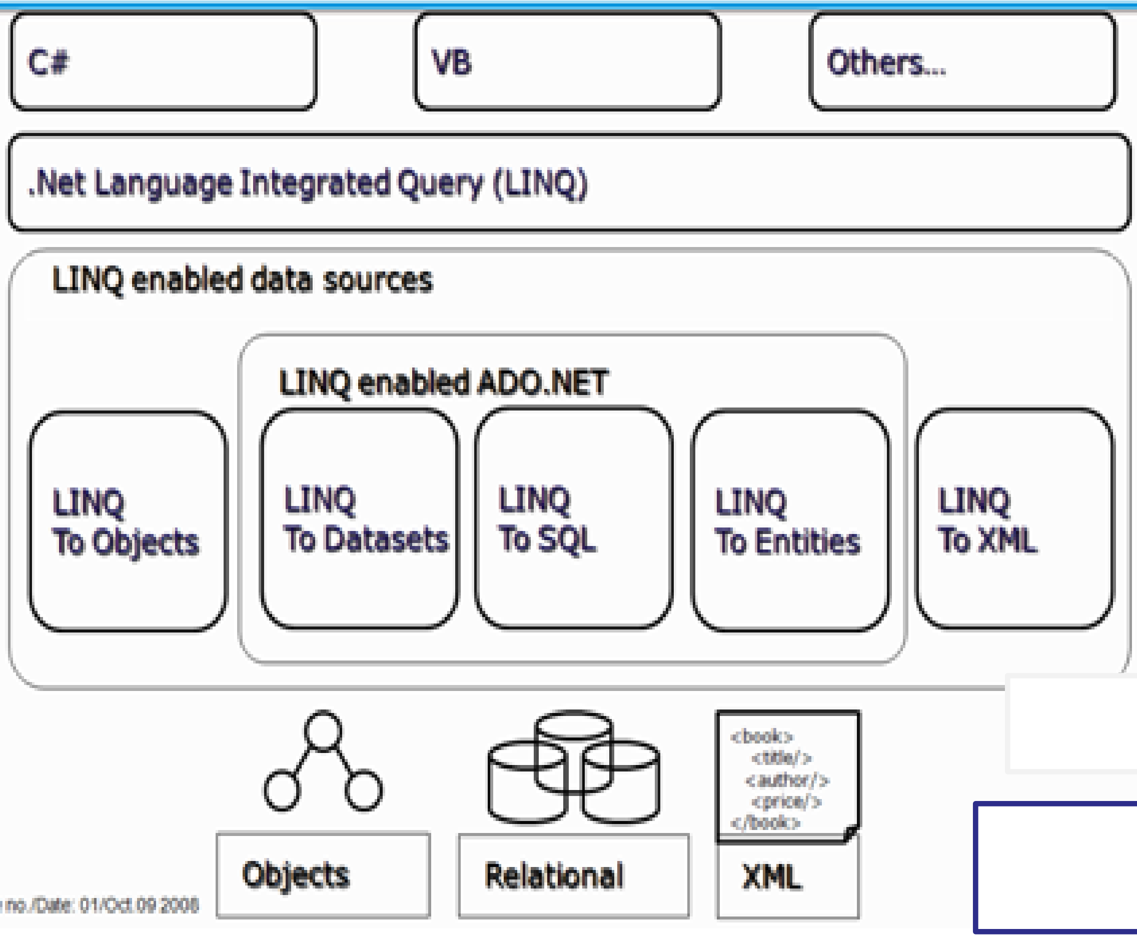
```
 var x= from item in data_source
                 where condition
                 select item
```

- The query variable x only stores the query commands
- The actual execution happens only when some operation is requested like iteration. This is refereed to as *deferred execution.*
- While the syntax allows usage of “var” keyword, what the query really returns is a **`IEnumerable`** object.
- Hence **`foreach`** can be used with the result of LINQ.
- Note that LINQ query is case sensitive.

LINQ


Developer
Uses
L I N Q
To Query
Object Collection
LINQ to Objects
ADO.Net Dataset
LINQ to DataSet
XML Document
LINQ to XML
Entity Framework
LINQ to Entities
SQL Database
LINQ to SQL
Other data sources..
By implementing IQueryable

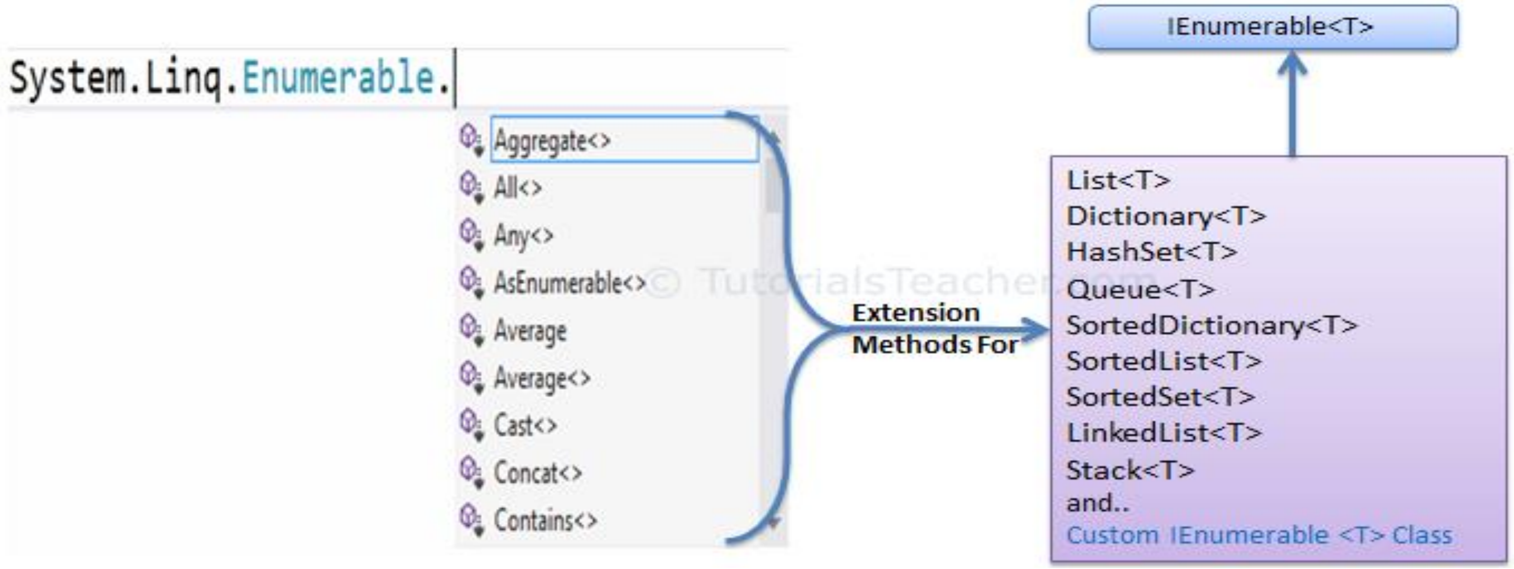
LINQ Architecture



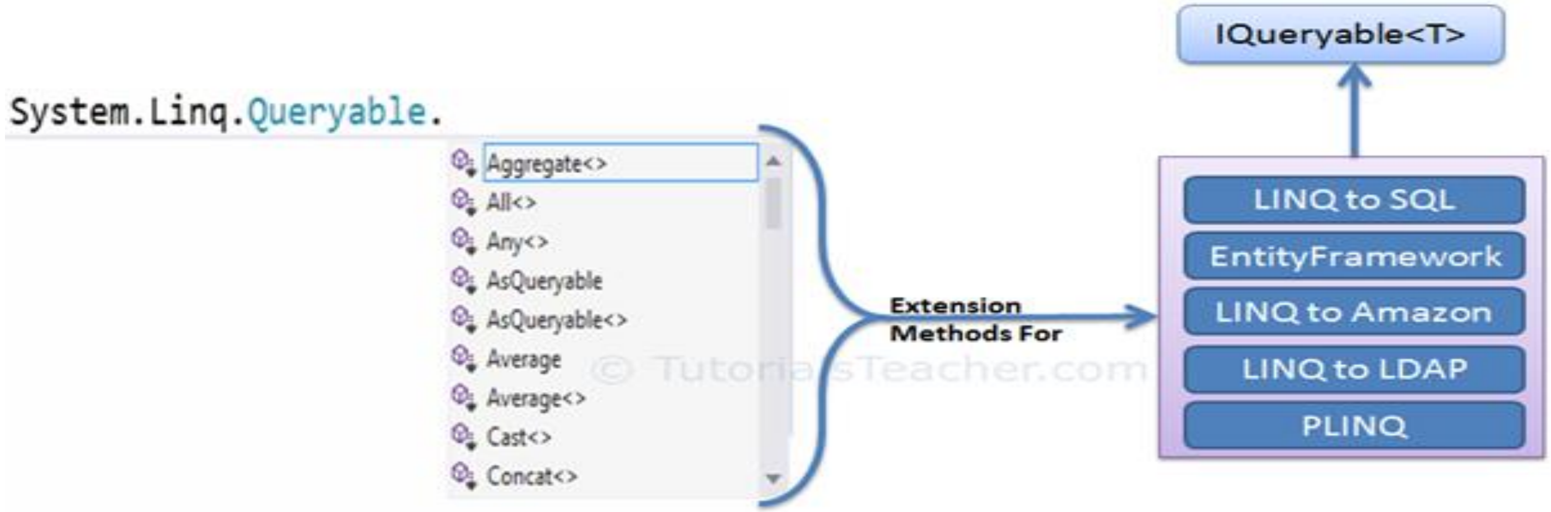
Core Assemblies in LINQ

- The core assemblies in LINQ are:
  - using System.Linq
    - Provides Classes & Interface to support LINQ Queries
  - using System.Collections.Generic
    - Allows the user to create Strongly Typed collections that provide type safety and performance (LINQ to Objects)
  - using System.Data.Linq
    - Provides the functionality to access relational databases (LINQ to SQL)
  - using System.Xml.Linq
    - Provides the functionality for accessing XML documents using LINQ (LINQ to XML)
  - using System.Data.Entity
    - Provides the functionality to access relational databases (Entity Framework)

LINQ



System.Linq.Enumerable.
Aggregate<>
All<>
Any<>
AsEnumerable<>
Average
Average<>
Cast<>
Concat<>
Contains<>
Extension Methods For
IEnumerable<T>
List<T>
Dictionary<T>
HashSet<T>
Queue<T>
SortedDictionary<T>
SortedList<T>
SortedSet<T>
LinkedList<T>
Stack<T>
and..
Custom IEnumerable <T> Class



System.Linq.Queryable.
Aggregate<>
All<>
Any<>
AsQueryable
AsQueryable<>
Average
Average<>
Cast<>
Concat<>
Extension Methods For
IQueryable<T>
LINQ to SQL
EntityFramework
LINQ to Amazon
LINQ to LDAP
PLINQ

LINQ Query Syntax

- There are two basic ways to write a LINQ query to IEnumerable collection or IQueryable data sources.
  - Query Syntax or Query Expression Syntax
  - Method Syntax or Method extension syntax
- Query Syntax:
- Query syntax is similar to SQL (Structured Query Language) for the database. It is defined within the C# or VB code.

from *<range variable>* in *<IEnumerable<T> or IQueryable<T> Collection>*
<Standard Query Operators>
<select or groupBy operator> *<result formation>*

LINQ Query Syntax

- Declarative query syntax :

```
var x= from item in data_source
                where condition
                select item
```

- The query variable x only stores the query commands
- The actual execution happens only when some operation is requested like iteration. This is refereed to as *deferred execution.*
- While the syntax allows usage of “var” keyword, what the query really returns is a **`IEnumerable`** object.
- Hence **`foreach`** can be used with the result of LINQ.
- Note that LINQ query is case sensitive.

Standard Query Operators

Classification	Standard Query Operators
Filtering	Where, OfType
Sorting	OrderBy, OrderByDescending, ThenBy, ThenByDescending, Reverse
Grouping	GroupBy, ToLookup
Join	GroupJoin, Join
Projection	Select, SelectMany
Aggregation	Aggregate, Average, Count, LongCount, Max, Min, Sum
Quantifiers	All, Any, Contains
Elements	ElementAt, ElementAtOrDefault, First, FirstOrDefault, Last, LastOrDefault, Single, SingleOrDefault
Set	Distinct, Except, Intersect, Union
Partitioning	Skip, SkipWhile, Take, TakeWhile
Concatenation	Concat
Equality	SequenceEqual
Generation	DefaultEmpty, Empty, Range, Repeat
Conversion	AsEnumerable, AsQueryable, Cast, ToArray, ToDictionary, ToList

LINQ with Objects

- The term "LINQ to Objects" refers to the use of LINQ queries Objects that implement **`IEnumerable`** , meaning all collection classes like **`List`**, **`Dictionary`** as well as arrays and string can use LINQ.
- The collection name become the data source.

```
var x= from item in data_source
            where condition
            select item
```

- The **`from`** clause similar to the **`for-each`** statement.
- An identifier is used to refer to individual item in the collection. The **`where`** clause uses this identifier name to filter the collection.
- The query returns **`IEnumerable`** object.
- The assembly **`System.Linq`** needs to be included to work with LINQ.

Example : LINQ with array

```
using System;
using System.Linq;
class Program{
 static void Main(string[] args) {
   string[] flowers = { "dahlia", "rose", "lotus",
   "lily", "hibiscus", "daffodil" };
```

Or `IEnumerable<string>`

↑

```
   var fQuery =
   from flower in flowers
   where (flower.StartsWith("d"))
   select flower;
   foreach (string f in fQuery) {
                Console.WriteLine(f);
   }  }}
```

Execution happens here

On execution :

```
dahlia
daffodil
```

The same query can be run multiple times since query itself produce any results.

Example : LINQ with string

- `String` is nothing but an array of characters.
- Therefore LINQ query can be used with the string to search based on characters.

```
class Program{
    static void Main(string[] args)          {
      string poem = @"What is this life if, full of care,
                  We have no time to stand and stare.
                  No time to stand beneath the boughs
                  And stare as long as sheep or cows.
                  No time to see, when woods we pass,
            Where squirrels hide their nuts in grass.
                 No time to see, in broad daylight,
            Streams full of stars, like skies at night.
                 No time to turn at Beauty's glance,
                 And watch her feet, how they can dance.
                 No time to wait till her mouth can
                 Enrich that smile her eyes began.
                 A poor life this if, full of
                 We have no time to stand and
```

```
 var matchQuery = from c in poem
                  where c == ','
                  select c;
      int i = 0;
      foreach (char c in matchQuery)
            {
                i++;
            }
            Console.WriteLine(i);
        }
```

On execution : **11**

More on select and from clause

- Select can be used to return a computed value as well.

```
var fQuery =
        from flower in flowers
        where (flower.StartsWith("d"))
        select flower.ToUpper();
```

- For the collection that implements `IEnumerable<T>` it is not compulsory to specify the type in the from clause. But for the collection that implements `IEnumerable,` the type has to be specified in from clause

```
select flower;
var fQuery =
from string flower in flowers
where (flower.StartsWith("d"))
   select flower;
```

Exercise

- *Given an array of numbers. Find the cube of the numbers that are greater than 100 but less than 1000 using LINQ.*
- *Change some of the array elements and execute the same query again.*
- *Hint : use the logical operators of C# to combine the conditions*

(15 mins)

Multiple `where` clause and `let`

- Query can have any number of where clause to filter that data.

```
var fQuery =from flower in flowers
        where flower.StartsWith("d")
        where flower.Length>7
        select flower;
```

- This is same as

```
 var fQuery =  from flower in flowers
        where flower.StartsWith("d") || flower.Length>7
        select flower;
```

- The keyword `let` can be used retain temporary value.

```
var lquery = from flower in flowers

      let len = flower.Length

       where len > 5 && len <7

       select flower;
```

Compound from clauses

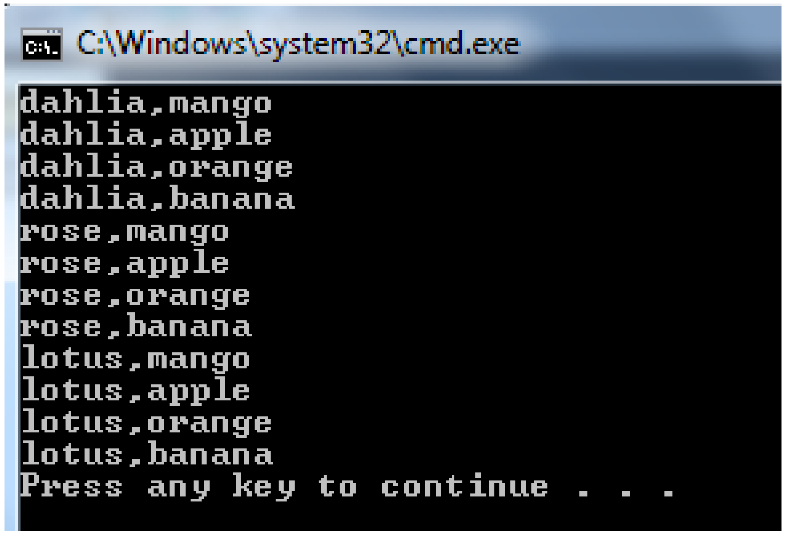
- Data from multiple data sources can be obtained using multiple from clause. The example listed results in producing Cartesian product between the two data sources.

```
using System;
using System.Linq;
using System.Collections.Generic;

struct Flowerfruit{
    public string flower;
    public string fruit;
    public Flowerfruit(string fl, string fr)    {
        flower=fl;
        fruit = fr;
    }
}
```

```
class Program{
    static void Main(string[] args)    {
        string[] flowers = { "dahlia", "rose", "lotus" };
        string[] fruits = { "mango", "apple", "orange",
"banana" };

        var fQuery =
        from flower in flowers
        from fruit in fruits
        select  new Flowerfruit(flower, fruit);
        foreach (Flowerfruit f in fQuery)
        {
            Console.WriteLine(f.flower+"," +f.fruit);
        }
    }
}
```


C:\Windows\system32\cmd.exe

```
dahlia,mango
dahlia,apple
dahlia,orange
dahlia,banana
rose,mango
rose,apple
rose,orange
rose,banana
lotus,mango
lotus,apple
lotus,orange
lotus,banana
Press any key to continue . . .
```

Exercise

- *Given a list of participants for a tennis match. Split the list into 2 equal halves and display all the possible combination of matches possible between the participants in the two lists. A condition is that no player should have an opponent who is from his own his own country.*

(45 mins)

Sorting

- `orderby` clause is used to sort on one or more fields.
- `orderby` default arranges the elements in ascending order.
- `orderby ascending or order by descending` can also be used to arranges the elements in ascending order or descending order.

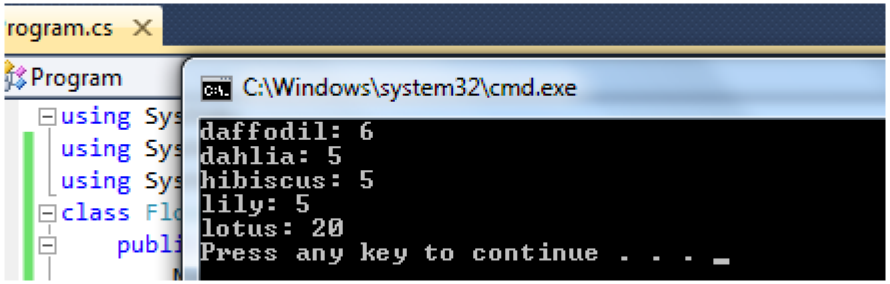
```
using System;
using System.Linq;
using System.Collections.Generic;
class Flower{
    public Flower(string n, int p)    {
        Name = n;
        Petals = p;    }
    public string Name { get; set; }
    public int Petals { get; set; }
}
```

```
class Program{
 static void Main(string[] args) {
List<Flower> FlowerList = new List<Flower>();
FlowerList.Add(new Flower("dahlia", 5));
FlowerList.Add(new Flower("lotus", 20));
FlowerList.Add(new Flower("lily", 5));
FlowerList.Add(new Flower("daffodil", 6));
FlowerList.Add(new Flower("hibiscus", 5));

// Using LINQ with Collections
var lquery = from Flower flower in FlowerList
             where flower.Petals > 4
             orderby flower.Name , flower.Petals descending
             select flower;

foreach (Flower f in lquery)
    Console.WriteLine(f.Name + ": " + f.Petals);
 }
}
```

Note that we need to specify the type here



```
daffodil: 6
dahlia: 5
hibiscus: 5
lily: 5
lotus: 20
Press any key to continue . . .
```

Note how collection has been used in LINQ.

Exercise

- *Create an Order class that has order id, item name, order date and quantity. Create a collection of Order objects. Display the data day wise from most recently ordered to least recently ordered and by quantity from highest to lowest.*

Hint: Use order date type as System.DateTime . Use DateTime(int year, int month, int day) constructor.

(45 mins)

Exercise

- *Create an Order class that has order id, item name, order date and quantity. Create a collection of Order objects. Display the data day wise from most recently ordered to least recently ordered and by quantity from highest to lowest.*

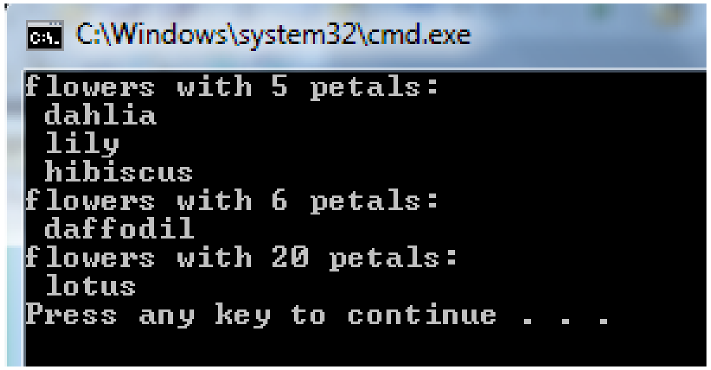
Hint: Use order date type as System.DateTime . Use DateTime(int year, int month, int day) constructor.

(45 mins)

group

- A LINQ query starts with **from** clause and end with either a **select** clause or **group** clause.
- **group** clause allows grouping the results with respect to certain criteria.

```
var lquery = from Flower flower in FlowerList
                     orderby flower.Petals
                     group flower by flower.Petals;
foreach (var f in lquery) {
         Console.WriteLine("flowers with "+ f.Key+ " 
petals: ");
            foreach (var nm in f)
                Console.WriteLine(" "+nm.Name);
```



C:\Windows\system32\cmd.exe

```
flowers with 5 petals:
 dahlia
 lily
 hibiscus
flowers with 6 petals:
 daffodil
flowers with 20 petals:
 lotus
Press any key to continue . . .
```

Exercise

- *For the previous exercise, write a LINQ query that displays the details grouped by the month in the descending order of the order* date.

(30 mins)

Joining

- Joining refers to combining data from two data sources based on some common fields in both the data sources.
- Syntax:

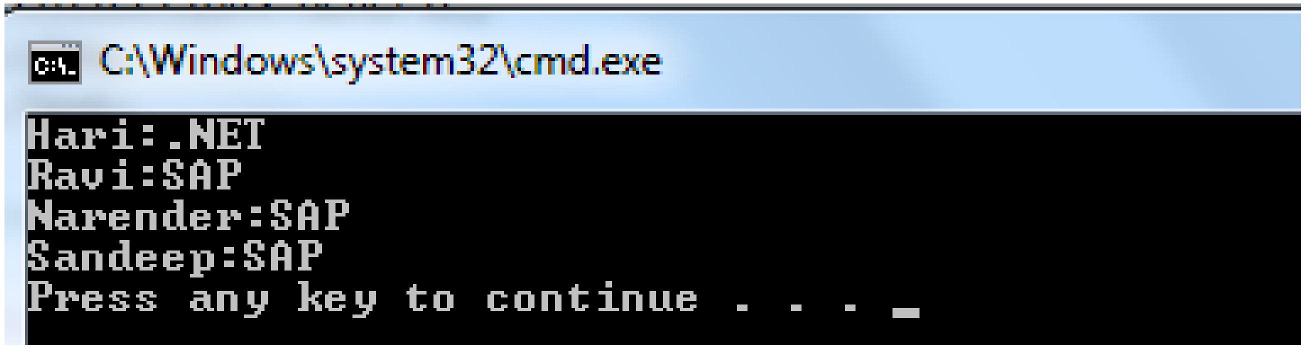
```
from var1 in DataSource1
join var2 in DataSource2
on var1.property equals var2.property
```

Example

```
class Student  {
     public int Id{get; set;}
     public string Name { get; set; }
     public Student(int id, string name) {
         this.Id = id;
         this.Name = name;
     }
  }
  class Enroll  {
      public int Id { get; set; }
      public string CourseName{ get; set; }
      public Enroll(int id, string name) {
         this.Id = id;
         this.CourseName = name;
     }
  }
```

```
class StudentEnroll  {
     public int Id { get; set; }
     public string Name { get; set; }
     public string CourseName { get; set; }
     public StudentEnroll(int id, string name,
                               string cname) {
        this.Id = id;
        this.Name = name;
        this.CourseName = cname;
    }
 }
class Program    {
       static void Main(string[] args) {
           Student[] students = { new Student(1,
"Hari"), new Student(2, "Ravi"), new Student(3,
"Narender"), new Student(4, "Sandeep") };
           Enroll[] enrollments = { new Enroll(1,
".NET"), new Enroll(2, "SAP"), new Enroll(3, "SAP"), new
Enroll(4, "SAP") };
var join = from s in students
      join e in enrollments on s.Id equals e.Id
select new StudentEnroll(s.Id, s.Name, e.CourseName);

 foreach (var ex in join) {
  Console.WriteLine(ex.Name + ":" + ex.CourseName);
   }
  }
 }
```



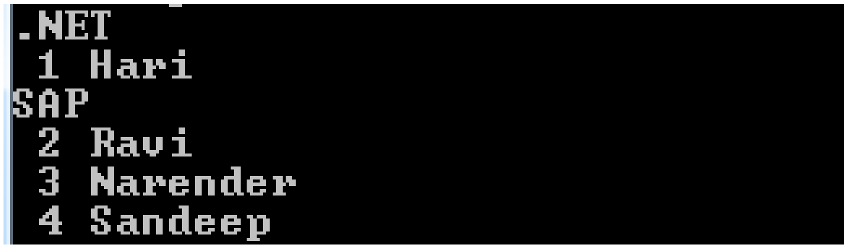
C:\Windows\system32\cmd.exe

```
Hari:.NET
Ravi:SAP
Narender:SAP
Sandeep:SAP
Press any key to continue . . . _
```

Query continuation

- The temporary results can be saved and can be used in the subsequent part of the query. This is called query continuation or just continuation.
- `into` clause is used to achieve this.
- If we need the result of the previous example grouped by the course name then the query would be:

```
 var join1 = from s in students
join e in enrollments on s.Id equals e.Id
select new StudentEnroll(s.Id, s.Name, e.CourseName) into
es
group es by es.CourseName;
     foreach (var ex in join1) {
      Console.WriteLine(ex.Key);
         foreach (var ex1 in ex)
             Console.WriteLine(" "+ex1.Id+ " "+ ex1.Name);}
```



```
.NET
 1 Hari
SAP
 2 Ravi
 3 Narender
 4 Sandeep
```

Exercise

- *You have created Order class in the previous exercise and that has order id , item name, order date and quantity . Create another class called Item that has item name and price. Write a LINQ query such that it returns order id, item name, order date and the total price (price * quantity) grouped by the month in the descending order of the order* date.

(45 mins)

Anonymous types

- Instead of creating a class for maintaining relationship, like **`StudentEnroll`** class, C# allows creation of anonymous types (class with no name).
- This is used with the select clause to return an object in cases where there are no class to represent the object (as a result of join or if the query has only subset of the fields in a data source).
- Syntax: **`new { field1=value1, field2=value2, …}`**
- Example:

```
 var join = from s in students
 join e in enrollments on s.Id equals e.Id
 select new { ID = s.Id, Name = s.Name, CName =
 e.CourseName };
foreach (var ex in join) {
     Console.WriteLine(ex.Name + ":" + ex.CName); }
```

An anonymous class with ID, Name and CName read-only properties are created.

Exercise

- *Do the previous exercise using anonymous types.*

(30 mins)

Query Methods

- Query methods provide a short cut way of writing queries.
- These methods can be used on any enumerable object.
- `System.Linq.Enumerable` methods have query methods and these extend the functionality of `IEnumerable<T>`
- Methods:
  - `Select`
  - `Where`
  - `OrderBy, OrderByDescending`
  - `Join`
  - `GroupBy`

Example: `var l= FlowerList.Select(e => e.`

Lambda expression

- The conditional expression that the LINQ uses with the where clause is actually passed as an argument to the Where method:
  `Where(flower.Petals => flower.Petals == 5)`.
- The above expression is called Lambda expression.
- A lambda expression is an anonymous function that can be used to create delegates or expression tree types.
- **=>** is the lambda operator, which is read as "goes to".
- While many LINQ can be written without the knowledge of Lambda expression, some queries can only be expressed in method syntax and require us to use lambda expressions.
- It is very easy to use aggregate functions with lambda expression.

Lambda expression example

```
using System;
class X
{
    delegate int cube(int i);
    static void Main(string[] args)
    {
        cube myDelegate = x => x*x*x;
        int j = myDelegate(5);
        Console.Write(j);
    }
}

//Prints 125
```

Using Lambda expression in LINQ

- This example converts the first example that we created using lambda expression.

```
string[] flowers = { "dahlia", "rose", "lotus",
"lily", "hibiscus", "daffodil" };
IEnumerable<string>f =
flowers.Where(flower=>flower.StartsWith("d"));
foreach (string g in f)  {
      Console.WriteLine(g);
}
```

Examples using query methods

For the students and enrollments collection

1.
   ```
   var q=  students.Where(s => s.Id == 4).Select(st =>
   st);
   ```

2.
   ```
   var q=  students.OrderBy(s=>s.Name).Select(st => st);
   ```

3.
   ```
   var q=  enrollments.GroupBy(e=> e.CourseName);
   ```

4.
   ```
   var q = students.Join(enrollments, s => s.Id, e =>
   e.Id, (s, e) => new { s.Name, e.CourseName });
   ```

5.
   ```
   var q = (students.Join(enrollments, s => s.Id, e =>
   e.Id, (s, e) => new { s.Name, e.CourseName
   })).GroupBy(k => k.CourseName);
   ```

The last one is the same query as the example for Query continuation

Other LINQ Methods

- `All()`
- `Any()`
- `Contains()`

Uses the `Equals()` method of the class to determine if the element specified is in the collection.

- `First()`
- `Last()`

```
Console.WriteLine(students.Any(x => x.Id > 5)); // False
```

Exercise

- *Check if all the quantities in the Order collection is >0.*
- *Get the name of the item that was ordered in largest quantity in a single order. (Hint: use LINQ methods to sort)*
- *Find if there are any orders placed before Jan of this year.*

(30 mins)

Aggregate methods

- `Count()`
- `Sum()`
- `Min()`
- `Max()`
- `Average()`

- These LINQ methods produce single (non-sequential) result. So in such cases, immediate execution takes place.

```
Console.WriteLine(flowers.Count(x =>x.StartsWith("d")));
```

Example: Count

```
string poem = @"What is this life if, full of care,
      We have no time to stand and stare.
      No time to stand beneath the boughs
      And stare as long as sheep or cows.
      No time to see, when woods we pass,
      Where squirrels hide their nuts in grass.
      No time to see, in broad daylight,
      Streams full of stars, like skies at night.
      No time to turn at Beauty's glance,
      And watch her feet, how they can dance.
      No time to wait till her mouth can
      Enrich that smile her eyes began.
      A poor life this if, full of care,
      We have no time to stand and stare";
      int i = (from c in poem where c == ','select c).Count();
      Console.WriteLine(i);
      }
```

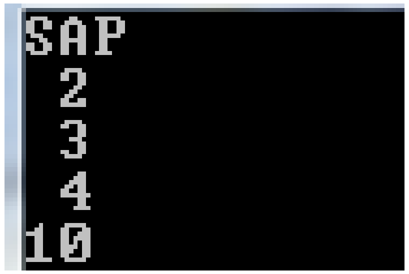
Prints: 11

Query count with continuation

- The temporary results can be saved and can be used in the subsequent part of the query.
- This is called query continuation or just continuation.
- `into` clause is used to achieve this.

```
var lquery = from Enroll s in enrollments
             group s by s.CourseName into fs
             where fs.Count() > 1
             select fs;

             foreach (var f in lquery) {
                 Console.WriteLine(f.Key );
                 foreach (var nm in f)
                     Console.WriteLine("  " + nm.Id);
             }
```



```
SAP
 2
 3
 4
10
```

Activity

- *Rewrite the last two example of that used Count using LINQ query methods entirely.*

(15 minutes)

Exercise

- *Given the array of numbers. Count and display even numbers.*

  (15 minutes)

- *Write LINQ to get the sum of quantities for each item and also find out and display the item that has overall maximum orders.*

  (45 mins)

Distinct

- `Distinct()` returns result set without the duplicate values.

```
int[] nums = { 1, 1, 2, 3, 5, 7, 5 };
  var results = nums.Select(e=>e).Distinct();
foreach (var n in results){
Console.WriteLine(n);
}
```

LINQ with XML

- .NET provides a set of new class are provided to work with LINQ.
- **`XDocument , XElement, XAttribute`** represent an xml document, element and attribute.

`XElement` members

- `Name, Value, FirstNode, LastNode, FirstAttribute, LastAttribute, HasAttribute, NextNode, PrevNode, Document, Document, NodeType`
- `void Add(Object), void AddFirst(Object),`
- `XmlReader CreateReader() , XmlWriter CreateWriter()`
- `void Load(String), void Load(Stream), void Load(XmlWriter), void Load(TextReader)`
- `void Remove(), void RemoveAll()`
- `void Save(String), void Save(Stream), void Save( (XmlWriter), void Save(TextReader)`
- `XDocument` also more or less has same members.

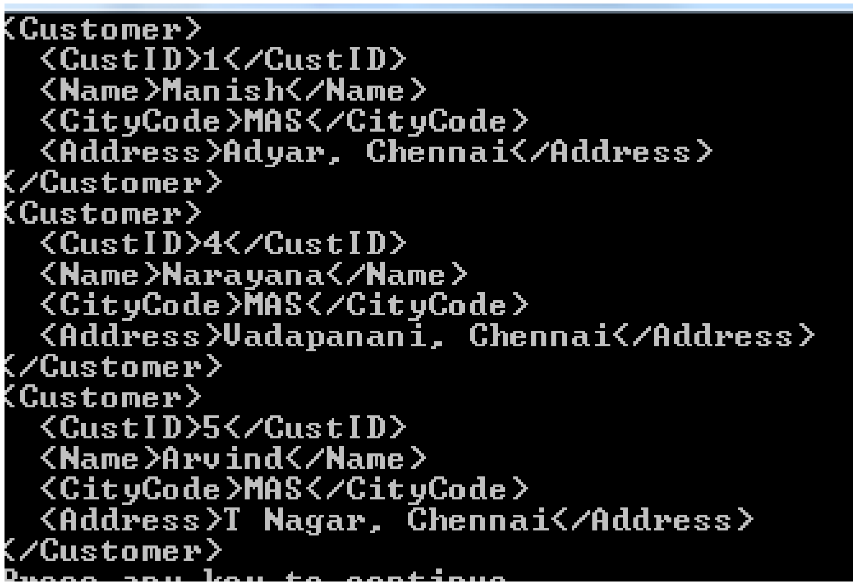
Example: filtering xml document

- This example returns only those customers who live in chennai from the xml document given below:

```
<?xml version="1.0" standalone="yes"?>
<Customers>
  <Customer>
    <CustID>1</CustID>
    <Name>Manish</Name>
    <CityCode>MAS</CityCode>
    <Address>Adyar, Chennai</Address>
  </Customer>
  <Customer>
    <CustID>2</CustID>
    <Name>Priya</Name>
    <CityCode>BLR</CityCode>
    <Address>Banashankari, Bangalore</Address>
  </Customer>
<Customer>
    <CustID>3</CustID>
    <Name>Surya</Name>
    <CityCode>KOL</CityCode>
    <Address>Park Street, Kolkatta</Address>
  </Customer>
<Customer>
    <CustID>4</CustID>
    <Name>Narayana</Name>
    <CityCode>MAS</CityCode>
    <Address>Vadapanani, Chennai</Address>
  </Customer>
  <Customer>
    <CustID>5</CustID>
    <Name>Arvind</Name>
    <CityCode>MAS</CityCode>
    <Address>T Nagar, Chennai</Address>
  </Customer>
  <Customer>
    <CustID>6</CustID>
    <Name>Hari</Name>
    <CityCode>KOL</CityCode>
    <Address>Kalayani, Kolkata</Address>
  </Customer>
</Customers>
```

```
 class Program {
        static void Main(string[] args) {
            string fname=@"E:\cust.xml";
            XElement Customers = XElement.Load(fname);

            IEnumerable<XElement> custs =
                        from cust in
Customers.Descendants("Customer")
                        where cust.Element("CityCode").Value
== "MAS"
                         select cust;
       foreach (XElement e in custs) {
                 Console.WriteLine(e);
            }
}
```



```
<Customer>
  <CustID>1</CustID>
  <Name>Manish</Name>
  <CityCode>MAS</CityCode>
  <Address>Adyar, Chennai</Address>
</Customer>
<Customer>
  <CustID>4</CustID>
  <Name>Narayana</Name>
  <CityCode>MAS</CityCode>
  <Address>Vadapanani, Chennai</Address>
</Customer>
<Customer>
  <CustID>5</CustID>
  <Name>Arvind</Name>
  <CityCode>MAS</CityCode>
  <Address>T Nagar, Chennai</Address>
</Customer>
Press any key to continue
```

LINQ way of building a XML tree

```
class Program    {
        static void Main(string[] args)   {
             XElement Employees =
                    new XElement("Employees",
                     new XElement("Employee",
                     new XElement("Name", "Sahana"),
                     new XElement("Phone", "9915550144",
                        new XAttribute("Type", "Home")),
                        new XElement("phone", "9195550145",
                        new XAttribute("Type", "Work")) ),

                     new XElement("Employee",
                     new XElement("Name", "Anjana"),
                     new XElement("Phone", "9215550144",
                        new XAttribute("Type", "Home")),
                        new XElement("phone", "9134550145",
                        new XAttribute("Type", "Work")) ));

            Employees.Save("E:/Employees.xml");
}
```

On execution the Employees.xml file that was generated :

```
<?xml version="1.0" encoding="utf-8"?>
<Employees>
  <Employee>
    <Name>Sahana</Name>
    <Phone Type="Home">9915550144</Phone>
    <phone Type="Work">9195550145</phone>
  </Employee>
  <Employee>
    <Name>Anjana</Name>
    <Phone Type="Home">9215550144</Phone>
    <phone Type="Work">9134550145</phone>
  </Employee>
</Employees>
```

Exercise

- *Look at the XML given below:*

```
<?xml version="1.0" encoding='UTF-8'?>
<employees>
 <person id="1234">
      <name>Gayathri Sardar</name>
      <office>Arihant Towers, Kandanchavadi</office>
      <city>Chennai</city>
 </person>
<person id="2345">
      <name>Bobby Mahajan</name>
      <office>SJR, Electronic City</office>
      <city>Bangalore</city>
</person>
</employees>
```

1. *Write a program to accept input and build the XML like the one.*
2. *Write a program to print the names of the person who are from Chennai*

(1 hour)