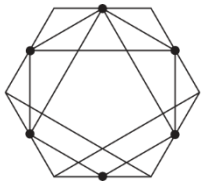
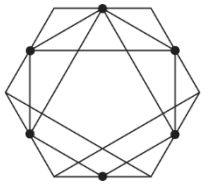
LINQ – LANGUAGE INTEGRATED QUERY



OVERVIEW

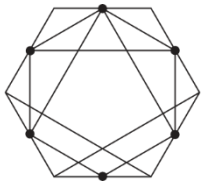
- LINQ Overview
- Lambda Expressions & Anonymous Types
- Method Syntax vs. Query Syntax
- Some LINQ Operators
- Demo & Exercise



LINQ – LANGUAGE INTEGRATED QUERY

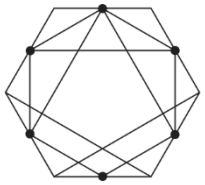
- Query language integrated directly in the programming language

- LINQ providers allow querying lots of different data sources:
  - LINQ to Objects (regular .NET objects)
  - LINQ to XML
  - LINQ to SQL
  - Custom providers can be implemented



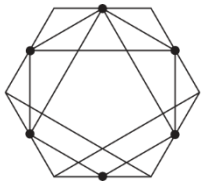
LINQ – LANGUAGE INTEGRATED QUERY

- “Lazy Evaluation”: expressions are only computed when needed
- Queries can be expressed based on “Lambda expressions”
- Types can be constructed dynamically within the query (“Anonymous types”)
- Different syntax types:
  - Method Syntax – “regular C# methods”
  - Query Syntax – like SQL



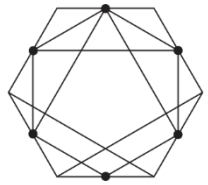
LAMBDA EXPRESSIONS

- Anonymous methods without an identifier
- Based on “Lambda Calculus” (Alonzo Church, 1936)
- One of the key approaches in the LISP programming language
- Can today be found in many programming languages and are very handy in certain situations



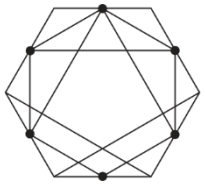
ANONYMOUS TYPES

- Anonymous types have no name explicitly assigned to them
- However, the compiler dynamically infers a type and provides full type checking
- Read-only in C#



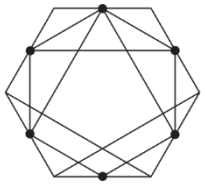
DEMO: ANONYMOUS TYPES & LAMBDA EXPRESSIONS

- Anonymous Types
- Lambda Expressions



LINQ

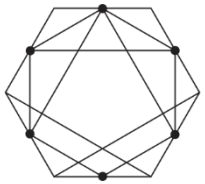
- Two general ways of using LINQ:
  - Method Syntax – collection.Where(…).Select(…)
  - Query Syntax – from c in collection where … select …



WHERE & SELECT

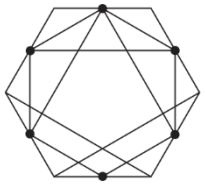
- “Where” filters a sequence based on a predicate

- “Select” projects each element of a sequence to a new form
  - Use existing elements directly
  - Construct new anonymous types based on the elements’ properties



Where & Select

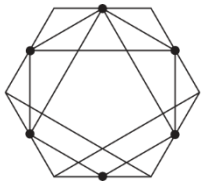
```
List<int> numbers = new List<int>() { 0, 1, 2, 3, 4, 5, 6, 7 };

// method syntax
var even = numbers.Where(number => number % 2 == 0);

// query syntax
var odd =
    from number in numbers
    where number % 2 == 1
    select number;
```



SELECT MANY

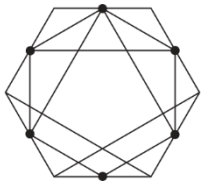
- “SelectMany” projects and flattens several sequences into a single sequence

- A two-dimensional “sequence of sequences” is combined into a one-dimensional sequence



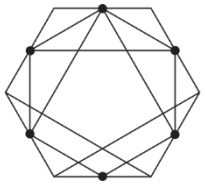
SelectMany

```
List<List<int>> listsOfNumbers = new List<List<int>>()
{
    new List<int>() { 0, 2, 4 },
    new List<int>() { 1, 3, 5 },
    new List<int>() { 0, 1, 2 }
};

var flattenedListMethod = listsOfNumbers.SelectMany(numbers => numbers);

var flattenedListQuery =
    from numbers in listsOfNumbers
    from n in numbers
    select n;
```



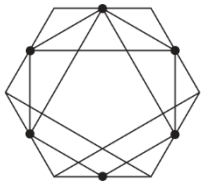
ORDER BY

- Ordering requires selecting a key according to which a sequence should be ordered
- Order By – Ascending
- Order By Desc - Descending



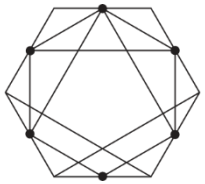
OrderBy

```
List<int> numbers = new List<int>() { 4, 3, 2, 1, 4, 2, 0, 7 };

var numbersDesc = numbers.OrderByDescending(n => n);

var numbersAsc = from n in numbers orderby n select n;
```



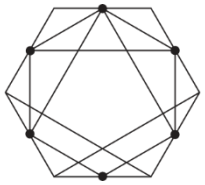
SUM

- Computes the sum of a sequence of numeric values
- For a list of only numeric values **.Sum()** can be used
- For complex types, a selector can be defined **.Sum(x => x.SomeNumber)**



Sum

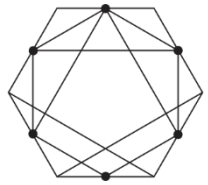
```
List<int> numbers = new List<int>() { 0, 1, 2, 3, 4, 5, 6, 7 };

int sum = numbers.Sum();
```



JOIN

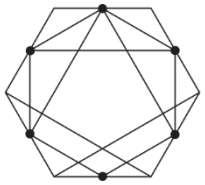
- Two sequences can be joined based on a matching common value (“key”)

- A join requires:
  - Outer table + key selector
  - Inner table + key selector
  - Result selector



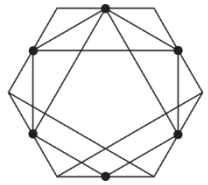
```
var productInformation = new[]
{
    new { ID = 0, Description = "Description 0"},
    new { ID = 1, Description = "Description 1"},
    new { ID = 2, Description = "Description 2"}
};

var productPrice = new[]
{
    new { ID = 0, Price = 2.99m},
    new { ID = 1, Price = 1.50m},
    new { ID = 2, Price = 1.00m}
};

var joinMethod =
    productInformation.Join( // outer table
    productPrice, // inner table
    pi => pi.ID, // outer key
    pd => pd.ID, // innter key
    (pi, pd) => new { pi.ID, pi.Description, pd.Price }); // result

var joinQuery =
    from pi in productInformation // outer table
    join pd in productPrice // inner table
    on pi.ID equals pd.ID // outer and inner keys
    select new { pi.ID, pi.Description, pd.Price }; // result
```

Join



DEMO: BANK ACCOUNTS

- Banks
- Accounts
- Credit Ratings



EXERCISE: FRUIT SUPPLIERS

- Implement the TODO’s in the FruitSupply Project