

# Module 03

"LINQ"



**TEKNOLOGISK**  
**INSTITUT**

# Agenda

- ▶ **Introducing LINQ**
- ▶ LINQ Query Keywords
- ▶ LINQ Query Operator Methods
- ▶ LINQ to Entities
- ▶ LINQ to XML
- ▶ Expression Trees
- ▶ Lab 3
- ▶ Discussion and Review

# Motivation for LINQ

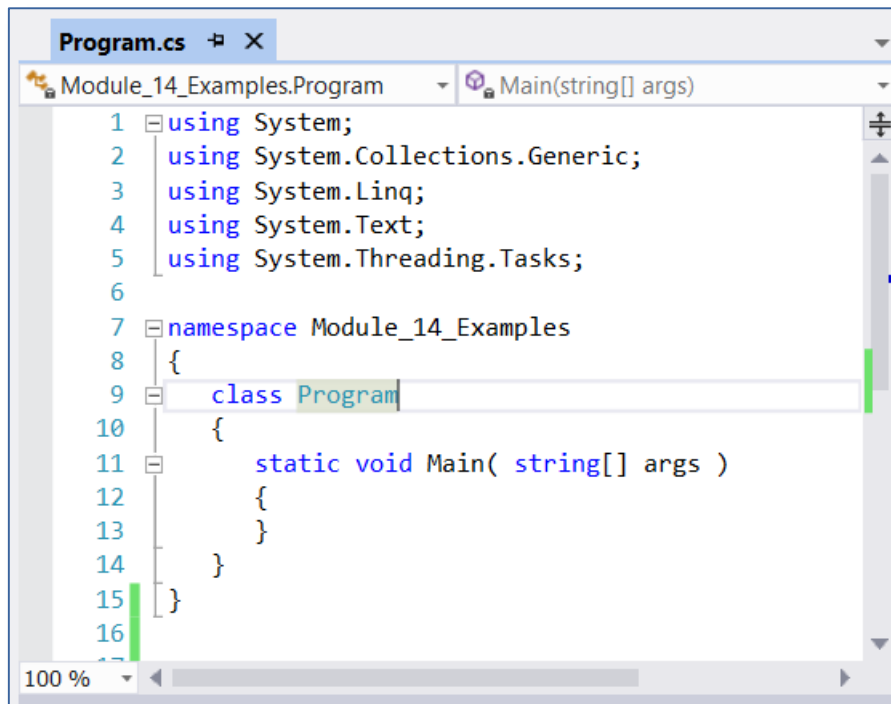
- ▶ LINQ = Language **IN**tegrated **Q**uery
  
- ▶ Several distinct motivations for LINQ
  - Uniform programming model for any kind of data
  - A better tool for embedding SQL queries into type-safe code
  - Another data abstraction layer
  - ...
  
- ▶ All of these descriptions to some extent hold true

# LINQ Components

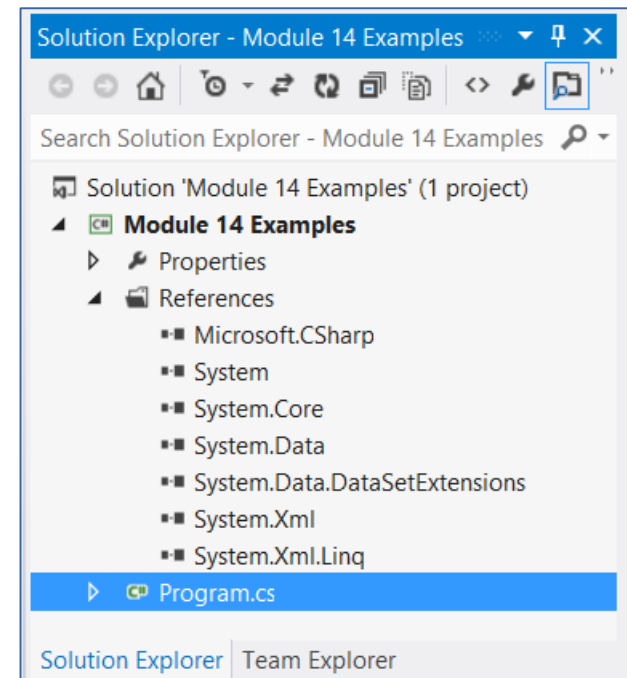
- ▶ LINQ to Objects
  - ▶ LINQ to XML
  - ▶ LINQ to SQL
  - ▶ LINQ to DataSet
  - ▶ LINQ to Entities
  - ▶ Parallel LINQ
  - ▶ ...
- 
- ▶ Later we will see
    - LINQ to Entities
    - LINQ to XML
    - Parallel LINQ (in Module 5)

# Starting LINQ to Objects

- ▶ Main LINQ features live in **System.Core.dll** in the **System.Linq** namespace



```
1 using System;
2 using System.Collections.Generic;
3 using System.Linq;
4 using System.Text;
5 using System.Threading.Tasks;
6
7 namespace Module_14_Examples
8 {
9     class Program
10     {
11         static void Main( string[] args )
12         {
13         }
14     }
15 }
16
17
```



# A First Example

- ▶ Find all games with more than 18 characters in the title

```
string[] wiiGames = {  
    "Super Mario Galaxy",  
    "FIFA 09",  
    "Guitar Hero III",  
    "Wii Sports",  
    "Wii Fit",  
    "Legend of Zelda: Twilight Princess"  
};
```

```
IEnumerable<string> query = from g in wiiGames  
                             where g.Length >= 18  
                             select g;
```

```
foreach( string s in query )  
{  
    Console.WriteLine( s );  
}
```



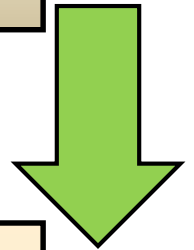
# Implicitly Typed Variables

- ▶ Query results can be of a multitude of types

```
int[] numbers = {10, 20, 30, 40, 1, 2, 3, 8};  
IEnumerable<int> query = from i in numbers  
                        where i < 10 select i;  
foreach( int i in query )  
{  
    Console.WriteLine( i );  
}
```

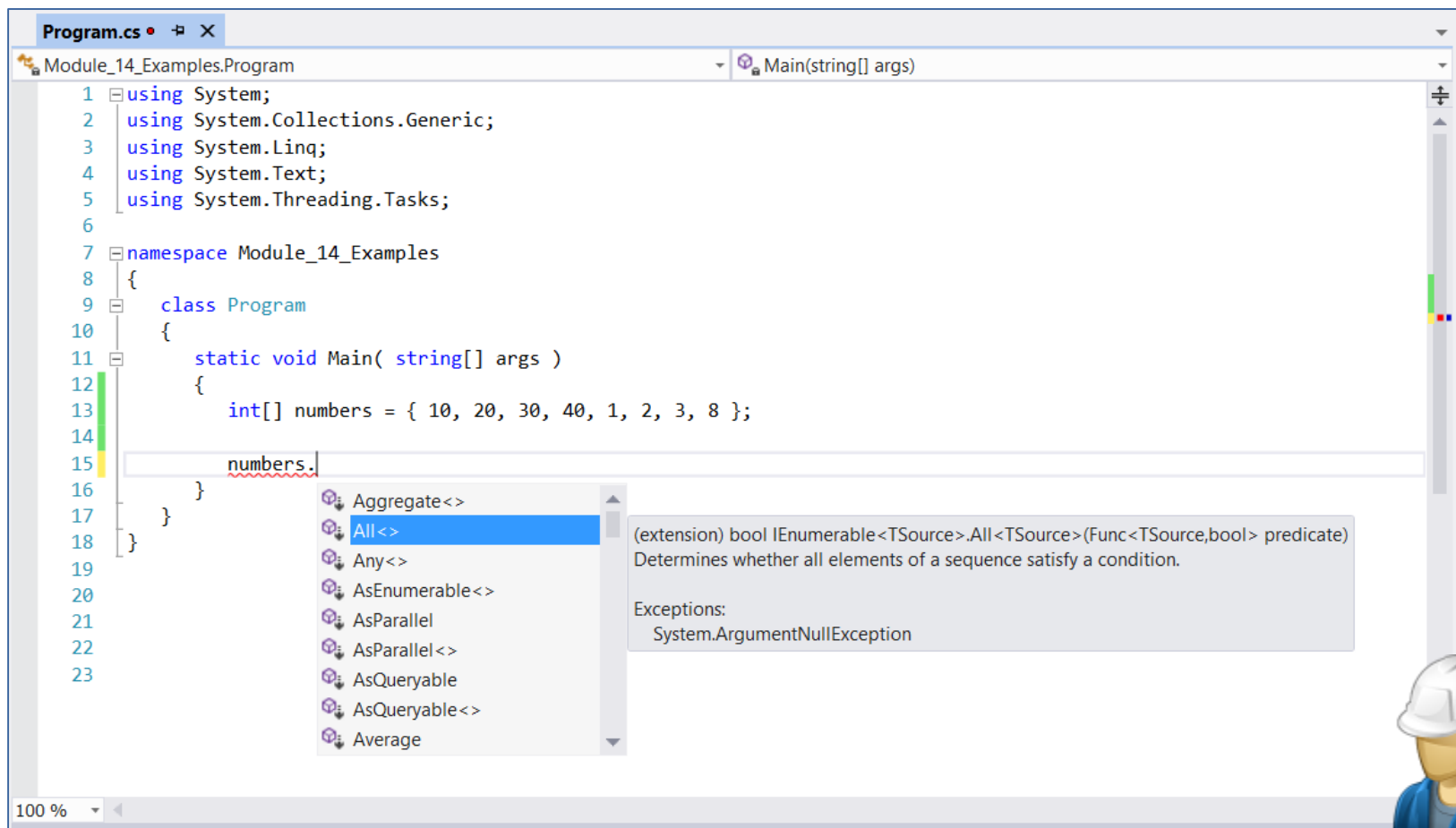
- ▶ Innocently-looking modifications might change underlying type
- ▶ Make all query variables implicitly typed...!

```
int[] numbers = {10, 20, 30, 40, 1, 2, 3, 8};  
var query = from i in numbers where i < 10 select i;  
foreach( var i in query )  
{  
    Console.WriteLine( i );  
}
```



# Enumerable Extension Methods

- ▶ The **System.Linq.Enumerable** class provides a lot of extension methods



```

Program.cs
Module_14_Examples.Program
Main(string[] args)
1 using System;
2 using System.Collections.Generic;
3 using System.Linq;
4 using System.Text;
5 using System.Threading.Tasks;
6
7 namespace Module_14_Examples
8 {
9     class Program
10    {
11        static void Main( string[] args )
12        {
13            int[] numbers = { 10, 20, 30, 40, 1, 2, 3, 8 };
14
15            numbers.
16        }
17    }
18 }
19
20
21
22
23

```

IntelliSense dropdown for `numbers.`:

- Aggregate<>
- All<> (extension) bool IEnumerable<TSource>.All<TSource>(Func<TSource,bool> predicate) Determines whether all elements of a sequence satisfy a condition. Exceptions: System.ArgumentNullException
- Any<>
- AsEnumerable<>
- AsParallel
- AsParallel<>
- AsQueryable
- AsQueryable<>
- Average





# Deferred Execution

- ▶ Query expressions are not evaluated until they're enumerated!
- ▶ This is called *Deferred Execution*

```
int[] numbers = { 10, 20, 30, 40, 0, 1, 2, 3, 8 };  
var query = from i in numbers where i < 10 select 87 / i;  
  
foreach( var i in query )  
{  
    Console.WriteLine( i );  
}
```

- ▶ You can force evaluation through the Visual Studio debugger
  - Use the Results View of the query variable



# Immediate Execution

- ▶ You can force evaluation by using conversion extension methods

```
int[] numbers = { 10, 20, 30, 40, 0, 1, 2, 3, 8 };  
var query = from i in numbers where i < 10 select i;  
  
int[] intNumbers = query.ToArray();  
List<int> listNumbers = query.ToList();
```

- ▶ There are other such extension methods, e.g.
  - **ToDictionary<T,K>**



# LINQ and Generic Collections

- ▶ LINQ can query data in various members of `System.Collections.Generic`

```
Stack<int> stack = new Stack<int>( new int[]{ 42, 87, 112, 255 } );  
var query = from i in stack where i < 100 select i;
```

```
List<Car> cars = new List<Car>() {  
    new Car{ PetName="Henry", Color="Silver", Speed=100, Make="VW" },  
    new Car{ PetName="Daisy", Color="Tan", Speed=90, Make="BMW" },  
    new Car{ PetName="Mary", Color="Black", Speed=55, Make="VW" },  
    new Car{ PetName="Clunker", Color="Rust", Speed=5, Make="Yugo" },  
    new Car{ PetName="Melvin", Color="White", Speed=43, Make="Ford" }  
};
```

```
var query = from c in cars  
    where c.Speed > 90 && c.Make == "BMW"  
    select c;
```



# LINQ and Nongeneric Collections

- ▶ Nongeneric collections lack the **IEnumerable<T>** infrastructure for querying
- ▶ This can be provided using the **OfType<T>** extension method

```
ArrayList cars = new ArrayList() {  
    new Car{ PetName="Henry", Color="Silver", Speed=100, Make="BMW" },  
    new Car{ PetName="Daisy", Color="Tan", Speed=90, Make="BMW" },  
    new Car{ PetName="Mary", Color="Black", Speed=55, Make="VW" },  
    new Car{ PetName="Clunker", Color="Rust", Speed=5, Make="Yugo" },  
    new Car{ PetName="Melvin", Color="White", Speed=43, Make="Ford" }  
};
```

```
IEnumerable<Car> enumerableCars = cars.OfType<Car>();  
var query = from c in enumerableCars  
            where c.Speed > 90 && c.Make == "BMW"  
            select c;
```



# LINQ and Custom Collections

- ▶ LINQ queries can be performed directly on any **IEnumerable<T>** type
  - Even your own types!

```
Node<int> tree = new Node<int>(
    42,
    new Node<int>( ... ),
    new Node<int>( 256 )
);

var query = from i in tree
             where i % 2 == 0
             select i;
```

```
class Node<T> : IEnumerable<T>
{
    protected T _value;

    protected Node<T> _left;
    protected Node<T> _right;

    ...
}
```



# Agenda

- ▶ Introducing LINQ
- ▶ **LINQ Query Keywords**
- ▶ LINQ Query Operator Methods
- ▶ LINQ to Entities
- ▶ LINQ to XML
- ▶ Expression Trees
- ▶ Lab 3
- ▶ Discussion and Review

# The **from** Clause

- ▶ Range variables and data source are specified in the **from** clause

```
Stack<int> stack = new Stack<int>( new int[]{ 42, 87, 112, 255} );  
var query = from i in stack where i < 10 select i;
```

- ▶ It can define the type of the range variable as well

```
ArrayList cars = new ArrayList {  
    new Car{ PetName="Henry", Color="Silver", Speed=100, Make="BMW" },  
    ...  
};  
var query = from Car c in cars  
            where c.Speed > 90 && c.Make == "BMW"  
            select c;
```

- ▶ Can in fact have multiple **from** clauses...



# The **where** Clause

- ▶ Filtering conditions are specified by a boolean expression in a **where** clause

```
List<Car> cars = new List<Car> {  
    new Car{ PetName="Henry", Color="Silver", Speed=100, Make="BMW" },  
    ...  
};  
var query = from c in cars  
             where c.Speed > 90 && c.Make == "BMW"  
             select c;
```

```
var query = from c in cars  
             where c.Speed > 90  
             where SomePredicate( c )  
             select c;
```

- ▶ Can have multiple **where** clauses also





# The **select** Clause

- ▶ Projections of results are done through the **select** clause

```
List<Car> cars = new List<Car> {  
    new Car{ PetName="Henry", Color="Silver", Speed=100, Make="BMW" },  
    ...  
};  
var query = from c in cars  
            where c.Speed > 90 && c.Make == "BMW"  
            select c.Make;
```

```
var query = from c in cars  
            where c.Speed > 90 && c.Make == "BMW"  
            select new { c.Make, c.Color };
```

- ▶ Projections can create new (anonymous) data types



# The **orderby** Clause

- ▶ Results can be sorted using the **orderby** clause

```
List<Car> cars = new List<Car> {  
    new Car{ PetName="Henry", Color="Silver", Speed=100, Make="BMW" },  
    ...  
};  
var query = from c in cars  
            where c.Speed >= 55  
            orderby c.PetName  
            select c;
```

- ▶ The order can be **ascending** (the default) or **descending**

```
var query = from c in cars  
            where c.Speed >= 55  
            orderby c.PetName descending, c.Color  
            select c;
```



# The **group** Clause

- ▶ Use the **group** keyword or the **GroupBy()** method
  - Resulting query yields a set of keyed result groups

```
var query = from i in numbers
             group i by i % 2;

foreach ( var group in query )
{
    Console.WriteLine( group.Key );
    foreach ( var i in group )
    {
        Console.WriteLine( "\t" + i );
    }
}
```

- ▶ There is also a more sophisticated **group into** syntax



# The **join** Clause

- ▶ Use the join keyword to join elements on equality

```
var query = from c in customers
            join o in orders on c.Id equals o.CustomerId
            select new
            {
                Name = c.Name,
                Product = o.Product
            };
foreach ( var cop in query )
{
    Console.WriteLine( "{0} bought {1}", cop.Name,
                        cop.Product.Name );
}
```

- ▶ Other variations of join can be expressed in a number of ways...



# The **let** Clause

- ▶ Local expression or queries can be stored in variables for use later in the query

```
string[] sentences = { ... }  
  
var query = from sentence in sentences  
            let words = sentence.Split( ' ' )  
            orderby words.Length  
            select sentence;
```

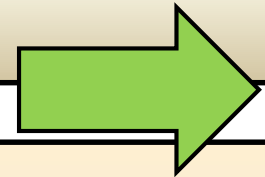
- ▶ Locally introduced variable
  - can be a simple type or a full query
  - is read-only



# Query Operators Resolution

- ▶ These query operators are keywords with syntax highlighting and IntelliSense
- ▶ But they are resolved as extension methods in the **Enumerable** class

```
var query = from g in wiiGames
             where g.Length >= 18
             orderby g.Length, g
             select g.ToUpper();
```



```
var query = wiiGames.Where( g => g.Length >= 18 )
                    .OrderBy( g => g.Length )
                    .ThenBy( g => g )
                    .Select( g => g.ToUpper() );
```

- ▶ You can use either syntax or use delegates instead of anonymous methods etc.

# Agenda

- ▶ Introducing LINQ
- ▶ LINQ Query Keywords
- ▶ **LINQ Query Operator Methods**
- ▶ LINQ to Entities
- ▶ LINQ to XML
- ▶ Expression Trees
- ▶ Lab 3
- ▶ Discussion and Review

# Count<T>

- ▶ You can compute the number of items in the result set with **Count<T>**

```
string[] wiiGames = {  
    "Super Mario Galaxy",  
    "FIFA 09",  
    "Guitar Hero III",  
    "Wii Sports",  
    "Wii Fit",  
    "Legend of Zelda: Twilight Princess"  
};  
var query = from g in wiiGames  
            where g.Length >= 18  
            select g;  
Console.WriteLine( "{0} games match the query", query.Count() );
```

- ▶ This forces an evaluation of the query expression!





# Reverse<T>

- ▶ You can reverse the result sequence with **Reverse<T>**

```
string[] wiiGames = {  
    "Super Mario Galaxy",  
    ...  
};  
  
var query = ( from g in wiiGames select g ).Reverse();
```

- ▶ Note that this does not evaluate the query expression...!



# Set Operations: **Except<T>**

- ▶ Differences between queries can be computed with **Except<T>**

```
string[] wiiGames = {  
    "Super Mario Galaxy", ...  
};  
string[] xbox360Games = {  
    "Halo", ...  
};  
  
var query = ( from g in wiiGames select g ).Except(  
    from g in xbox360Games select g );  
var query2 = wiiGames.Except( xbox360Games );
```

- ▶ Do you think this will evaluate the query expression? ☺
- ▶ **Union<T>**, **Intersect<T>**, and **Except<T>** constitute the set operations (**Distinct<T>** is also helpful!)



# Singleton Operations

- ▶ A single element can be retrieved from a query result

- `First<T>`
- `Last<T>`
- `Single<T>`

```
var query = wiiGames.Intersect( xbox360Games );  
  
var first = query.First();  
var last = query.Last();  
var theOnlyOne = query.Single();  
  
Console.WriteLine( first );  
Console.WriteLine( last );  
Console.WriteLine( theOnlyOne );
```

- ▶ Each of these has an `...OrDefault<T>` version
  - `FirstOrDefault<T>`
  - `LastOrDefault<T>`
  - `SingleOrDefault<T>`



# Partitioning Operators

- ▶ **Take()** and **Skip()**

```
string[] wiiGames = {  
    "Super Mario Galaxy", ...  
};  
string[] xbox360Games = {  
    "Halo", ...  
};  
  
var query1 = wiiGames.Union( xbox360Games ).Take( 7 );  
var query2 = wiiGames.Union( xbox360Games ).Skip( 3 );
```

- ▶ There are also
  - **TakeWhile()**
  - **SkipWhile()**



# Aggregation Operators

- ▶ **Aggregate()** computes a running value

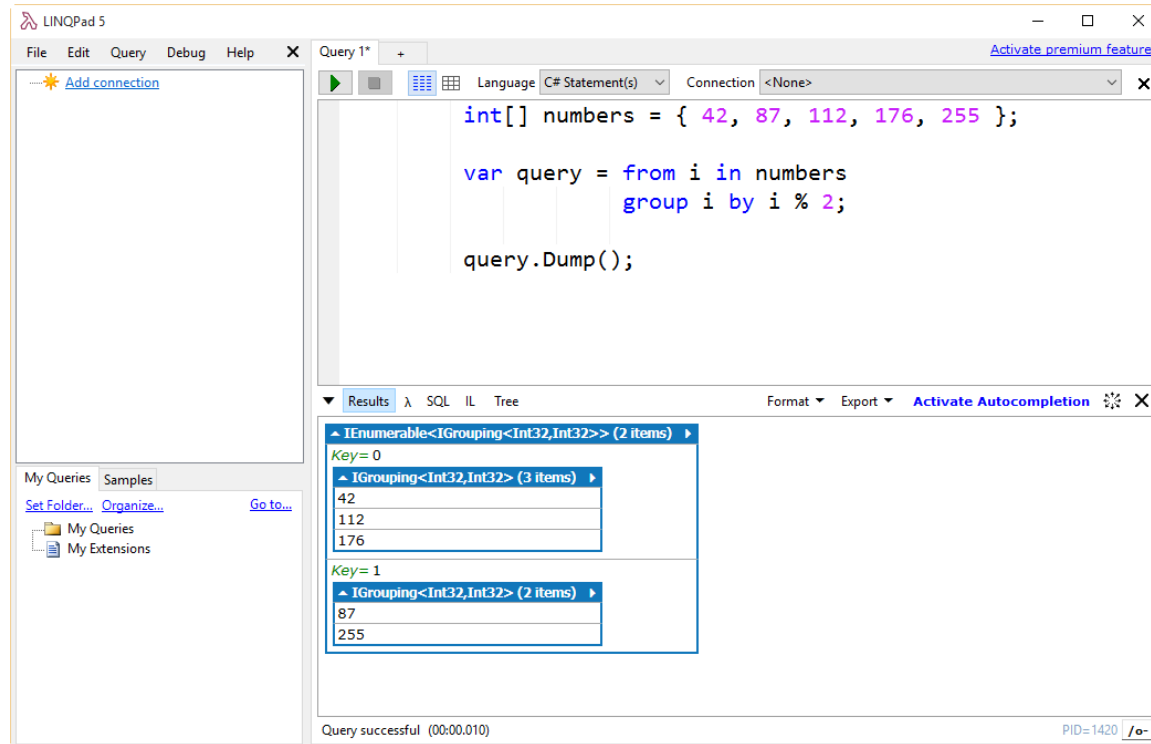
```
int[] numbers = { 42, 87, 112, 176, 255 };  
var result = numbers.Aggregate( 1, ( product, i ) => product * i );  
  
Console.WriteLine( "The product of numbers is " + result );
```

- ▶ Other aggregation operators include
  - Count()
  - Sum()
  - Min()
  - Max()
  - Average()



# LINQPad

- ▶ LINQPad by Joseph Albahari is indispensable!



- ▶ Get it from <http://www.linqpad.net>



# Agenda

- ▶ Introducing LINQ
- ▶ LINQ Query Keywords
- ▶ LINQ Query Operator Methods
- ▶ **LINQ to Entities**
- ▶ LINQ to XML
- ▶ Expression Trees
- ▶ Lab 3
- ▶ Discussion and Review

# ADO.NET Entity Framework

- ▶ The de-facto standard for disconnected data access providing
  - Entity Data Models (EDM)
  - Entity SQL
  - Object Services
- ▶ It supports
  - Writing code against a conceptual model
  - Type-safe data access
  - Robustness and indepedance across storage systems
  - Maintainability
- ▶ Tools and wizards supporting
  - Database-first design
  - Code-first design





# Querying and Updating Data

- ▶ Using LINQ to Entities to query data

```
using( ShopEntities entities = new ShopEntities() )
{
    var query = from c in entities.Customers
                 where c.Orders.Count > 0
                 select c;

    ...
}
```

- ▶ DbContext-generated class
  - keeps tracks of updates
  - saves back to database

```
using( ShopEntities entities = ... )
{
    ...
    entities.SaveChanges();
}
```



# Customizing Classes

- ▶ Never modify the auto-generated classes!!
  - Instead, augment the auto-generated partial classes

```
public partial class Customer
{
    public string FullName
    {
        get
        {
            return FirstName + " " + LastName;
        }
    }
    public int Age
    {
        get { return ...; }
    }
}
```



# Agenda

- ▶ Introducing LINQ
- ▶ LINQ Query Keywords
- ▶ LINQ Query Operator Methods
- ▶ LINQ to Entities
- ▶ **LINQ to XML**
- ▶ Expression Trees
- ▶ Lab 3
- ▶ Discussion and Review

# Introducing LINQ to XML

- ▶ Provides querying facilities over XML documents
  - Introduces a new **XDocument** class set deriving from **Xobject**
  - In **System.Xml.Linq** namespace
- ▶ **XAttribute**
- ▶ **XNode**
  - *XContainer*
    - **XDocument**
    - **XElement**
  - **XComment**
  - **XText**
    - **XCDATA**
- ▶ . . .

# XDocument

- ▶ Provides main access to XML document handling
- ▶ **XDocument.**
  - Load() static
  - Parse() static
  - Save()

```
XDocument doc = XDocument.Load( @"C:\Tmp\Movies.xml" );
```

```
XDocument doc = XDocument.Parse( "<Customers>...</Customers>" );
```

```
doc.Save( @"C:\Tmp\CustomersOrders.xml" );
```



# Querying with LINQ to XML

- ▶ Use LINQ queries over the DOM provided by the **XDocument** hierarchy classes

```
XDocument doc = XDocument.Load( @"C:\Tmp\CustomersOrders.xml" );  
var query = from order in doc.Descendants( "Order" )  
            where order.Attribute( "OrderID" ).Value == "10677"  
            select new  
            {  
                OrderID = (int) order.Attribute( "OrderID" ),  
                CustomerID =  
                    (string) order.Parent.Attribute( "CustomerID" ),  
                Freight = (decimal) order.Attribute( "Freight" )  
            };
```

- ▶ The full power of LINQ is available, e.g. **join**, **group** etc.



# Transforming XML to Objects

- ▶ LINQ to XML is perfect for transforming XML
  - XML -> objects
  - XML -> text
  - XML -> XML

```
List<Customer> customersOrders =  
    ( from c in doc.Descendants( "Customer" )  
      select new Customer  
      {  
          Id = c.Attribute( "CustomerID" ).Value,  
          Name = c.Attribute( "CompanyName" ).Value,  
          Orders = ( from o in c.Elements( "Order" )  
                     select new Order  
                     {  
                         Id = (int) o.Attribute( "OrderID" ),  
                         Freight = (decimal) o.Attribute( "Freight" )  
                     } ).ToList()  
      } ).ToList();
```



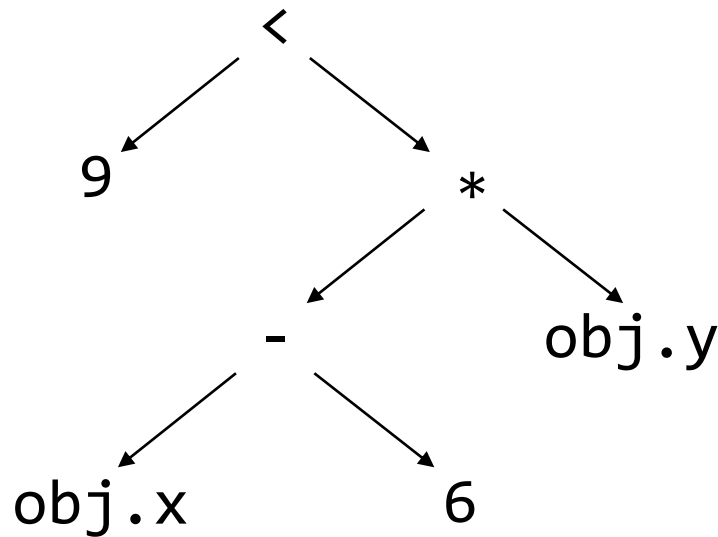
# Agenda

- ▶ Introducing LINQ
- ▶ LINQ Query Keywords
- ▶ LINQ Query Operator Methods
- ▶ LINQ to Entities
- ▶ LINQ to XML
- ▶ **Expression Trees**
- ▶ Lab 3
- ▶ Discussion and Review



# What is an Expression Tree?

- ▶ The expression `9 < ( obj.x - 6 ) * obj.y` is



- ▶ The **Expression** class captures expression trees
  - Each node derive from **Expression**



# Expression Types

## ► Expression

- ConstantExpression
- MemberExpression
- ParameterExpression
- UnaryExpression
- BinaryExpression
- LambdaExpression
  - **Expression<TDelegate>**
- ...

## ► Abstract base class providing static methods

- 15 classes derive from **Expression** with 46 operands



# Compiling Lambda Expression Trees

- ▶ Expression trees can be compiled to the underlying delegate type at runtime!
  - `Expression<TDelegate>.Compile()`

```
Expression<Func<int, int, int>> addition = ( x, y ) => x + y;  
Func<int, int, int> add = addition.Compile();  
Console.WriteLine( add( 5, 7 ) );
```

- ▶ Main purpose is not necessarily the compilation in itself – but to “treat code as data”
- ▶ Perfect tool to construct dynamic LINQ queries...!

```
Expression<Func<object, bool>> predicate = ...;  
var query = data.Where( predicate.Compile() );
```

- ▶ But...



# IQueryable<T>

- ▶ Remote LINQ providers has to be based upon **IQueryable<T>** instead of **IEnumerable<T>**, e.g.
  - Entity Framework
  - LINQ to SQL

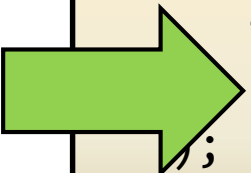
```
public interface IQueryable : IEnumerable
{
    Type ElementType { get; }
    Expression Expression { get; }
    IQueryProvider Provider { get; }
}
```

- ▶ Otherwise data retrieval would be hopelessly inefficient! 😊
- ▶ The actual providers implement **IQueryProvider**
  - Instructs .NET what to actually do when manipulating queries

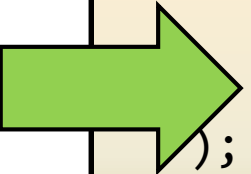


# Queryable Extension Methods

- ▶ **Queryable** static class implements **IQueryable<T>** extension methods!
- ▶ LINQ query methods are essentially doubly implemented



```
public static class Enumerable
{
    public static IEnumerable<T> Where<T>(
        this IEnumerable<T> source,
        Func<T, bool> predicate
    );
    ...
}
```



```
public static class Queryable {
    public static IQueryable<T> Where<T>(
        this IQueryable<T> source,
        Expression<Func<T, bool>> predicate
    );
    ...
}
```



# Lab 3: Creating LINQ Queries

- ▶ Lab 3.1 – 3.6



# Discussion and Review

- ▶ Introducing LINQ
- ▶ LINQ Query Keywords
- ▶ LINQ Query Operator Methods
- ▶ LINQ to Entities
- ▶ LINQ to XML
- ▶ Expression Trees



WINCUBATE

*Jesper Gulmann Henriksen*

PhD, MCT, MCSD, MCPD

Phone : +45 22 12 36 31

Email : [jgh@wincubate.net](mailto:jgh@wincubate.net)

WWW : <http://www.wincubate.net>

Hasselvangelen 243

8355 Solbjerg

Denmark