Advanced C#

LINQ

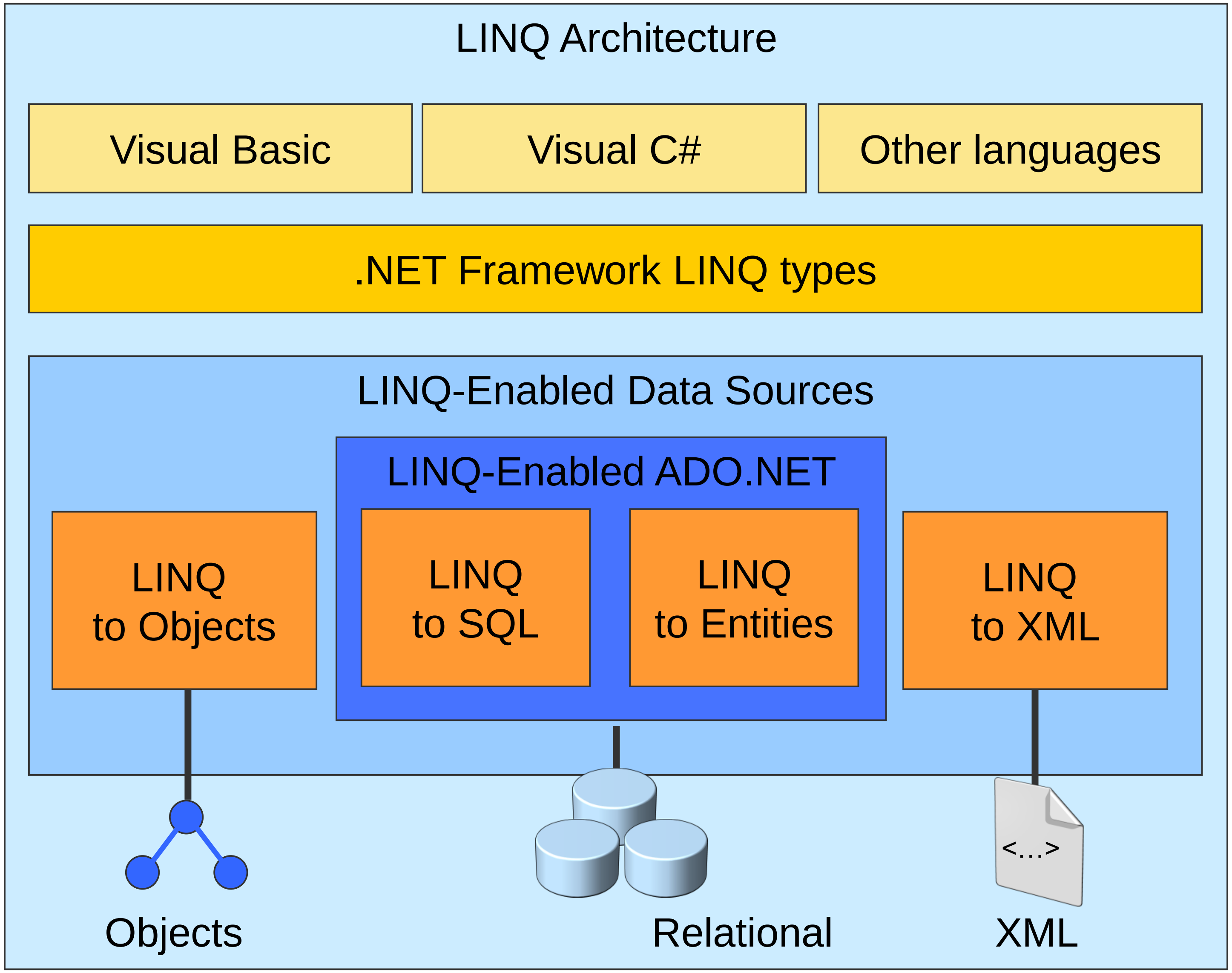
Introduction

- Name
- Company Affiliation
- Title / Function
- Job Responsibility
- Programming Experience
  - C# 1.0, 2.0
- Expectations for the Course

Introduction LINQ

- **L**anguage **IN**tegrated **Q**uery
  - Enables writing queries in C#
  - Like SQL, but strongly typed, Intellisense
- Deferred execution
  - The result of a LINQ query is a query object
- Queries can be executed on:
  - Objects
  - Databases (LINQ to SQL, LINQ to Entities)
  - XML documents (LINQ to XML)
- “Multicore Ready”

Introduction to LINQ



LINQ – Basic Concepts

The class System.Linq.Enumerable contains methods for the IEnumerable<T> interface

```
List<Person> band = new List<Person> {
       new Person{ Name="John"},
       new Person{ Name="Paul"},
       new Person{ Name="George"},
       new Person{ Name="Ringo"}
};

IEnumerable<Person> subSet =
       Enumerable.Where(band, person => person.Name.Length == 4);

IEnumerable<Person> orderedSet =
       Enumerable.OrderBy(band, person => person.Name);

IEnumerable<string> names =
       Enumerable.Select(band, person => person.Name);
```

LINQ – Basic Concepts

These methods are extension methods of IEnumerable<T>

```
List<Person> band = new List<Person> {
        new Person{ Name="John"},
        new Person{ Name="Paul"},
        new Person{ Name="George"},
        new Person{ Name="Ringo"}
};

IEnumerable<string> selectedOrderedNames =
        band.Where(person => person.Name.Length == 4)
        .OrderBy(person => person.Name)
        .Select(person => person.Name);
```

LINQ – Basic Concepts

C# 3.0 introduces the Comprehension Syntax

```
List<Person> band = new List<Person> {
        new Person{ Name="John"},
        new Person{ Name="Paul"},
        new Person{ Name="George"},
        new Person{ Name="Ringo"}
};

IEnumerable<string> selectedOrderedNames =
        from person in band
        where person.Name.Length == 4
        orderby person.Name
        select person.Name;
```

During compilation to IL this syntax is re-written to method calls

LINQ – Basic Concepts

When typing method calls to Enumerable methods the intellisense shows ‘strange’ parameter types:

- Func<TResult >
  - `delegate TResult Func<TResult>()`
  - Used to make a delegate to a parameterless method that returns a value of type TResult
  - Func<T, TResult>
    Func<T1, T2 , TResult >
    Func<T1, T2, T3 , TResult >
    Func<T1, T2, T3, T4 , TResult >

LINQ – Basic Concepts

Deferred execution:

A LINQ expression is evaluated when the resulting collection is enumerated. It is NOT evaluated at the time of declaration.

- Be aware of performance costs
- Be aware of changes to the collection(s) and outer variables involved

LINQ – Basic Concepts

Deferred execution

```
IEnumerable<string> selectedOrderedNames =
        from person in band
        where person.Name.Length > 4
        orderby person.Name
        select person.Name;

band.Add(
        new Person { Name = "Stuart", Address = "Hamburg" }
);

foreach (string name in selectedOrderedNames)
{
        Console.WriteLine(name);
}
```

LINQ – Basic Concepts

Deferred execution and outer variables

```
List<Person> band = new List<Person> {
        new Person{ Name="John"},
        new Person{ Name="Paul"},
        new Person{ Name="George"},
        new Person{ Name="Ringo"}
};

string filter = "e";

var selection = band.Where(person=>person.Name.Contains(filter));

filter = "o";

selection =
        selection.Where(person => person.Name.Contains(filter));

Console.WriteLine("Found {0} persons.", selection.Count());
```

LINQ – Basic Concepts

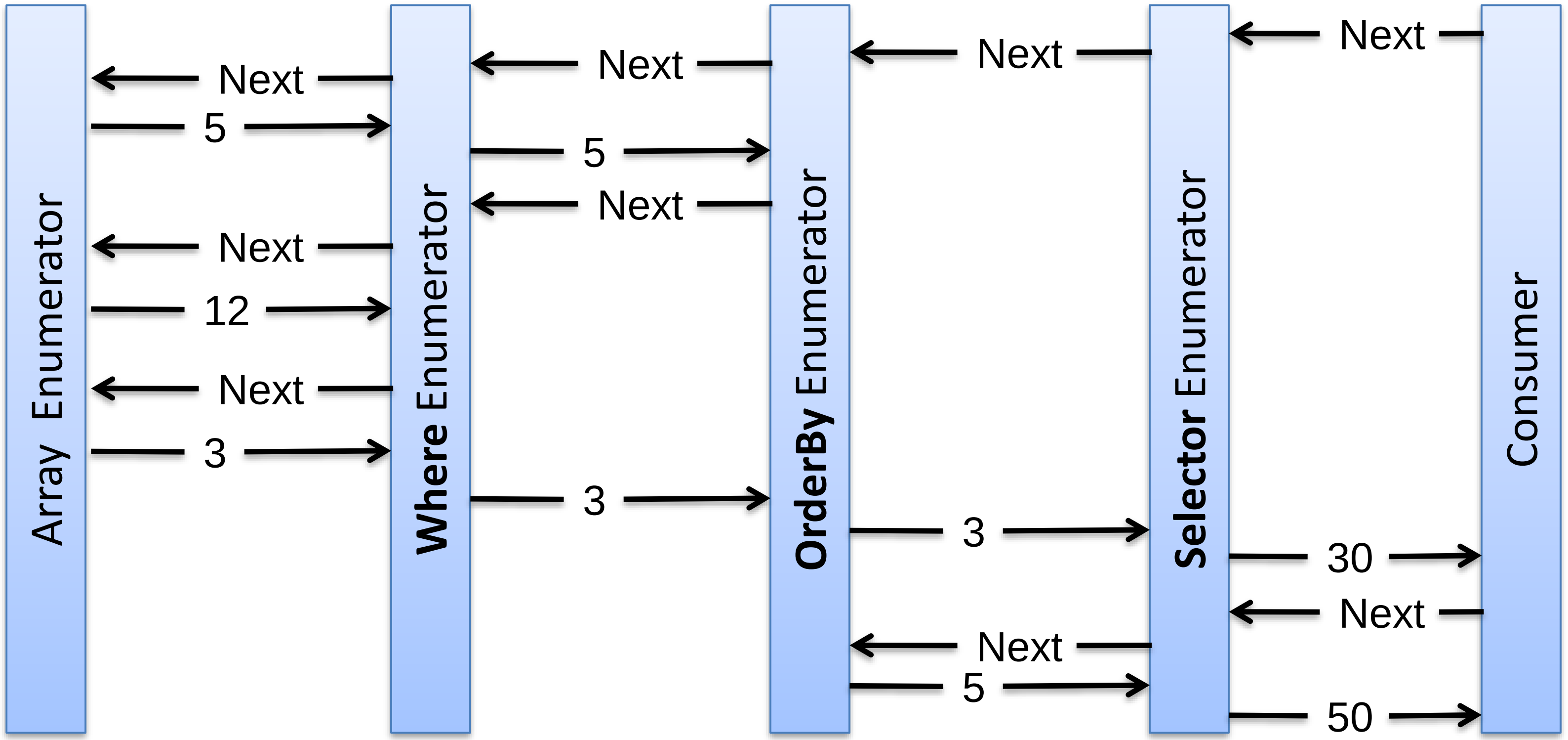
Execution is forced by using

- ToList()
  ToArray()
  ToDictionary()
  ToLookup()
- First()
  Count()
  Average()
  etc.

LINQ – Basic Concepts

Execution

```
IEnumerable<int> lessThanTen = new int[] { 5, 12, 3 }
        .Where(n => n < 10)
        .OrderBy(n => n)
        .Select( n => n * 10);
```



LINQ - Syntax

Query Expression =

from-clause

$$\boxed{\begin{array}{l} \textit{from} \quad\quad\quad 0..* \\ \textit{let} \\ \textit{where} \\ \textit{join} \\ \textit{orderby} \end{array}}$$

select-clause | group-clause

LINQ – Syntax

The *from-clause* and *select-clause*(1)

```
IEnumerable<string> addresses =
                       from person in band
                       select person.Address;
```

– The first *from-clause* creates a range variable which ranges over a sequence

LINQ – Syntax

The *from-clause* and *select-clause*(2)

```
var info =
       from person in band
       select new {Initial=person.Name[0], person.Address};

foreach (var item in info)
{
       Console.WriteLine(
              "{0}, {1}", item.Initial, item.Address);
}
```

- The *select-clause* specifies **one** object only
- Create an anonymous type if no known type is applicable

LINQ – Syntax

The *from-clause* and *group-clause*

```
IEnumerable<IGrouping<int, Person>> groups =
       from person in band
       group person by person.Name.Length;

foreach (IGrouping<int, Person> group in groups)
{
       Console.WriteLine(
              "Group: {0} characters in Name", group.Key);
       foreach (Person person in group)
       {
              Console.WriteLine(person.Name);
       }
       Console.WriteLine();
}
```

LINQ – Syntax

The optional *from-clause*

```
List<Person> beatles = new List<Person>{
       new Person{Name="Paul", Address="Liverpool", Instruments =
              new List<string>{"Bass", "Guitar", "Vocals"}},
       new Person{Name="John", Address="New York", Instruments =
              new List<string>{"Guitar", "Piano", "Vocals"}},
       new Person{Name="George", Address="Liverpool", Instruments =
              new List<string>{"Guitar", "Vocals"}},
       new Person{Name="Ringo", Address="Los Angeles", Instruments
              = new List<string>{"Drums", "Vocals" }}};
var singersAndGuitarPlayers =
       from person in beatles
       from instrument in person.Instruments
       where instrument == "Guitar" || instrument == "Vocals"
       select person.Name;
```

LINQ – Syntax

The optional *let-clauses*(1) (efficiency)

```
// selecting names and number of instruments of those beatle
// members that play more than 2 instruments
var collection =
       from person in beatles
       let numberOfInstruments = person.Instruments.Count
       where numberOfInstruments > 2
       select new { person.Name,
                    NumberOfInstruments = numberOfInstruments };
```

LINQ – Syntax

The optional *let-clauses(2)* (readability)

```
// selecting names and stringed instruments beatle members play
var collection =
       from person in beatles
       let stringInstruments =
              from instrument in person.Instruments
              where  instrument == "Guitar" || instrument == "Bass"
                     || instrument == "Piano"
              select instrument
       select new { person.Name, stringInstruments};
```

LINQ – Syntax

into and **where** work like **having** in SQL

```
// selecting groups of band members that live in the same city
var groups2 =
       from person in band
       group person by person.Address into citizens
       where citizens.Count() > 1
       select citizens;
```

LINQ – Syntax

The optional *join-clauses*(1) (inner join)

```
var fellowCitizens =
    from beatle in beatles
    join rollingStone in rollingStones
       on beatle.Address equals rollingStone.Address
    select new { Beatle = beatle, RollingStone = rollingStone };

foreach (var pair in fellowCitizens)
{
       Console.WriteLine("{0} & {1}",
              pair.Beatle.Name, pair.RollingStone.Name);
}
```

LINQ – Syntax

The optional *join-clauses*(2) (left join)

```
var fellowCitizens =

    from beatle in beatles
    join rollingStone in rollingStones
       on beatle.Address equals rollingStone.Address into stoneList
    from stone in stoneList.DefaultIfEmpty()

    select new { Beatle = beatle, RollingStone = stone };


foreach (var pair in fellowCitizens)

{

    Console.WriteLine("{0} & {1}",
        pair.Beatle.Name,
        pair.RollingStone == null ? "-" : pair.RollingStone.Name);

}
```

LINQ – Syntax

The optional *orderby-clauses*(1)

```
var persons =
       from beatle in beatles
       orderby beatle.Address descending, beatle.Name
       select beatle;

foreach (var person in persons)
{
       Console.WriteLine("{0}, {1}", person.Address, person.Name);
}
```

LINQ – Syntax

The optional *orderby-clauses*(2)

```
var persons =
       (from beatle in beatles select beatle)
       .OrderBy(
              beatle=>beatle.Address,
              StringComparer.InvariantCultureIgnoreCase)
       .ThenBy(beatle=>beatle.Name);


foreach (var person in persons)
{
       Console.WriteLine("{0}, {1}", person.Address, person.Name);
}
```

LINQ – Syntax

Important extension methods *without* counterpart in comprehension syntax:

Set Operations	• Union, Concat • Intersect, Except
Aggregate Functions	• Min, Max, All, Any • Sum, Count, Average
Selection Functions	• First, Single • Take, Skip
Type Operations	• OfType<T> • Cast<T>

LINQ - Optimizing

Memory

– LINQ expressions can generate a lot of objects during execution

Performance

– LINQ expressions can be written and executed in different ways resulting in faster or slower execution

LINQ - Optimizing

- Use the CLR Profiler to determine the memory usage of a query
  - http://msdn.microsoft.com/en-us/library/ms979205.aspx
  - On x64 systems: regsvr32 ProfilerOBJ.dll (run as Admin)
- Use the System.Diagnostics.Stopwatch class to measure the query execution time
  - or use the profiling options of Visual Studio
- Use Reflector to see the translation from Comprehensive Syntax to method calls
  - http://reflector.red-gate.com/download.aspx

Questions & Answers

General

- Labs