Queries with LINQ

Syntax and Possibilities

Overview

- **Anatomy Of A Query**
  - Sequences
  - Deferred execution
  - Selecting, Filtering, Grouping, Joins
- **Advanced Queries**
  - Composition
  - Dynamic queries

Anatomy Of a Query

1. A sequence (can be local or remote)
2. A *range variable* (to range over the sequence)
   - Can appear in later clauses
3. Query operators
   - Emits another local sequence (think of a pipeline)

- This style of query formally known as *comprehension query*

```
var employees = new EmployeeRepository().GetAll();
int maxNameLength = 5;

var employeesWithShortNames =
    from employee in employees
    where employee.Name.Length <= maxNameLength
    select employee;
```

Comprehension Query versus Lamba Query

- Lambda queries
  - Generally offer more control and flexibility
  - *Chaining* of query operators looks like a pipeline
  - *Select* operator is optional (when not doing a projection)
  - Many query operators have no comprehension query equivalent
- Neither query syntax will modify the original sequence
- It is possible to mix syntaxes

```
var employeesWithShortNames =
    employees.Where(e => e.Name.Length <= maxNameLength)
             .Select(e => e);
```

Deferred Execution

- **How to implement a custom Where operator?**
  - Use an extension method or instance method

```
public static class EmployeeExtensions
{
    public static IEnumerable<Employee> Where(
         this IEnumerable<Employee> sequence, Predicate<Employee> predicate)
    {
        List<Employee> list = new List<Employee>();
        foreach(Employee employee in sequence)
        {
            if (predicate(employee))
                list.Add(employee);
        }
        return list;
    }
}
```

Wrong!

Deferred Execution via C# Iterators

```
public static class EmployeeExtensions
{
    public static IEnumerable<Employee> Where(
          this IEnumerable<Employee> sequence, Predicate<Employee> predicate)
    {
        foreach (Employee employee in sequence)
        {
            if (predicate(employee))
            {
                yield return employee;
            }
        }
    }
 }
```

Right!

Deferred Execution “Surprises”

- **Query re-evaluated on each iteration**
  - Can see different results from same query if data changed
- **Greedy operators**
  - Some are obviously greedy, others are not so obvious…
- **Streaming operators versus non-streaming**
  - Once execution begins - non-streaming operators consume everything

```csharp
var sortedEmployees =
     from employee in employees
     where employee.Name.Length <= 4
     orderby employee.Name ascending
     select employee;
```

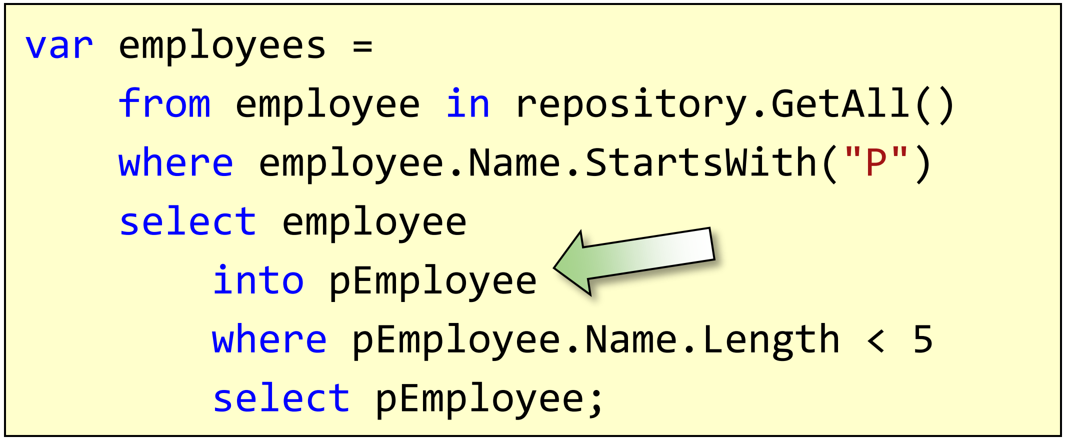
The `let` keyword

- **Projects a new range variable**
  - Allows for expression re-use

```
var employees =
    from employee in repository.GetAll()
        let lowercaseName = employee.Name.ToLower()
    where lowercaseName.StartsWith(lowercaseName.Substring(
                                       lowercaseName.Length - 1))
    select employee;
```

Using `into`

- **The `into` keyword is to continue a query after a projection.**
  - Original range variable goes out of scope



```
var employees =
    from employee in repository.GetAll()
    where employee.Name.StartsWith("P")
    select employee
        into pEmployee
        where pEmployee.Name.Length < 5
        select pEmployee;
```

- **Common use of `into` is with grouping…**

Grouping

- **Transforms sequence into a sequence of groups**
  - A group implements IGrouping<Tkey, T>
  - The group operator will end a query, use `into` to continue the query

```
var groupedEmployees =
    from employee in repository.GetAll()
    group employee by employee.Name[0] into letterGroup
    orderby letterGroup.Key ascending
    select letterGroup;
```

```
foreach (var group in groupedEmployees)
{
    Console.WriteLine(group.Key);
    foreach (var employee in group)
    {
        Console.WriteLine("\t{0}", employee.Name);
    }
}
```

Grouping with a Composite Key

- **Create an object to serve as key (named or anonymous type)**

```
var groupedEmployees =
    from employee in repository.GetAll()
    group employee by new { employee.DepartmentID,
                            FirstLetter = employee.Name[0] };
```

```
foreach(var group in groupedEmployees)
{
    Console.WriteLine("\t{0} - {1}",
          group.Key.DepartmentID,
          group.Key.FirstLetter);
    foreach (var employee in group)
    {
        Console.WriteLine(employee.Name);
    }
}
```

Grouping and Projecting

```
var groupedEmployees =
    from employee in repository.GetAll()
    group employee
        by new  { employee.DepartmentID,
                  FirstLetter = employee.Name[0] }
        into gEmployee
        where gEmployee.Count() > 1
        select new {
                    DepartmentID = gEmployee.Key.DepartmentID,
                    FirstLetter = gEmployee.Key.FirstLetter,
                    Count = gEmployee.Count()
                   };
```

Grouping and Projection – Lambda Style

- We can do anything query expressions can do (and more)
- Preference will depend upon individual sense of aesthetics

```
var groupedEmployees =
    repository.GetAll()
        .GroupBy(e => new { e.DepartmentID,
                            FirstLetter = e.Name[0]})
        .Where(g => g.Count() > 1)
        .Select(g => new {
                            g.Key.DepartmentID,
                            g.Key.FirstLetter,
                            Count = g.Count()
                        });
```

Nested Queries

- **Nested queries are used in scenarios similar to nested SELECT commands in T-SQL.**
  - Beware of performance!
  - How many times does the inner query execute for IEnumerable<T>?

```
var engineeringEmployees =
    from employee in employeeRepository.GetAll()
    where employee.DepartmentID ==
                (from department in departmentRepository.GetAll()
                 where department.Name == "Engineering"
                 select department).First().ID
    select employee;
```

Correlated Subqueries

- **Outer range variable appears inside the nested query**
  - Performance issues still possible
- **In the following code, a join is preferable**

```
var employees =
        from employee in employeeRepository.GetAll()
        select new
        {
            Name = employee.Name,
            Department = (from department in departmentRepository.GetAll()
                          where department.ID == employee.DepartmentID
                          select department).First().Name
        };
```

Joins

- **Connects an outer and inner sequence**
  - Uses *equals* keyword, not an == expression
- **Inner sequence loaded into keyed collection**
  - Much faster than a subquery for in-memory sequence
- **Equivalent to INNER JOIN in SQL**
  - Only returns the intersection of two sequences
  - Produces a flat sequence

```
var employees =
        from employee in employeeRepository.GetAll()
        join department in departmentRepository.GetAll()
           on employee.DepartmentID equals department.ID
        select new { employee.Name, Department = department.Name };
```

Group Joins

- Occurs when `into` appears after a `join`
- Outputs groups of sequences and preserves hierarchy

```
var employeesByDepartment =
        from department in departmentRepository.GetAll()
        join employee in employeeRepository.GetAll()
           on department.ID equals employee.DepartmentID
           into eg
        select new { Name = department.Name, Employees = eg };
```

```
foreach (var department in employeesByDepartment)
{
    Console.WriteLine(department.Name);
    foreach (var employee in department.Employees)
    {
        Console.WriteLine("\t{0}", employee.Name);
    }
}
```

Join Hints

- **Can use many join keywords in same query**
  - Join Customers to Orders to OrderItems
- **Possible to join with composite keys**
  - Same strategy as composite grouping – need to construct a key object
- **What about a SQL "LEFT JOIN"**
  - A group join IS a left join
  - A join is a inner join
- **Cross join (for completeness)**

```
var query =
        from employee in employeeRepository.GetAll()
        from department in departmentRepository.GetAll()
        select new { EName = employee.Name, DName = department.Name };
```

Sorting

- We can use multiple expressions after the orderby keyword
- Default sort is ascending

```
var employees =
    from employee in employeeRepository.GetAll()
    orderby employee.DepartmentID ascending,
            employee.Name descending
    select employee;
```

```
var employees = employeeRepository.GetAll()
                    .OrderBy(e => e.ID)
                    .ThenByDescending(e => e.Name);
```

Composition

- **LINQ’s deferred execution allows us to compose queries**
  - Instead of passing data from lower tiers, pass the “query”

```
public IEnumerable<Employee> GetByDepartmentID(int departmentID)
{
    return
        from employee in _employees
        where employee.DepartmentID == departmentID
        select employee;
}
```

```
// this screen sorts by Name
var employees = new EmployeeRepository().GetByDepartmentID(1);

var sorted =
    from employee in employees
    orderby employee.Name ascending
    select employee;
```

Still not executed

Conditional Composition

```
var employees = new EmployeeRepository().GetByDepartmentID(1);

if (sortByName)
{
    employees = employees.OrderBy(e => e.Name);
}
else
{
    employees = employees.OrderBy(e => e.ID);
}

DoDataBinding(employees);
```

Dynamic Queries

- **Static typing of LINQ is both a blessing and a curse**
- **What if we need an expression that can’t be formulated at compile time?**
- **Remember, the C# compiler can translate lambda expression into expression trees**
  - We must write our expression trees the hard way
  - Expression class provides factory methods to help

```
employees = employees.OrderBy( ? );
```

Dynamic OrderBy

- **Build our own expression**
  - Easy to run into typing issues

```
// still static
Expression<Func<Employee, string>> orderExpression = e => e.Name;
employees = employees.OrderBy(orderExpression.Compile());
```

```
string field = "Name";

var parameter =
    Expression.Parameter(typeof(Employee), "e");
var getter =
    Expression.Property(parameter, typeof(Employee).GetProperty(field));
var lambda =
    Expression.Lambda<Func<Employee, string>>(getter, parameter);
employees = employees.OrderBy(lambda.Compile());
```

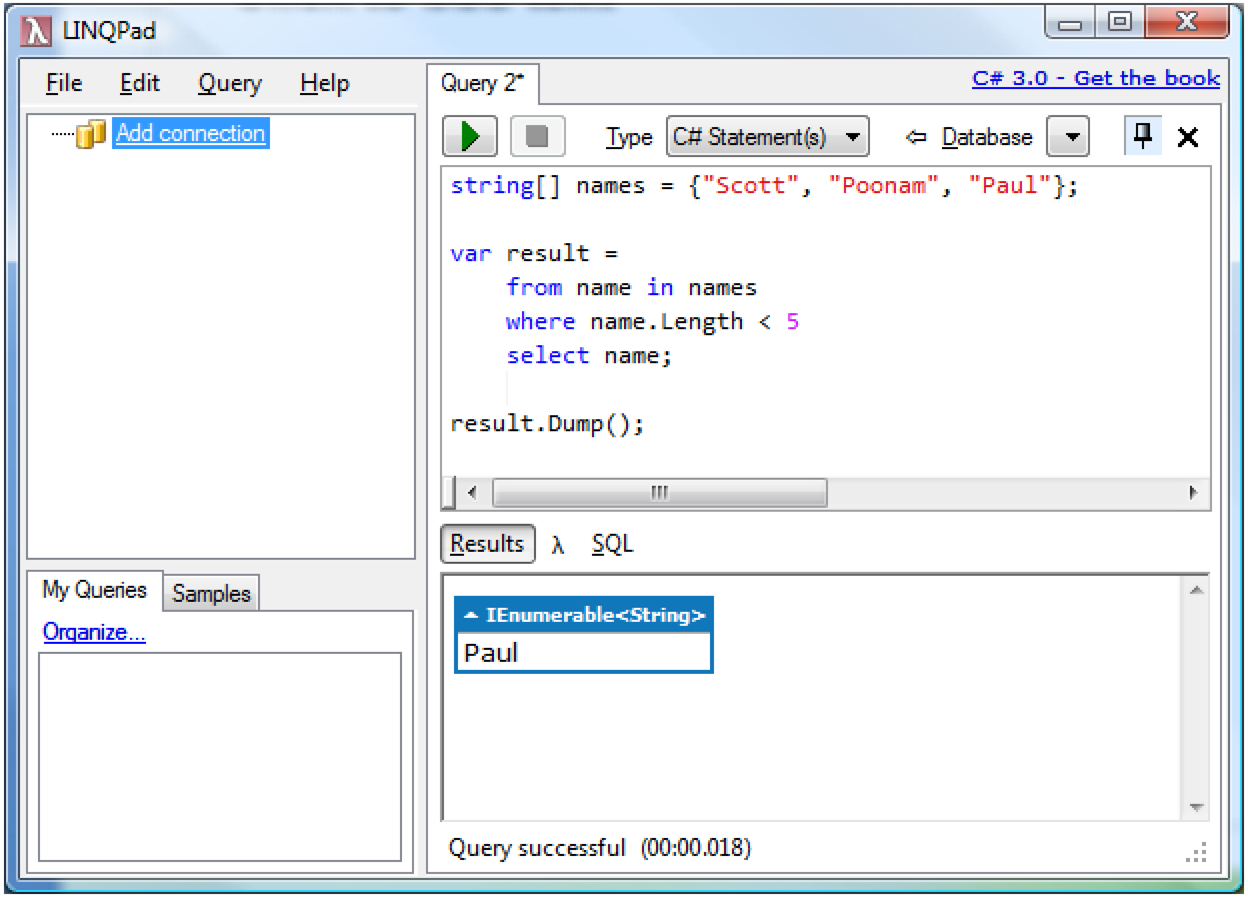
LINQ Dynamic Query Library

- **Dynamic query library is part of VS2008 samples**
  - □ A set of extension methods in the System.Linq.Dynamic namespace

```
var employees = new EmployeeRepository().GetAll();

employees = employees.AsQueryable()
                     .Where("DepartmentID = 1")
                     .OrderBy(“Name");
```

LINQPad



```
string[] names = {"Scott", "Poonam", "Paul"};

var result =
    from name in names
    where name.Length < 5
    select name;

result.Dump();
```

```
IEnumerable<String>
Paul
```

Query successful (00:00.018)

Summary

- **Query syntax and lambda syntax achieve the same goal**
- **Deferred execution is powerful**
  - Compose queries
  - Be aware of its presence
- **LINQ allows joining, grouping, ordering**
  - More operations in the next module

References

- VS 2008 Samples (Dynamic Query Library) http://msdn2.microsoft.com/en-us/vcsharp/bb894665.aspx
- http://weblogs.asp.net/scottgu/archive/2008/01/07/dynamic-linq-part-1-using-the-linq-dynamic-query-library.aspx
- http://www.linqpad.net/