LINQ and Lambdas

transforming performance
through learning

Contents

- **Objectives**
  - Learn about and use LINQ and Lambda expressions
- **Contents**
  - The role of LINQ in .NET
  - Query syntax – Basic queries, Aggregation
  - Create and use Lambda expressions
- **Hands on Labs**
  - Using LINQ and the supporting language features

What is Language Integrated Query ?

- **Extend .NET languages to add native data querying capability**
- **It has been compared to SQL language**

  - SQL is an extremely rich language for databases
  - LINQ is infinitely extensible for any dataset

- **From a developer's point of view**
  - It is relatively easy to code
  - In one line it does what was traditionally done in 20!
  - Your code has to be minutely tested

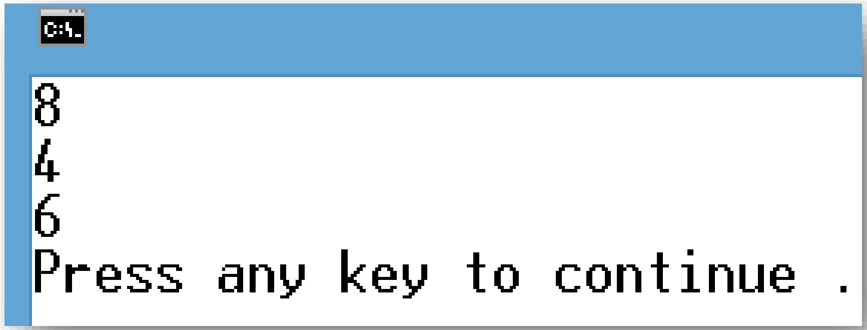
- **Ok, let's see a few examples and then look deeper into LINQ**

Simple query

```
int[] numbers = { 1, 9, 2, 8, 3, 7, 4, 6, 5 };
var query = from num in numbers
            where num > 3 && num % 2 == 0
            select num;

foreach (int n in query) {
    Console.WriteLine(n);
}
```

Not using LINQ

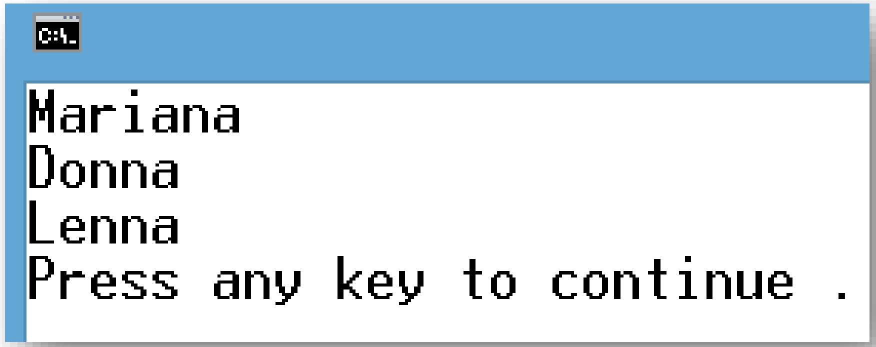
```
int[] numbers = { 1,9,2,8,3,7,4,6,5 };
List<int> query = new List<int>();
foreach (int num in numbers) {
    if (num > 3 && num % 2 == 0)
                query.Add(num);
}
// then a foreach to print ‘query’
```



```
8
4
6
Press any key to continue .
```

A more complex filter

```
List<string> names = new List<string> {
               "Azzie", "Mariana", "Nancie", "Bob",
               "Anna", "Freddie", "Donna", "Lenna", "David" };


var query = from name in names
            where name.Length > 4 && name.EndsWith("na")
            select name;


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



```
Mariana
Donna
Lenna
Press any key to continue .
```

What is the var keyword?

- var tells the complier to infer the actual type
- Does not equate to Object (C#) or Variant (VBA) type
- Makes life of a developer a bit easier!
  - Compare these code samples:

```
int[] numbers = { 1, 9, 2, 8, 3, 7, 4, 6, 5 };

IEnumerable<int> query = from num in numbers
                                  where num > 3 && num % 2 == 0
                               select num;
```

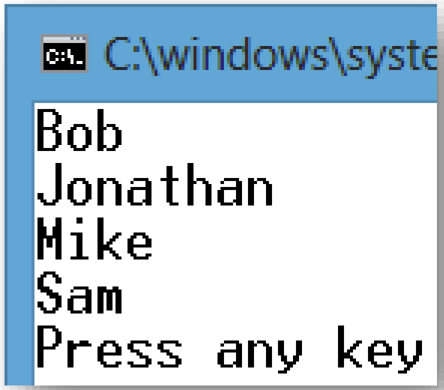
  - Or

```
int[] numbers = { 1, 9, 2, 8, 3, 7, 4, 6, 5 };

var query = from num in numbers
                 where num > 3 && num % 2 == 0
                 select num;
```

- **IEnumerable<T>** is the base interface of collections that can be enumerated
  - i.e. you can use “foreach” on it

Ordering the result – orderby

```
List<Student> students = new List<Student> {
        new Student() {Age=18,Name="Sam"},
        new Student() {Age=20,Name="Mike"},
        new Student() {Age=19,Name="Bob"},
        new Student() {Age=21,Name="Jonathan"}
};

var query = from stu in students
                where stu.Age > 17
                orderby stu.Name ascending
                select stu.Name;


foreach (string name in query) {

    Console.WriteLine(name);

}
```


C:\windows\syste

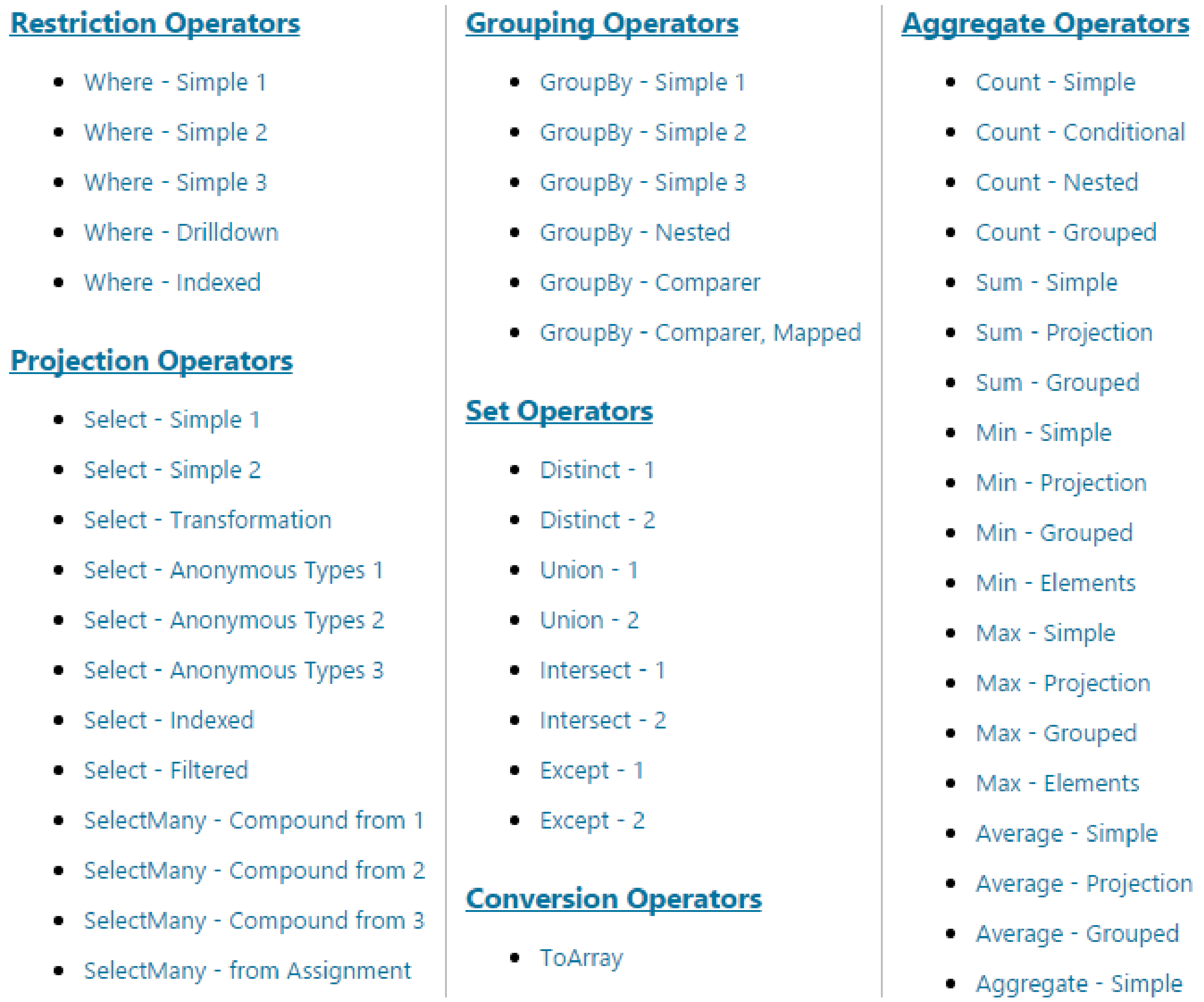
```
Bob
Jonathan
Mike
Sam
Press any key
```

Returning selected attributes - using anonymous type

```
List<Student> students = new List<Student> {
    new Student() { Age = 18, Name = "Sam", Subject = "C#" },
    new Student() { Age = 20, Name = "Mike", Subject = "Java" },
    new Student() { Age = 19, Name = "Bob", Subject = "Networking" },
    new Student() { Age = 21, Name = "Jonathan", Subject = "Marketing"
} };
var query = from stu in students
               where  stu.Age > 18
               select new { stu.Name, stu.Subject };


foreach (var stu in query) {
    Console.WriteLine($"Name:{stu.Name}, Subject:{stu.Subject}");
}
```

Need to learn more?

- **Google “101 Linq Samples” for all the LINQ examples**



Restriction Operators

- Where - Simple 1
- Where - Simple 2
- Where - Simple 3
- Where - Drilldown
- Where - Indexed

Projection Operators

- Select - Simple 1
- Select - Simple 2
- Select - Transformation
- Select - Anonymous Types 1
- Select - Anonymous Types 2
- Select - Anonymous Types 3
- Select - Indexed
- Select - Filtered
- SelectMany - Compound from 1
- SelectMany - Compound from 2
- SelectMany - Compound from 3
- SelectMany - from Assignment

Grouping Operators

- GroupBy - Simple 1
- GroupBy - Simple 2
- GroupBy - Simple 3
- GroupBy - Nested
- GroupBy - Comparer
- GroupBy - Comparer, Mapped

Set Operators

- Distinct - 1
- Distinct - 2
- Union - 1
- Union - 2
- Intersect - 1
- Intersect - 2
- Except - 1
- Except - 2

Conversion Operators

- ToArray

Aggregate Operators

- Count - Simple
- Count - Conditional
- Count - Nested
- Count - Grouped
- Sum - Simple
- Sum - Projection
- Sum - Grouped
- Min - Simple
- Min - Projection
- Min - Grouped
- Min - Elements
- Max - Simple
- Max - Projection
- Max - Grouped
- Max - Elements
- Average - Simple
- Average - Projection
- Average - Grouped
- Aggregate - Simple

Let’s use a couple of those keywords

```
List<string> names = new List<string> {

    "Azzie", "Mariana", "Nancie", "Bob", "Anna", "Freddie", "Donna",
        "Lenna", "David","Azzie", "Mariana", "Nancie", "Bob",

};

var query =  from name in names
             where name.Length > 4
             select name;

Console.WriteLine( query.Distinct().Count() );
```

```
int[] numbers = { 1, 22, 2, 88, 55, 44, 33, 99, 22, 55 };

Console.WriteLine( numbers.Distinct().Average() );
```

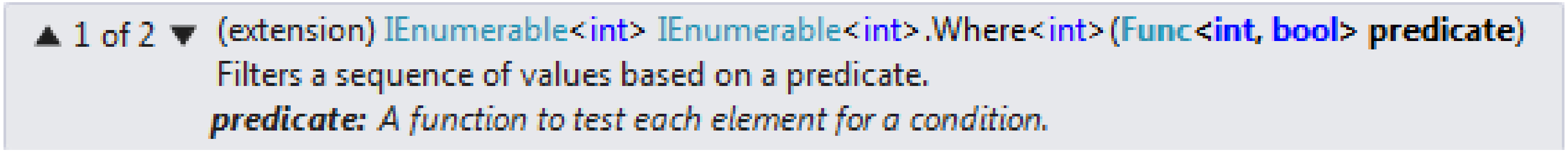
Lambda Expressions

Using a Func<> - Part 1/2

- **Slightly shorter way**

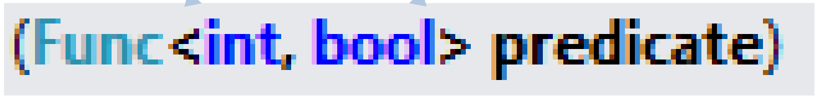
```
int[] numbers = { 1, 1, 2, 3, 5, 8, 13, 21, 34 };
```

```
var result = numbers.Where()
```



```
▲ 1 of 2 ▼ (extension) IEnumerable<int> IEnumerable<int>.Where<int>(Func<int, bool> predicate)
           Filters a sequence of values based on a predicate.
           predicate: A function to test each element for a condition.
```

- **The method asks for a function name that returns a bool and has a parameter of type int.**



```
(Func<int, bool> predicate)
```

Using a Func<> - Part 2/2

- **Lets create the function**

```
int[] numbers = { 1, 1, 2, 3, 5, 8, 13, 21, 34 };
var query = numbers.Where(IsOdd);
```

```
static bool IsOdd(int n)
{
    return (n % 2 == 1);
}
```

The system will call **IsOdd()** passing each `int` in *numbers* and only returns those which are odd (return is ***true***)

Lambda – Towards a more compact code

- **Even simpler, use a lambda expression**

```
int[] numbers = { 1, 1, 2, 3, 5, 8, 13, 21, 34 };
    var query= numbers.Where(n => n % 2 == 1);
```

```
static bool IsOdd(int n)
{
    return (n % 2 == 1);
}
```

Not needed

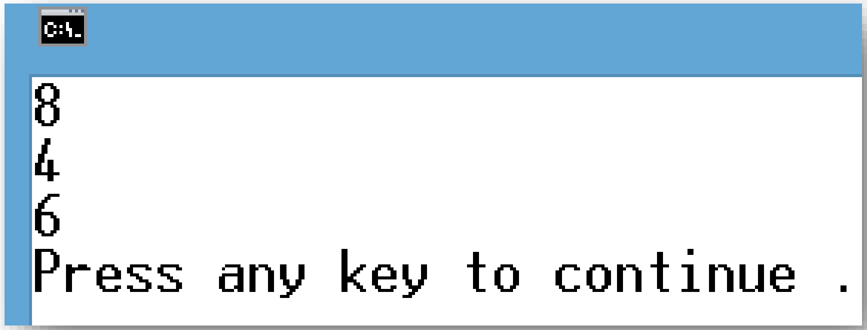
The system will pass each `int` in *numbers* into the (n % 2 == 1) **code** and only returns those which return **`true`**

`Let's learn by example`

Simple query the Lambda way

```
int[] numbers = { 1, 9, 2, 8, 3, 7, 4, 6, 5 };

var query = from num in numbers
             where num > 3 && num % 2 == 0
             select num;

foreach (int n in query) {

    Console.WriteLine(n);

}
```

```
var query =
           numbers.Where(num => num > 3 && num % 2 == 0);
// then the foreach loop to display them
```



```
8
4
6
Press any key to continue .
```

A more complex filter

```
List<string> names = new List<string> {
"Azzie", "Mariana", "Nancie", "Bob",
"Anna", "Freddie", "Donna", "Lenna", "David" };

var query = from name in names
            where name.Length > 4 && name.EndsWith("na")
            select name;
```

```
var query =
       names.Where(name=> name.Length > 4 && name.EndsWith("na"));
```

Ordering the result – orderby

```
List<Student> students = new List<Student> {
        new Student() {Age=18,Name="Sam"},
        new Student() {Age=20,Name="Mike"},
        new Student() {Age=19,Name="Bob"},
        new Student(){Age=21,Name="Jonathan"}
};

var query = from stu in students

            where stu.Age > 17

            orderby stu.Name ascending

            select stu.Name;
```

```
var query =

      students.Where(stu => stu.Age > 17).OrderBy(stu => stu.Name);
```

LINQ – Aggregation Operators

- General purpose `Count` operator (useful with Grouping)
- 4 common numerical aggregation methods
  - `Sum, Min, Max, Average`

```
int[] numbers = { 3, 5, 7, 9 };
string[] names = {"Paul", "Steve", "Peter", "Laurence"};


int totalOfNumbers = numbers.Sum();
int totalLengthOfNames = names.Sum(s => s.Length);
```

List<> FindAll method

See if you can figure out what these statements do:

```
List<int> numbers = new List<int>{ 1, 1, 2, 3, 5, 8};

List<int> oddOnes = numbers.FindAll(n => n % 2 == 1);
```

```
double avg1 = numbers.FindAll(n => n % 2 == 1).Average();

double avg2 =
    numbers.FindAll(n => n % 2 == 1).Distinct().Average();
```

Returning selected attributes - using anonymous type

```
List<Student> students = new List<Student> {

    new Student() { Age = 18, Name = "Sam", Subject = "C#" },

    new Student() { Age = 20, Name = "Mike", Subject = "Java" },

    new Student() { Age = 19, Name = "Bob", Subject = "Networking" },

    new Student() { Age = 21, Name = "Jonathan", Subject = "Marketing" }

};


var query = students.Select( stu => new { stu.Name, stu.Subject });


foreach (var stu in query)
{
    Console.WriteLine($"Name:{stu.Name}, Subject:{stu.Subject}");
}
```

Review

- In this chapter you learned how to use Lambda operations

Lab – LINQ and Lambda

- **Please see your Lab Guide**
  - Lambda