Introduction to LINQ

Introduction

- LINQ stands for **L**anguage **IN**tegrated **Q**uery.
- LINQ enables us to query any type of data source:
  - Databases
  - XML documents
  - Collections in memory etc.

- Another benefit of using LINQ is that it provides IntelliSense and compile-time error checking.

Introduction

- LINQ's syntax is very similar to SQL.
- Consider this SQL query:

```
SELECT * from Books WHERE QuantityInStock > 50;
```

- It can be written in LINQ like this:

```
var result = from book in Books
             where book.QuantityInStock > 50
             select book;
```

Introduction

- You can declare a local variable and let the compiler infer the variable's type based on the variable's initializer.
- The `var` keyword is used in place of the variable's type when declaring the variable.
- A LINQ query begins with a `from` clause, which specifies a range variable (`book`) and the data source to query (`Books`).
- The range variable represents each item in the data source, much like the control variable in a `foreach` statement.

- If the condition in the `where` clause evaluates to `true`, the element is selected.
- The `select` clause determines what values appear in the `result`.

Querying an Array Using LINQ

- Let's consider we have an array.

```
int[] intArray = { 2, 9, 6, 0, 7, 1, 4, 8, 5 };
```

- Fetch all the values greater than 4 using LINQ:

```
var filtered = from value in intArray
               where value > 4
               select value;
```

Output:

```
Array values greater than 4:
9 6 7 8 5
```

orderby Clause

- The `orderby` clause sorts the query results in ascending order.
- The `descending` modifier in the `orderby` clause sorts the results in descending order.

```
var sorted = from value in intArray
             orderby value
             select value;
```

```
Output:

Original array sorted:
0 1 2 4 5 6 7 8 9
```

```
var sorted = from value in intArray
             orderby value descending
             select value;
```

```
Output:

Original array sorted, descending:
9 8 7 6 5 4 2 1 0
```

Querying a `List<>` Using LINQ

- Let's consider we have a `List<>`.

```
List<string> colors = new List<string>();

colors.Add("aQua");
colors.Add("RusT");
colors.Add("yElLow");
colors.Add("rEd");
```

Querying a `List<>` Using LINQ

- Convert the list of colors to uppercase and search for those that begin with "R".

```
var startsWithR = from color in colors
                  let uppercaseString = color.ToUpper()
                  where uppercaseString.StartsWith("R")
                  orderby uppercaseString
                  select uppercaseString;
```

```
foreach (var i in startsWithR)
      Console.WriteLine(i);
```

Output:

```
RED
RUST
```

Querying a `List<>` Using LINQ

- LINQ's `let` clause can be used to create a new range variable to store a temporary result for use later in the LINQ query.
- The string method `ToUpper()` converts a string to uppercase.
- The string method `StartsWith()` performs a case sensitive comparison to determine whether a string starts with the string received as an argument.

Deferred Execution

- LINQ uses *deferred execution* – the query executes only when you access the results, not when you define the query.
- Let's add two more colors to the list:

```
items.Add("rUbY");
items.Add("SaFfRon");
```

- And print the result.

```
foreach (var i in startsWithR)
        Console.WriteLine(i);
```

```
Output:

RED
RUBY
RUST
```

Querying a `List<>` of Objects using LINQ

- Let's consider we have a class `Employee`.

```
class Employee
{
        public string FirstName { get; set; }
        public string LastName { get; set; }
        public decimal MonthlySalary { get; set; }

        public Employee(string firstname, string lastname, decimal salary)
        {
                FirstName = firstname;
                LastName = lastname;
                MonthlySalary = salary;
        }

        public override string ToString()
        {
                return $"{FirstName,-10} {LastName,-10} {MonthlySalary,10:C}";
        }
}
```

Querying a List<> of Objects using LINQ

- And using that class, let's assume we have list of employees.

```
List<Employee> employees = new List<Employee>
                           {
                                 new Employee("Jason", "Red", 5000),
                                 new Employee("Ashley", "Green", 7600),
                                 new Employee("Matthew", "Indigo", 3587.5),
                                 new Employee("James", "Indigo", 4700.77),
                                 new Employee("Luke", "Indigo", 6200),
                                 new Employee("Jason", "Blue", 3200),
                                 new Employee("Wendy", "Brown", 4236.4)
                           };
```

Querying a `List<>` of Objects using LINQ

- Using LINQ, fetch all the employees earning in the range $4000 and $6000:

```
var between4K6K = from emp in employees
                  where emp.MonthlySalary >= 4000 && emp.MonthlySalary <= 6000
                  select emp;
```

- A `where` clause can access the properties of the range variable.
- The conditional AND (`&&`) operator can be used to combine conditions.

```
Output:

Employees earning in the range $4,000.00-$6,000.00 per month:
Jason       Red          $5,000.00
James       Indigo       $4,700.77
Wendy       Brown        $4,236.40
```

Querying a `List<>` of Objects using LINQ

- An `orderby` clause can sort the results according to multiple properties, specified in a comma-separated list.

```
var nameSorted = from emp in employees
                 orderby emp.LastName, emp.FirstName
                 select emp;
```

```
Output:

Employees sorted by last name, then first name:
Jason       Blue          $3,200.00
Wendy       Brown         $4,236.40
Ashley      Green         $7,600.00
James       Indigo        $4,700.77
Luke        Indigo        $6,200.00
Matthew     Indigo        $3,587.50
Jason       Red           $5,000.00
```

Querying a `List<>` of Objects using LINQ

- The query result's `Any()` method returns `true` if there is at least one element, and `false` if there are no elements.
- The query result's `First()` method returns the first element in the result.

```
var nameSorted = from emp in employees
                 orderby emp.LastName, emp.FirstName
                 select emp;

Console.WriteLine("First employee when sorted by last name:");
if (nameSorted.Any())
       Console.WriteLine(nameSorted.First());
else
       Console.WriteLine("Not found");
```

Output:

```
First employee when sorted by last name:
Jason       Blue          $3,200.00
```

Querying a `List<>` of Objects using LINQ

- The `Count()` method of the query result returns the number of elements in the results.
- The `select` clause can be used to select a member of the range variable rather than the range variable itself.
- The `Distinct()` method removes duplicate elements, causing all elements in the result to be unique.

```
var lastNames = from emp in employees
                select emp.LastName;

Console.WriteLine("Unique last names:");
foreach (var e in lastNames.Distinct())
     Console.WriteLine(e);
```

```
Output:

Unique last names:
Red
Green
Indigo
Blue
Brown
```

Querying a `List<>` of Objects using LINQ

- The `select` clause can create a new object of *anonymous type* (a type with no name), which the compiler generates for you based on the properties listed in the curly braces (`{}`).
- By default, the name of the property being selected is used as the property's name in the result.

```
var names = from emp in employees
            select new { emp.FirstName, Last = emp.LastName };
```

- You can specify a different name for the property inside the anonymous type definition.

Output:

```
{ FirstName = Jason, Last = Red }
{ FirstName = Ashley, Last = Green }
{ FirstName = Matthew, Last = Indigo }
{ FirstName = James, Last = Indigo }
{ FirstName = Luke, Last = Indigo }
{ FirstName = Jason, Last = Blue }
{ FirstName = Wendy, Last = Brown }
```

Do It Yourself!

- **Exercise: Duplicate Word Removal:**
- Write an app that inputs a sentence from the user (assume no punctuation), then determines and displays the non-duplicate words in alphabetical order.
- Treat uppercase and lowercase letters the same.

- Hint:
  - You can use `string` method `Split` with no arguments, as in `sentence.Split()`, to break a sentence into an array of strings containing the individual words.
  - By default, `Split` uses spaces as delimiters.
  - Use `string` method `ToLower` in the `select` and `orderby` clauses of your LINQ query to obtain the lowercase version of each word.

Do It Yourself!

- **Exercise: Sorting Letters and Removing Duplicates:**
- Write an app that inserts 30 random letters into a `List<char>`.
- Perform the following queries on the `List` and display your results:
  - Use LINQ to sort the `List` in ascending order.
  - Use LINQ to sort the `List` in descending order.
  - Display the `List` in ascending order with duplicates removed.

- Hint:
  - Strings can be indexed like arrays to access a character at a specific index.

Do It Yourself!

- **Exercise: Find Small Words:**
- Consider the following array of strings:

```
string[] words = {"hello", "wonderful", "LINQ", "beautiful", "world"};
```

- Write a C# program that uses LINQ to find the words having length 5 or less.

Thank You

Copyright

The materials provided in class and in SLATE are protected by copyright. They are intended for the personal, educational uses of students in this course and should not be shared externally or on websites such as Chegg, Course Hero or OneClass. Unauthorized distribution may result in copyright infringement and violation of Sheridan policies.

References

Material has been taken from:

- Visual C# 2012: How to Program:
- https://learning.oreilly.com/library/view/visual-c-2012/9780133380170/ch09.html

- Professional C# 7 and .NET Core 2.0:
- https://learning.oreilly.com/library/view/professional-c-7/9781119449270/c12.xhtml