ASP.NET MVC

LINQ

issntt@nus.edu.sg

Objectives

At the end of this lesson, students will be able to

- Describe the roles of LINQ in .NET platform
- Describe the basic LINQ syntax for querying, filtering, ordering, projection, grouping, joining and aggregation
- Distinguish between two types of LINQ syntax: Query Syntax and Method Syntax
- Solve different types of problems using LINQ

Problem

Given a **list** or **array** of *Person* objects

```
List<Person> personList = new List<Person>()
{
   new Person() { Name = "John", Age = 32, Kids = 2},
   new Person() { Name = "Jessica", Age = 28, Kids = 3},
   new Person() { Name = "Mary", Age = 42, Kids = 2 },
   new Person() { Name = "Jason", Age = 33, Kids = 1 },
   new Person() { Name = "Mike", Age = 22, Kids = 0}
};
```

How can we get the **names** of the **3 youngest** people?

Object Initializer

Problem Solution - Algorithm

1

Sort the list of people **by age**, in **ascending order**

2

Get the **first 3** people from the sorted list

3

Get the respective **names**

Problem Solution – C#

```
static void SortList(List<Person> personList) {
  for (int i = 0; i < personList.Count; i++) {
    for (int j = i + 1; j < personList.Count; j++) {
      if (personList[j].Age < personList[i].Age) {
         // Switch position of two person
         Person temp = personList[i];
         personList[i] = personList[j];
         personList[j] = temp;
      }
    }
  }
}
```

1. **Sort** the list by age using Bubble Sort

```
static List<string>
        GetTopThreeNames(List<Person> personList) {
   List<string> retNames = new List<string>();
   for (int i = 0; i < 3; i++) {
      retNames.Add(personList[i].Name);
   }

   return retNames;
}
```

2. Get the **first 3** people, and

3. Get the respective **names**

Problem

If the list of people is stored in a **SQL Server database** table named `Person`, we can use **SQL**

```
SELECT TOP 3 Name
FROM Person
ORDER BY Age
```

Is there anything **like that** in C#?

LINQ

- Retrieves data from **C# collections**, e.g., List, Dictionary, Queue, Stack… and **database**
- Makes querying data a **first-class** programming concept
  - Queries as **part of C# code**
  - **Errors** are caught at **compile time**
- Bridges the **gap** between **object-oriented programming** languages and **relational databases**

Outline

- Making Queries and using Results
- Filtering
- Ordering
- Projecting
- Grouping
- Multiple Items
- Query Syntax vs Method Syntax
- Aggregate Functions
- Query Syntax and Method Syntax combination

Making Queries and Using Results

A LINQ query can be used to **retrieve data from** a **.NET collection** object

```
1 List<string> nameList =
      new List<string> {
          "John", "Mary", "Alexandra",
          "Christin", "Jessica" };

3 IEnumerable<string> iter =
        2 from name in nameList
          select name;

4 foreach (string na in iter)
      Console.WriteLine(na);
```

1. Let's say we have a list of strings in this example.
2. **LINQ** queries retrieve data from collections, in this case, the list *nameList*. Unlike SQL, *from* is **the first** statement of the query and *select* is always **the final**.
3. Depending on the query's *select* , the **return type** is different. In here, because we select *name* (**type string**), the return type will be an *IEnumerable<**string**>*.
4. Then we can **loop** over the *IEnumerable* object with *foreach*

```
John
Mary
Alexandra
Christin
Jessica
```

Filtering

Like SQL, query result can be **filtered** with ***where*** statement

```
List<Person> personList = new List<Person>()
{
   new Person() { Name = "John", Age = 32, Kids = 2},
   new Person() { Name = "Jessica", Age = 28, Kids = 3},
   new Person() { Name = "Mary", Age = 42, Kids = 2 },
   new Person() { Name = "Jason", Age = 33, Kids = 1 },
   new Person() { Name = "Mike", Age = 22, Kids = 0}
};

IEnumerable<Person> iter =
   from ps in personList
1  where ps.Age == 32 || ps.Kids == 2
   select ps;

foreach (Person p in iter)
{
   Console.WriteLine("{0}, {1}, {2}",
      p.Name, p.Age, p.Kids);
}
```

As a part of C# code, variable ***ps*** can **access** any public **methods** and **properties** of **class *Person***

1. Use *where* to filter, after which must be a boolean expression. Because LINQ query is a part of C# code, all **C# features** can be used. In this example, we:
   - Call **properties** (***Age, Kids***) of object *ps*
   - Use operators **==** and **||**

John, 32, 2
Mary, 42, 2

Ordering

Like SQL, query result can be **sorted** with ***orderby*** statement

```
List<Person> personList = new List<Person>()
{
   new Person() { Name = "John", Age = 32, Kids = 2},
   new Person() { Name = "Jessica", Age = 28, Kids = 3},
   new Person() { Name = "Mary", Age = 42, Kids = 2 },
   new Person() { Name = "Jason", Age = 33, Kids = 1 },
   new Person() { Name = "Mike", Age = 22, Kids = 0}
};

IEnumerable<Person> iter =
   from ps in personList
   where ps.Name.StartsWith("J")
1  orderby ps.Kids ascending
   select ps;

foreach (Person p in iter)
{
   Console.WriteLine("{0}, {1}, {2}",
      p.Name, p.Age, p.Kids);
}
```

1. Use *orderby* ***(ascending/descending)*** to sort. This statement is after *where*

Jason, 33, 1
John, 32, 2
Jessica, 28, 3

Demo

Image by Alexandra ♡A life without animals is not worth living♡ from Pixabay

Quiz

Given a **list** of *Person* objects

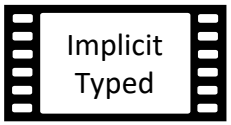
```
List<Person> personList = new List<Person>()
{
  new Person() { Name = "John", Age = 32, Kids = 2},
  new Person() { Name = "Jessica", Age = 28, Kids = 3},
  new Person() { Name = "Mary", Age = 42, Kids = 2 },
  new Person() { Name = "Jason", Age = 33, Kids = 1 },
  new Person() { Name = "Mike", Age = 22, Kids = 0}
};
```

Using LINQ, write a program displaying only people **having kids**, **sorted** by **name**?

Answer

```
public static void QuizFilterOrder() {
    var iter =
        from ps in personList
        where ps.Kids > 0
        orderby ps.Name
        select ps;

    foreach (var p in iter)
    {
        Console.WriteLine(p.Name + ", " +
                          p.Age + ", " + p.Kids);
    }
}
```



Projecting

Self study

Unlike SQL, result can only be projected using **C# anonymous type**

```
List<Person> personList = new List<Person>() {
   new Person() { Name = "John", Age = 32, Kids = 2},
   new Person() { Name = "Jessica", Age = 28, Kids = 3},
   new Person() { Name = "Mary", Age = 42, Kids = 2 },
   new Person() { Name = "Jason", Age = 33, Kids = 1 },
   new Person() { Name = "Mike", Age = 22, Kids = 0}
};
2 var iter =
   from ps in personList
   where ps.Kids >= 2
1 select new {
      ps.Name,
      ps.Age
   };
foreach (3 var p in iter)
{
   Console.WriteLine("{0}, {1}",
   4 p.Name, p.Age);
}
```

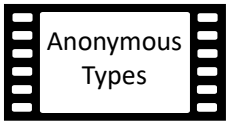
John, 32
Jessica, 28
Mary, 42

1. Unlike SQL, what comes after *select* statement has to be in a form of an **object**. Here we use C# **anonymous type** to create another type, which **only** has *Name* and *Age* properties (**no** *Kids* property).

2. Because we don't declare anonymous type anywhere, (i.e. unlike class *Person*), we cannot declare *IEnumerable<Type>* for variable *iter*. So we use *var* to ask the compiler to **infer** the necessary type for variable *iter* for us.

3. Same reason as 2., here we use *var* to ask the compiler to infer the necessary type for variable *p*.

4. The variable *p*, of the anonymous type, has **only** two properties, *Name* and *Age*, and does not have attribute *Kids*.



Projecting

Self study

Projecting with C# anonymous type can help us **create new properties** for object in the query result

```
List<Student> studentList = new List<Student>() {
   new Student() { First = "John", Last ="Tan", CAP = 3.5 },
   new Student() { First = "Jessica", Last="Ng", CAP = 3.0 },
   new Student() { First = "Mary", Last="Wong", CAP = 4.0 }
};

var iter =
   from student in studentList
   select new
   {
   1 Fullname = student.First + " " +
                                    student.Last,
      student.CAP
   };

foreach (var std in iter)
{
   Console.WriteLine("{0}, {1}",
   2 std.Fullname, std.CAP);
}
```

John Tan, 32
Jessica Ng, 28
Mary Wong, 42

1. The new property *FullName* is created. We can use C# expression to return its value and its type is inferred by the compiler. In here, its value is the concatenation of *First* and *Last* and its type is inferred to be string.

2. The variable *p*, with the anonymous type, has only two properties: *FullName* and *CAP*.

Quiz

Self study

Given a **list** of *Person* objects

```
List<Person> personList = new List<Person>()
{
    new Person() { Name = "John", Age = 32, Kids = 2},
    new Person() { Name = "Jessica", Age = 28, Kids = 3},
    new Person() { Name = "Mary", Age = 42, Kids = 2 },
    new Person() { Name = "Jason", Age = 33, Kids = 1 },
    new Person() { Name = "Mike", Age = 22, Kids = 0}
};
```

Write a program **displaying** each **person' name** and his/her **having kids status**. The output is as follows:

Hint: modular programming

John, Having 1 or 2 kids
Jessica, Having more than 2 kids
Mary, Having 1 or 2 kids
Jason, Having 1 or 2 kids
Mike, Having no kids yet

Answer

Self study

```
public static void QuizProjection() {
    var iter =
          from ps in personList
          select new {
              ps.Name,
              ChildrenStatus = GetChildrenStatus(ps.Kids)
          };

    foreach (var p in iter) {
        Console.WriteLine(p.Name + ", " + p.ChildrenStatus);
    }
}
```

```
public static string GetChildrenStatus(int noKids) {
    if (noKids == 0) {
        return "Having no kids yet";
    } else if (noKids == 1 || noKids == 2) {
        return "Having 1 or 2 kids";
    } else {
        return "Having more than 2 kids";
    }
}
```

Grouping

We can also group the objects in query results with ***group*** statement

```
List<Person> personList = new List<Person>()
{
   new Person() { Name = "John", Gender = "M", Kids = 2},
   new Person() { Name = "Jessica", Gender = "F", Kids = 3},
   new Person() { Name = "Mary", Gender = "F", Kids = 2},
   new Person() { Name = "Jason", Gender = "M", Kids = 1},
   new Person() { Name = "Mike", Gender = "M", Kids = 0},
   new Person() { Name = "David", Gender = "M", Kids = 2}
};

var iter = from ps in personList
      where ps.Kids > 0
   1 group ps by ps.Gender;

2 foreach (var grp in iter)
{
   Console.WriteLine("({0}) = {1}",
         3 grp.Key, grp.Count());

4 foreach (var p in grp)
      Console.WriteLine("   {0}, {1}",
         5 p.Name, p.Kids);
}
```

```
(M) = 3
   John, 2
   Jason, 1
   David, 2
(F) = 2
   Jessica, 3
   Mary, 2
```

1. Queries can end with *group* clause (i.e. without *select*). The results will take the form of a list of lists. Outside is a list of groups, and inside each group is a list of elements.

2. Iterate over each group.

3. Each group is an object that has a *Key* property and a list of elements that are grouped under the key. Besides, we can *Count()* or find *Max()*, *Min()*… in the group.

4. Iterate over each element. In this case, each element is a *Person* object because of the "from person" statement

5. Access the properties of the *Person's* object, as normal.

Grouping

Self study

If we must refer to the results of a group **operations** (like SQL Having), we can use the ***into*** keyword

```
List<Person> personList = new List<Person>() {
   new Person() { Name = "John", Gender = "M", Kids = 2},
   new Person() { Name = "Jessica", Gender = "F", Kids = 3},
   new Person() { Name = "Mary", Gender = "F", Kids = 2},
   new Person() { Name = "Jason", Gender = "M", Kids = 1},
   new Person() { Name = "Mike", Gender = "M", Kids = 0},
   new Person() { Name = "David", Gender = "M", Kids = 2}
};
var iter = from ps in personList
    where ps.Kids > 0
    group ps by ps.Gender
                    1 into personGroup
 2 where personGroup.Count() > 2
 3 select personGroup;

foreach (var grp in iter) {
   Console.WriteLine("({0}) = {1}",
               grp.Key, grp.Count());

  foreach (var p in grp)
      Console.WriteLine("   {0}, {1}",
               p.Name, p.Kids);
}
```

(M) = 3
John, 2
Jason, 1
David, 2

1. Use *into* to create an *identifier* for group for later use.

2. Use the group's operations through the identifier *personGroup*. Here, we use *Count()* method to filter only groups having more than 2 elements.

3. When using *into* keyword, we must declare the *select*. Note that we select the group using its identifier. In this case, it's *personGroup.*

Multiple Items

Like SQL JOIN, LINQ can query **multiple items** at once

```
string[] upperCase = { "A", "B", "C", "D" };
string[] lowerCase = { "c", "b", "a" };

var iter =
1 from up in upperCase
  from low in lowerCase
2 where up.ToLower() == low
  select new { up, low };

foreach (var item in iter)
  Console.WriteLine(item.up + "," + item.low);
```

1. We can use **multiple** *from* statements to query from **different sources**. Each has its own identifier.

2. Each identifier can call its **own methods** and use any operation that is allowed in C#. In this case, the type of variables *up* and *Low* are string. So variable *up* can use the string's *ToLower()* method.

```
A,a
B,b
C,c
```

What is the output if we remove the **where** statement?

Method Syntax

Self study

All queries so far are in **Query Syntax.** Queries can also be **directly** used within a collection object with **Method Syntax**

```
1 List<Person> personList = new
List<Person> {
   new Person() { Name = "John", Age = 32 },
   new Person() { Name = "Jessica", Age = 28 },
   new Person() { Name = "Mary", Age = 42 }
};

var iter = personList
   .Where(2 p =>3 p.Age > 30)
   .OrderByDescending(p => p.Age)
   .Select(p => p.Name);

foreach (var name in iter)
   Console.WriteLine(name);
```

```
from p in personlist
where p.age > 30
orderby p.age descending
select p.name;
```

Mary
John

Let's focus on explaining ***Where()*** function. Others follow the same manner

The input for *Where()* method is a **delegate function,** *Where(Func<TSource, Boolean> function).*

(1) ***TSource*** is the type inside the given collection. Here, because the type of variable *personList* is ***List<Person>***, *TSource* is ***Person***

It makes the delegate function input ***Func<Person, Boolean>,*** which should be a function having **1 argument typed *Person*** and **return a *Boolean* value**

Therefore, (2) **type of variable *p* is *Person*** and (3) we need an **expression that returns a *Boolean* value**. In here, it is *p.age > 30*

In short, *Where()* function **iterates** each *Person* object in the list. For each iteration, **the current object is referenced by variable *p*** (given by us, can be anything, e.g., x, y, z...). Only if *p.age > 30*, the current object will be stored in the final result.

Where https://docs.microsoft.com/en-us/dotnet/api/system.linq.enumerable.where
Select https://docs.microsoft.com/en-us/dotnet/api/system.linq.enumerable.select

Aggregate Functions

Self study

LINQ has built-in **aggregate** functions

```
List<Person> personList = new List<Person>()
{
      new Person() { name = "John", age = 32, kids = 2},
      new Person() { name = "Jessica", age = 28, kids = 3},
      new Person() { name = "Mary", age = 42, kids = 2 },
      new Person() { name = "Jason", age = 33, kids = 1 },
      new Person() { name = "Mike", age = 22, kids = 0}
};

int minAge = personList.Min(p => p.age);
int maxAge = personList.Max(p => p.age);
int avgAge = Convert.ToInt32(personList.Average(p => p.age));
int sumKids = personList.Sum(p => p.kids);
int numPerson = personList.Count();
```

Min age:22
Max age: 42
Average age: 31
Sum kids: 8
Num of persons: 5

Syntax Combination

We can **combine** Query Syntax and Method Syntax in **one query**

```
List<Person> personList = new List<Person>()
{
   new Person() { name = "John", age = 32, kids = 2},
   new Person() { name = "Jessica", age = 28, kids = 3},
   new Person() { name = "Mary", age = 22, kids = 2 },
   new Person() { name = "Jason", age = 33, kids = 1 },
   new Person() { name = "Mike", age = 22, kids = 0}
};

var youngestNames = from p in personList
          where p.age == personList.Min(p => p.age)
          select p.name;

foreach (var name in youngestNames)
{
   Console.WriteLine(who);
}
```

1

1. First, we use *method syntax* to find the **minimum age**. Then, we use *query syntax* to find people **having that minimum age.**

Mary
Mike

Great News!

All the LINQ queries we have learnt can be **also** used to **query** data from **database**!

Problem – A LINQ Solution

Get the **names** of the **3 youngest** people

```
var iter = (from ps in personList
            orderby ps.Age
            select ps.Name).Take(3);
```

Take(3) means only get the first 3 elements

Or

```
var iter = personList
            .OrderBy(ps => ps.Age)
            .Select(ps => ps.Name)
            .Take(3);
```

Then

```
foreach (var name in iter)
      Console.WriteLine(name);
```

Readings

- Basic LINQ Query Operations https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/concepts/linq/basic-linq-query-operations

- Object Initializer https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/classes-and-structs/object-and-collection-initializers