

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

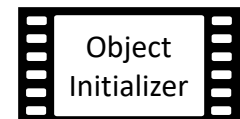
Problem

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

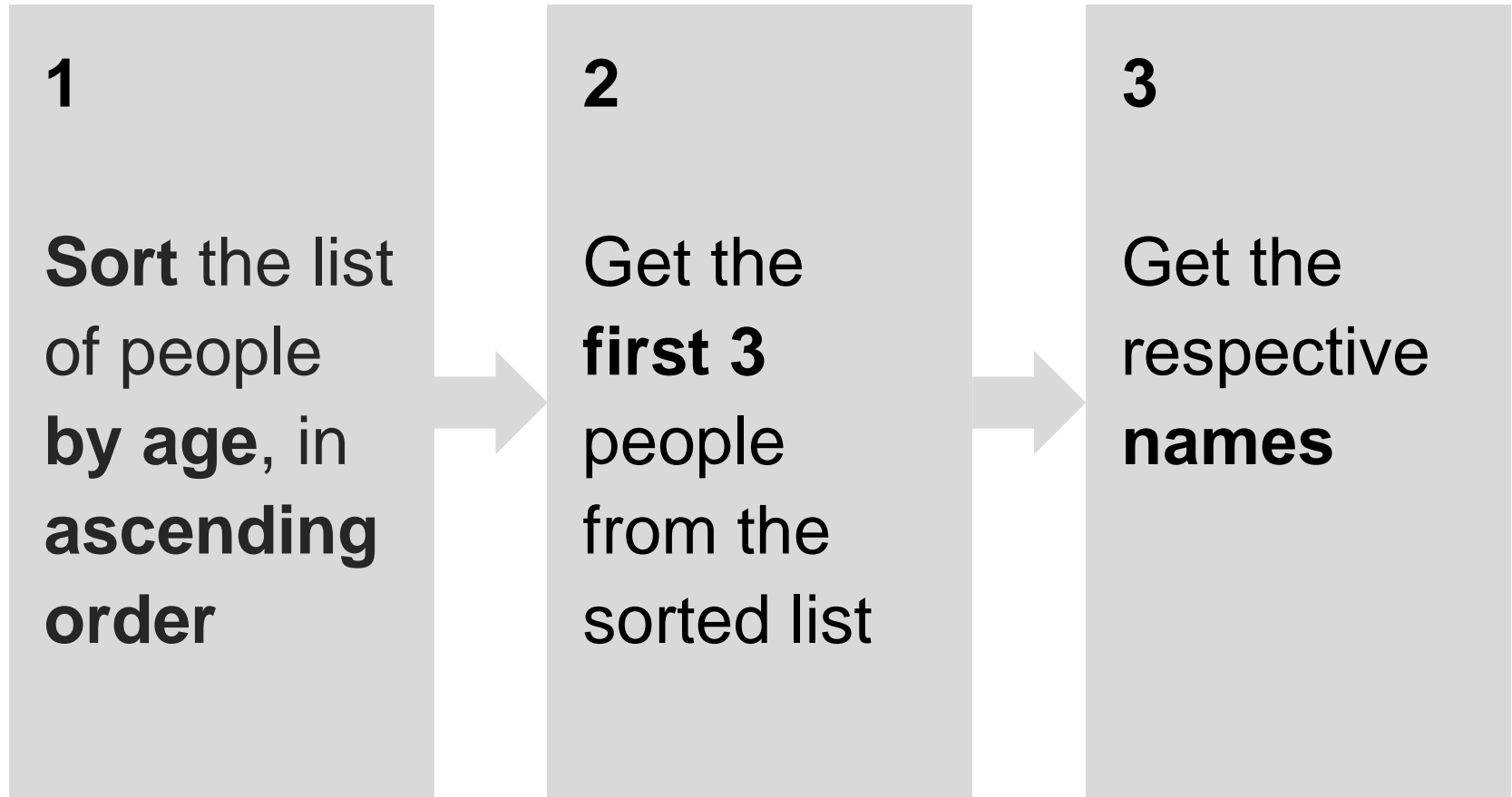
```
List<Person> personList = new List<Person>()  
{  
    new Person() { Name = "John", Age = 32, Kids = 2},  
    new Person() { Name = "Jessica", Age = 28, Kids = 3},  
    new Person() { Name = "Mary", Age = 42, Kids = 2 },  
    new Person() { Name = "Jason", Age = 33, Kids = 1 },  
    new Person()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



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



Problem Solution - Algorithm



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#?

- 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**

- 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);
```


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}
};
```



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

```
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);
}
```

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);
}
```



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**?

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);
}
```

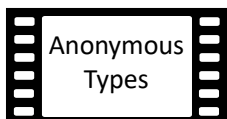
John, 32
Jessica, 28
Mary, 42

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*.

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's** 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

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);
}
```



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);
}
```

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*.

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

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);
```



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;
```

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 == 1 personList.Min(p => p.age)
                    select p.name;

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

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>