COMP4701 – Fall 2025

6 – LINQ and Entity Frameworks

Dr. Abdullah Al-Hamdani

5 – LINQ and Data Frameworks

- LINQ (Language Integrated Query)
- Lambda Expressions
- LINQ with Database Using Entity Framework

Exercise

- Define array with the following numbers

  99,44,33,77,88,55,66,22,11,1,123

Design a form that reads the range as two numbers (max, min) and displays the numbers from the array that are within the specified range.

Ex: min=50 and max=80

77,55,66

```
int [] list = {99,44,33,77,88,55,66,22,11,1,123};
var  result=from x in list
                where x>=50 && x<=80
                select s;
```

Random Numbers

```
Int size = 10;
int[] numbers=new int[size]; //define array of 10 elements

//different numbers in multiple runs
Random rand = new Random();

for (int i = 0; i < size; i++)
    numbers[i] = rand.Next(60, 101); //random number [60,100]

-----------------------------------------------------
***** Use seed to generate the same random numbers in
multiple runs
int seed =100; //use another number to display different sequence
Random rand = new Random(seed);
```

Introduction

- A `List` is similar to an array but provides additional functionality, such as dynamic resizing.
- A language called SQL is the international standard used to perform queries (i.e., to request information that satisfies given criteria) and to manipulate data.
- C#'s LINQ (Language-Integrated Query) capabilities allow you to write query expressions that retrieve information from a *variety* of data sources, not just databases.
- LINQ to Objects can be used to filter arrays and `Lists`, selecting elements that satisfy a set of conditions

Introduction (Cont.)

- There are two LINQ approaches
  - –One uses a SQL-like syntax :
    - LINQ
  - –The other uses method-call syntax.
    - introducing the notions for lambda expressions

Querying an Array of int Values Using LINQ

- Example: shows how to use *LINQ to Objects* to query an array of integers, selecting elements that satisfy a set of conditions
  - This process is called filtering

- The `System.Linq` namespace contains the LINQ to Objects provider.

Linq Clauses

- Similar to SQL statement, Linq has the following 5 clauses
  - from Clause
    - A LINQ query begins with a `from` clause, which specifies a range variable (`value`) and the data source to query (`values`).
      - The range variable represents each item in the data source, much like the control variable in a `foreach` statement.
  - where Clause
    - If the condition in the `where clause` evaluates to `true`, the element is selected.
    - Similar to the condition in if statements in C#
  - select clause:
    - Determines what value appears in the results.
  - orderby clause:
    - sorts the query results in ascending order.
    - The descending modifier in the `orderby` clause sorts the results in descending order.
  - Group by – Similar to Group by in SQL

Interface IEnumerable<T>

- The `IEnumerable<T>` interface describes the functionality of any object that can be iterated over and thus offers members to access each element.
- Arrays and collections already implement the `IEnumerable<T>` interface.
- A LINQ query returns an object that implements the `IEnumerable<T>` interface.
- With LINQ, the code that selects elements and the code that displays them are kept separate, making the code easier to understand and maintain.

```
2  // LINQ to Objects using an int array.
3  using System;
4  using System.Linq;
5
6  class LINQWithSimpleTypeArray
7  {
8     static void Main()
9     {
10       // create an integer array
11       var values = new[] {2, 9, 5, 0, 3, 7, 1, 4, 8, 5};
12
13       // display original values
14       Console.Write("Original array:");
15       foreach (var element in values)
16       {
17          Console.Write($" {element}");
18       }
19
20       // LINQ query that obtains values greater than 4 from the array
21       var filtered =
22          from value in values // data source is values
23          where value > 4
24          select value;
25
26       // display filtered results
27       Console.Write("\nArray values greater than 4:");
28       foreach (var element in filtered)
29       {
30          Console.Write($" {element}");
31       }
32
```

Linq - Order by

```
33          // use orderby clause to sort original values in ascending order
34          var sorted =
35             from value in values // data source is values
36             orderby value
37             select value;
38
39          // display sorted results
40          Console.Write("\nOriginal array, sorted:");
41          foreach (var element in sorted)
42          {
43             Console.Write($" {element}");
44          }
45
46          // sort the filtered results into descending order
47          var sortFilteredResults =
48             from value in filtered   // data source is LINQ query filtered
49             orderby value descending
50             select value;
51
52          // display the sorted results
53          Console.Write(
54             "\nValues greater than 4, descending order (two queries):");
55          foreach (var element in sortFilteredResults)
56          {
57             Console.Write($" {element}");
58          }
59
```

Linq - Where

```
60          // filter original array and sort results in descending order
61          var sortAndFilter =
62             from value in values     // data source is values
63             where value > 4
64             orderby value descending
65             select value;
66
67          // display the filtered and sorted results
68          Console.Write(
69             "\nValues greater than 4, descending order (one query):");
70          foreach (var element in sortAndFilter)
71          {
72             Console.Write($" {element}");
73          }
74
75          Console.WriteLine();
76       }
77    }
```

```
Original array: 2 9 5 0 3 7 1 4 8 5
Array values greater than 4: 9 5 7 8 5
Original array, sorted: 0 1 2 3 4 5 5 7 8 9
Values greater than 4, descending order (two queries): 9 8 7 5 5
Values greater than 4, descending order (one query): 9 8 7 5 5
```

Exercise

- Assume we have the following array

  int [] score = {99,70,88,21,66,81}

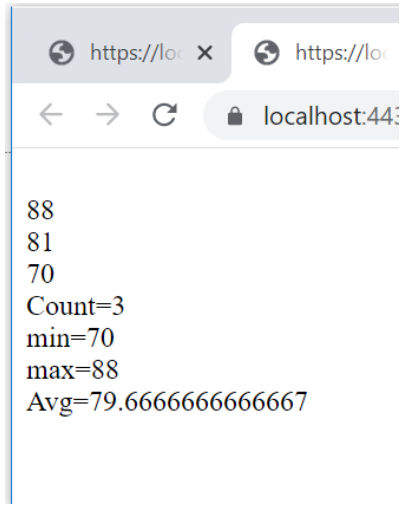
- Display the number of elements between 70 to 90

- Count the number of elements between 70 to 90

- Find the average, maximum and minimum value between 70 and 90

Exercise

```
int[] scores = { 99, 70, 88, 21, 66, 81 };

var list = from s in scores
           where s >= 70 && s <= 90
           orderby s descending
           select s;

ViewData["message"] = "";
foreach (int x in list)
    ViewData["message"] += $"<br/> {x}";

ViewData["message"] += "<br/> Count=" + list.Count();
ViewData["message"] += "<br/> min=" + list.Min();
ViewData["message"] += "<br/> max=" + list.Max();
ViewData["message"] += "<br/> Avg=" + list.Average();
```



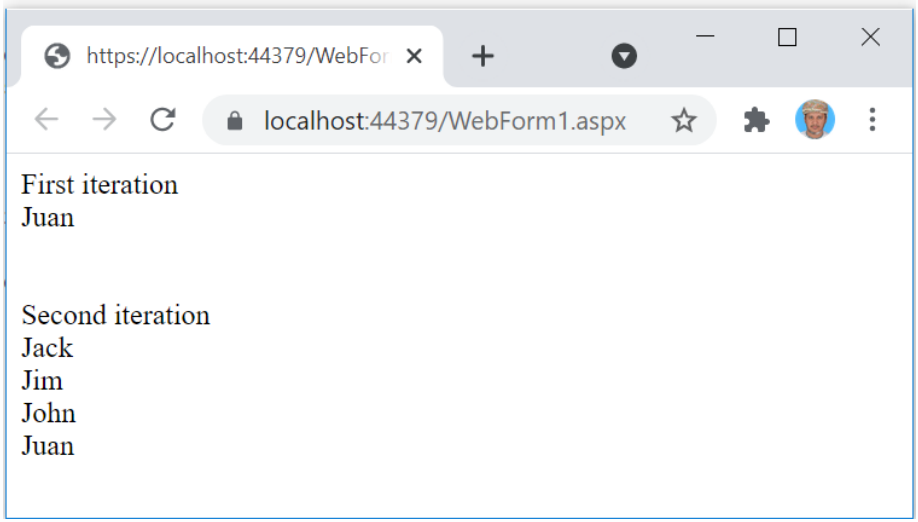
Example

```
int[] scores = { 99, 70, 88, 21, 66, 81 };

var list = from s in scores
           where s >= 70 && s <= 90
           orderby s descending
           select $"The number is {s}";

String message = "";
foreach (string x in list)
       message += $"<br/> {x}";
```

Deferred Query Execution

- When the query expression is defined during runtime, the query does not run. The query runs when the items are iterated.
- Linq query is invoked every time the query is used within an iteration.

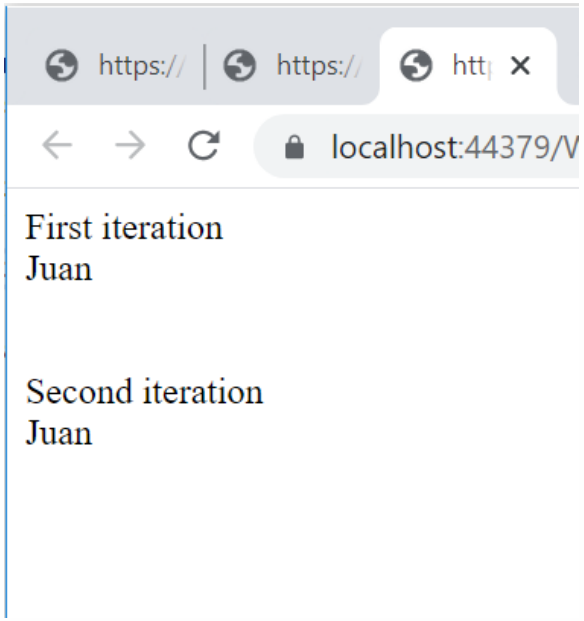
```
var names = new List<string> { "Nino", "Alberto", "Juan", "Mike", "Phil" };
var namesWithJ = from n in names
                 where n.StartsWith("J")
                 orderby n
                 select n;
ViewData["message"] ="First iteration";
foreach (string name in namesWithJ)
        message += "<br/>"+name;

ViewData["message"] += "<br/>";
names.Add("John");
names.Add("Jim");
names.Add("Jack");
names.Add("Denny");
ViewData["message"] += "<br/><br/>Second iteration";
foreach (string name in namesWithJ)
        ViewData["message"] += "<br/>" + name;
```



ToList/ToArray Method

- ToList iterates through the collection immediately and returns a collection implementing IList<string>.

```
var names = new List<string> { "Nino", "Alberto", "Juan", "Mike", "Phil" };
var namesWithJ = (from n in names
                  where n.StartsWith("J")
                  orderby n
                  select n).ToList();
ViewData["message"] ="First iteration";
foreach (string name in namesWithJ)
 ViewData["message"] += "<br/>"+name;

ViewData["message"] += "<br/>";
names.Add("John");
names.Add("Jim");
names.Add("Jack");
names.Add("Denny");
ViewData["message"] += "<br/><br/>Second iteration";
foreach (string name in namesWithJ)
      ViewData["message"] += "<br/>" + name;
```




```
List<string> namesWithJ = (from n in names where n.StartsWith("J")
                           orderby n select n).ToList();
```

```
String [] namesWithJ = (from n in names where n.StartsWith("J")
                        orderby n select n).ToArray();
```

Linq – Group by

```
List<int> list = new List<int>() { 1, 2, 3, 4, 5, 1, 2,
3, 2, 1, 4, 3, 5, 3, 2, 1 };

var tt = from x in list
         group x by x into w
         having
         select new { key= w.Key, count=w.Count() };

ViewData["message"] = "";
foreach (var m in tt)
    ViewData["message"] += $"<br/> {m.key} and
{m.count}";
```

Querying an Array of Employee Objects Using LINQ

- LINQ is not limited to querying arrays of simple types such as `integers`.
- It cannot be used when a query does not have a defined meaning—for example, you cannot use orderby on objects values that are not *comparable*.
- Comparable types in .NET are those that implement the `IComparable<T>`.
  - All built-in types, such as `string`, `int` and double implement `IComparable<T>`.

Defining Class

```
1  // Fig. 9.3: Employee.cs
2  // Employee class with FirstName, LastName and MonthlySalary properties.
3  class Employee
4  {
5     public string FirstName { get; } // read-only auto-implemented property
6     public string LastName { get; } // read-only auto-implemented property
7     private decimal monthlySalary; // monthly salary of employee
8
9     // constructor initializes first name, last name and monthly salary
10    public Employee(string firstName, string lastName,
11       decimal monthlySalary)
12    {
13       FirstName = firstName;
14       LastName = lastName;
15       MonthlySalary = monthlySalary;
16    }
17
```

Defining Class

```
18      // property that gets and sets the employee's monthly salary
19      public decimal MonthlySalary
20      {
21         get
22         {
23            return monthlySalary;
24         }
25         set
26         {
27            if (value >= 0M) // validate that salary is nonnegative
28            {
29               monthlySalary = value;
30            }
31         }
32      }
33
34      // return a string containing the employee's information
35      public override string ToString() =>
36         $"{FirstName,-10} {LastName,-10} {MonthlySalary,10:C}";
37   }
```

Defining List of Class

```
3  using System;
4  using System.Linq;
5
6  class LINQWithArrayOfObjects
7  {
8     static void Main()
9     {
10       // initialize array of employees
11       var employees = new[] {
12          new Employee("Jason", "Red", 5000M),
13          new Employee("Ashley", "Green", 7600M),
14          new Employee("Matthew", "Indigo", 3587.5M),
15          new Employee("James", "Indigo", 4700.77M),
16          new Employee("Luke", "Indigo", 6200M),
17          new Employee("Jason", "Blue", 3200M),
18          new Employee("Wendy", "Brown", 4236.4M)};
20       // display all employees
21       Console.WriteLine("Original array:");
22       foreach (var element in employees)
23       {
24          Console.WriteLine(element);
25       }
26
27       // filter a range of salaries using && in a LINQ query
28       var between4K6K =
29          from e in employees
30          where (e.MonthlySalary >= 4000M) && (e.MonthlySalary <= 6000M)
31          select e;
32
33       // display employees making between 4000 and 6000 per month
34       Console.WriteLine("\nEmployees earning in the range" +
35          $"{4000:C}-{6000:C} per month:");
36       foreach (var element in between4K6K)
37       {
38          Console.WriteLine(element);
39       }
```

LINQ query operators

- Where
- Select
- OrderBy
- ThenBy
- OrderByDescending
- ThenByDescending
- Reverse
- Join
- GroupJoin
- ToArray
- AsEnumerable
- ToList
- ToDictionary
- Cast<TResult>
- GroupBy
- ToLookup
- Any
- All
- Contains
- Distinct
- Union
- Intersect
- Except
- Zip
- Empty
- Range
- Repeat
- First
- FirstOrDefault
- Last
- LastOrDefault
- ElementAt
- ElementAtOrDefault
- Single
- SingleOrDefault
- Count
- Sum
- Min
- Max
- Average

Any, First, Count

- 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.
- The `Count` method of the query result returns the number of elements in the results.

```
40
41          // order the employees by last name, then first name with LINQ
42          var nameSorted =
43              from e in employees
44              orderby e.LastName, e.FirstName
45              select e;
46
47          // header
48          Console.WriteLine("\nFirst employee when sorted by name:");
49
50          // attempt to display the first result of the above LINQ query
51          if (nameSorted.Any())
52          {
53              Console.WriteLine(nameSorted.First());
54          }
55          else
56          {
57              Console.WriteLine("not found");
58          }
```

Distinct

```
59
60          // use LINQ to select employee last names
61          var lastNames =
62             from e in employees
63             select e.LastName;
64
65          // use method Distinct to select unique last names
66          Console.WriteLine("\nUnique employee last names:");
67          foreach (var element in lastNames.Distinct())
68          {
69             Console.WriteLine(element);
70          }
71
```

Using new { ....}

- **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 ({}).**

```
new {e.FirstName, e.LastName}
```

```
72          // use LINQ to select first and last names
73          var names =
74             from e in employees
75             select new {e.FirstName, e.LastName};
76
77          // display full names
78          Console.WriteLine("\nNames only:");
79          foreach (var element in names)
80          {
81             Console.WriteLine(element);
82          }
83
84          Console.WriteLine();
85       }
86    }
```

Output

```
Original array:
Jason       Red          $5,000.00
Ashley      Green        $7,600.00
Matthew     Indigo       $3,587.50
James       Indigo       $4,700.77
Luke        Indigo       $6,200.00
Jason       Blue         $3,200.00
Wendy       Brown        $4,236.40

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

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

Output

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

Names only:
{ FirstName = Jason, LastName = Red }
{ FirstName = Ashley, LastName = Green }
{ FirstName = Matthew, LastName = Indigo }
{ FirstName = James, LastName = Indigo }
{ FirstName = Luke, LastName = Indigo }
{ FirstName = Jason, LastName = Blue }
{ FirstName = Wendy, LastName = Brown }
```

Example

- Define Student Class as follows

```
public class Student {
    public int sid {  get; set; }
    public char gender {  get; set; }
    public int age { get; set; }
    public string grade { get; set; }
};
```

Student Collection

```
Student[] myS ={ new Student{ sid=111, gender='M', age=20,
grade="A"},new Student { sid=099, gender='F', age=21, grade="B"},new
Student { sid=222, gender='M', age=22, grade="A"},new Student { sid=333,
gender='F', age=22, grade="B"},new Student { sid=444,gender='M', age=20,
grade="C"},new Student { sid=555,gender='M', age=19, grade="B"}};

foreach(Student s in myS)
{
 <p>sid=@s.sid, name=@s.gender,
   Gender=@s.age,Age=@s.grade </p>
 }
```

sid=111, name=M, Gender=20, Age=A

sid=99, name=F, Gender=21, Age=B

sid=222, name=M, Gender=22, Age=A

sid=333, name=F, Gender=22, Age=B

sid=444, name=M, Gender=20, Age=C

sid=555, name=M, Gender=19, Age=B

Linq – Group by

- For all students with Sid>100, find number of students per each grade; given that there is at least two students in each grade

```
var list = from s in myS
           where s.sid > 100
           group s by s.grade into m
           where m.Count()>1
           select new { sGrade = m.Key, Nos = m.Count() };
<h2>Group by Age</h2>
foreach(var x in list)
{
   <p>Grade: @x.sGrade
     ==> student are @x.Nos</p>
}
```

Group by Age

Grade: A ==> student are 2

Grade: B ==> student are 2

Linq – Group by

- Find the number of students, average age, youest age and highest SID for each gender.

Group by Gender

Gender: M ==> #Students: 4, Average Age: 20.25, YougestAge: 19, Highest SID: 555

Gender: F ==> #Students: 2, Average Age: 21.5, YougestAge: 21, Highest SID: 333

```
var StudentsbyGender = from s in myS
    group s by s.gender into w
    select new { gender = w.Key,NoStudent = w.Count(),
            avgAge = w.Average(x => x.age),
            yougest = w.Min(x => x.age),
            highestSID = w.Max(x=>x.sid)};
<h2>Group by Gender</h2>
foreach (var x in StudentsbyGender){
   <p> Gender: @x.gender ==> #Students: @x.NoStudent,
      Average Age: @x.avgAge, YougestAge: @x.yougest,
      Highest SID: @x.highestSID</p> }
```

Lambda Expressions

- Similar to Linq, the Lamb expression is used to specify expressions using collections.

- Symbol => is the lambda operator which is used in all lambda expressions.
- The Lambda Expressions can be of two types:
  - Expression Lambda: Consists of the input and the expression.
    - Syntax: input => expression;
  - Statement Lambda: Consists of the input and a set of statements to be executed.
    - Syntax: input => { statements };

Lambda Expressions

- Shortened syntax can be used anywhere in ASP.Net and C# code.
- For example, if you were to build a trivial class to add two numbers, you might write the following:

```
class SimpleMath
{
  public int Add(int x, int y)
  {
    return x + y;
  }
  public void PrintSum(int x, int y)
 {
   Console.WriteLine(x + y);
  }
}
```

Can be simplified into

```
class SimpleMath
{
  public int Add(int x, int y) => x + y;
  public void PrintSum(int x, int y) => Console.WriteLine(x + y);
}
```

Lambda Expressions
Select Method

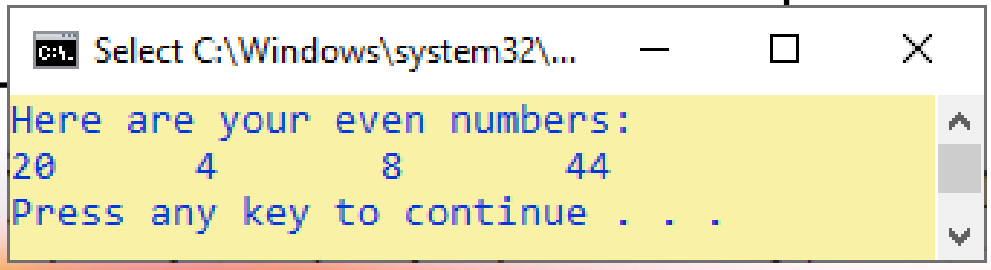
- **You also can use lambda expressions when you write LINQ in C#**

```
int[ ] numbers = { 2, 3, 4, 5 };
var squaredNumbers = numbers.Select(x => x * x);

ViewData["message"]="“;
foreach (int x in squaredNumbers)
      ViewData["message"] += $"   {x} ";
```

4 9 16 25

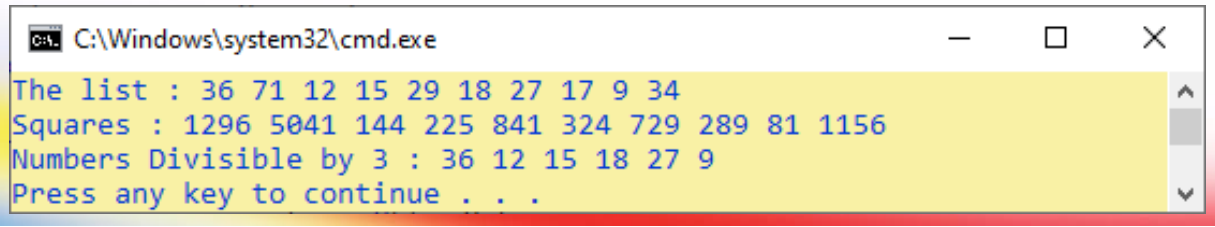
Lambda Expressions
FindAll Method

```
static void LambdaExpressionSyntax()
{
   List<int> list = new List<int>() { 20, 1, 4, 8, 9, 44 };

  // Now, use a C# lambda expression.
  List<int> evenNumbers=list.FindAll(i => (i % 2) == 0);

  Console.WriteLine("Here are your even numbers:");
  foreach (int evenNumber in evenNumbers)
  {
      Console.Write("{0}\t", evenNumber);
  }
  Console.WriteLine();
}
```



Example – Lambda Expression Select & FindAll

```
static void Main(string[] args) {
        // List to store numbers
        List<int> numbers = new List<int>() {36, 71, 12, 15, 29, 18, 27, 17, 9, 34};
        // foreach loop to dislay the list
        Console.Write("The list : ");
        foreach (var value in numbers) {  Console.Write("{0} ", value);}
        Console.WriteLine();

        // Using lambda expression to calculate square of each value in the list
        var square = numbers.Select(x => x * x);

        // foreach loop to display squares
        Console.Write("Squares : ");
        foreach (var value in square) {Console.Write("{0} ", value); }
        Console.WriteLine();

        // Using Lambda exression to find all numbers in the list  divisible by 3
        List<int> divBy3 = numbers.FindAll(x => (x % 3) == 0);

        // foreach loop to display divBy3
        Console.Write("Numbers Divisible by 3 : ");
        foreach (var value in divBy3) { Console.Write("{0} ", value); }
        Console.WriteLine();
}
```



Display the index for each element

```
int[] numbers = { 44, 33, 66, 77, 99, 22, 11, 1, 8, 4 };

<h2>List the index for each element</h2>
var numberWithIndex =
      numbers.Select((x, i) => new{i,x});

foreach(var w in numberWithIndex)
{
   <p>index=@w.i and value=@w.x</p>
}
```

index=0 and value=44

index=1 and value=33

index=2 and value=66

index=3 and value=77

index=4 and value=99

index=5 and value=22

index=6 and value=11

index=7 and value=1

index=8 and value=8

index=9 and value=4

Using Index as a filter

- Find elements with even index numbers (index=0, 2,4,..)

index 0 1 2 3 4 5 6 7 8 9

```
int[] numbers = { 44, 33, 66, 77, 99, 22, 11, 1, 8, 4 };
<h2>List elements with even indices</h2>
var numberWithIndex = numbers.Where((x, i) => i%2==0);
foreach(var w in numberWithIndex)
{
   <p>value=@w</p>
}
```

value=44

value=66

value=99

value=11

value=8

More LINQ Examples
Filtering

```
var racers =    from r in Formula1.GetChampions()
                where r.Wins > 15 &&
                (r.Country == "Brazil" || r.Country == "Austria")
                select r;
foreach (var r in racers)
      Console.WriteLine("{0:A}", r);
```

```
var racers = Formula1.GetChampions().
        Where(r => r.Wins > 15 &&
        (r.Country == "Brazil" || r.Country == "Austria"));
```

```
Niki Lauda, Austria, Starts: 173,
Wins: 25
Nelson Piquet, Brazil, Starts: 204,
Wins: 23
```

Filtering with Index

```
var racers = Formula1.GetChampions().
        Where((r, index) => r.LastName.StartsWith("A")
        && index % 2 != 0);
foreach (var r in racers)
{
    Console.WriteLine("{0:A}", r);
}
```

```
Alberto Ascari, Italy; starts: 32, wins:
10
Fernando Alonso, Spain; starts: 177,
wins: 27
```

Type Filtering

```
object[] data = { "one", 2, 3, "four", "five", 6 };
var query = data.OfType<string>();
foreach (var s in query)
{
    Console.WriteLine(s);
}
```

```
one
four
five
```

Compound from

```
var ferrariDrivers = from r in Formula1.GetChampions()
                     from c in r.Cars
                     where c == "Ferrari"
                     orderby r.LastName
                     select r.FirstName + " " + r.LastName;
```

```
Alberto Ascari
Juan Manuel Fangio
Mike Hawthorn
Phil Hill
Niki Lauda
Kimi Räikkönen
Jody Scheckter
Michael Schumacher
John Surtees
```

Order by

```
var racers = from r in Formula1.GetChampions()
where r.Country == "Brazil"
orderby r.Wins descending
select r;
```

```
var racers = Formula1.GetChampions().
Where(r => r.Country == "Brazil").
OrderByDescending(r => r.Wins).
Select(r => r);
```

Order By more than one attribute

```
var racers = (from r in Formula1.GetChampions()
orderby r.Country, r.LastName, r.FirstName
select r).Take(10);
```

```
var racers = Formula1.GetChampions().
OrderBy(r => r.Country).
ThenBy(r => r.LastName).
ThenBy(r => r.FirstName).
Take(10);
```

```
Argentina: Fangio, Juan Manuel
Australia: Brabham, Jack
Australia: Jones, Alan
Austria: Lauda, Niki
Austria: Rindt, Jochen
Brazil: Fittipaldi, Emerson
Brazil: Piquet, Nelson
Brazil: Senna, Ayrton
Canada: Villeneuve, Jacques
Finland: Hakkinen, Mika
```

Grouping

```
var countries =     from r in Formula1.GetChampions()
                    group r by r.Country into g
                    orderby g.Count() descending, g.Key
                    where g.Count() >= 2
                    select new { Country = g.Key, Count = g.Count()};
foreach (var item in countries)
        Console.WriteLine("{0, -10} {1}", item.Country, item.Count);
```

```
var countries =     Formula1.GetChampions().
                    GroupBy(r => r.Country).
                    OrderByDescending(g => g.Count()).
                    ThenBy(g => g.Key).
                    Where(g => g.Count() >= 2).
                    Select(g => new { Country = g.Key,
                                      Count = g.Count() });
```

```
Finland 3
Australia 2
Austria 2
Germany 2
Italy 2
USA 2
```

Exercise

- Define a class to maintain information about a product including product ID, product name, product quantity, and product price
- Define a list of products as a dynamic list with at least three products.
- Use LINQ queries to perform the following with appropriate ASP.Net controls.
  - Find products with a specific name
  - Find products with a price less than a given price
  - Find a product with ZERO quantity
  - List product name and price for all products ordered by product name.
  - List all products in decedent ordered of product price
  - Add new product
  - Remove a product by name
  - Find the product with the lowest price
  - Display the total quantity and price for all products
  - List product names and prices for all products within two given prices

Connecting Linq with Database using Entity Framework

Using Entity Data Model

- ADO.NET Entity Framework offers several layers to map database tables to objects.
- Database First
  - You can first create with a database schema using DBMS
  - Then, use a Visual Studio item template to create the complete mapping to Entity classes.
- Model First
  - You can also start designing entity classes with the designer (Model First) and
  - Map it to the database such that the tables and the associations between the tables can have a very different structure.

Using Entity Data Model

- Composed of three layers as follows:
  - Logical: Defines the relational data.
    - Create database
  - Conceptual: Defines the .NET entity classes.
    - Connect database to Linq using Entity Framework classes
  - Mapping: Defines the mapping from .NET classes to relational tables and associations
    - Use Linq expressions to manipulate the database
- Example:
  - mapping an existing database to Entity Model framework classes

1- Database Construction

- Create new Database with tables
  - Students, Courses, Results
- Import existing databases from other projects
  - Copy App_Data from other ASP.Net Project
  - Project ➔ show all folders
  - Right Click on App_Data Folder
  - Choose “Include in the project”

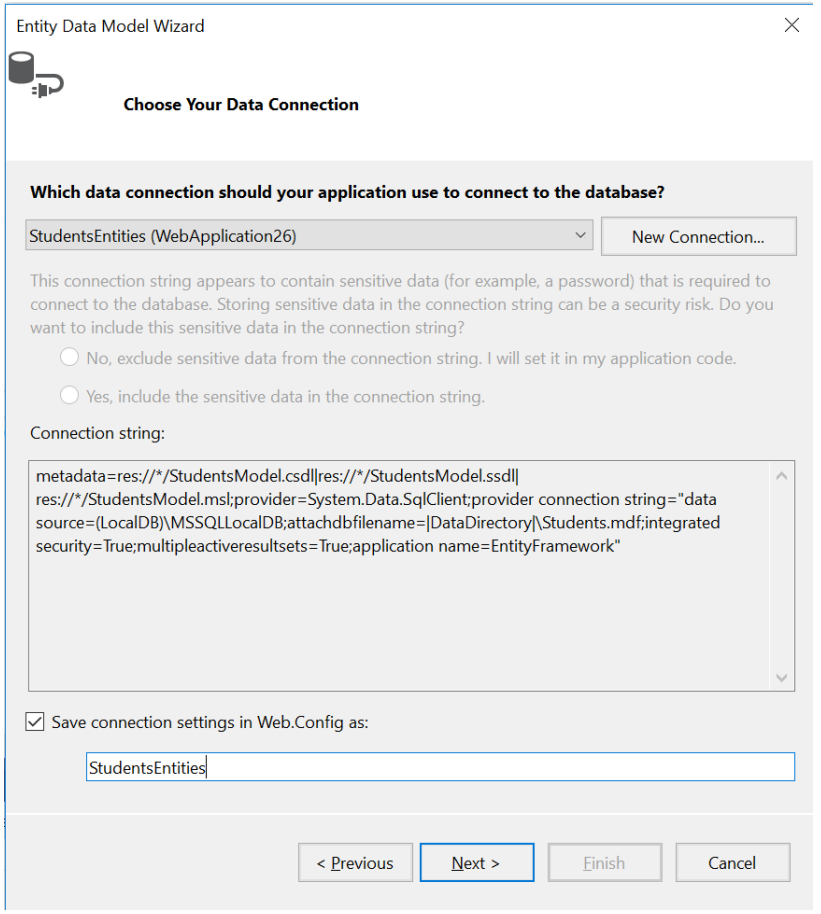
  OR
  - Project ➔ Add ASP.Net folder (if it does not exist)
  - Project Add existing item ➔ choose existing Database

Adding Entity Framwork

- From Project, go to Manage NuGet Package
- Search for “EntityFrameworkCore”
- Add the following packages
  - Microsoft.EntityFrameworkCore
  - Microsoft.EntityFrameworkCore.Relational
  - Microsoft.EntityFrameworkCore.Abstractions
  - Microsoft.EntityFrameworkCore.Analyzers
  - Microsoft.EntityFrameworkCore.Design
  - Microsoft.EntityFrameworkCore.Tools
  - Microsoft.EntityFrameworkCore.SqlServer
- You need to choose a compatible version for each package
  - e.g. If v9 is not working then use v8 or v7 for all packages

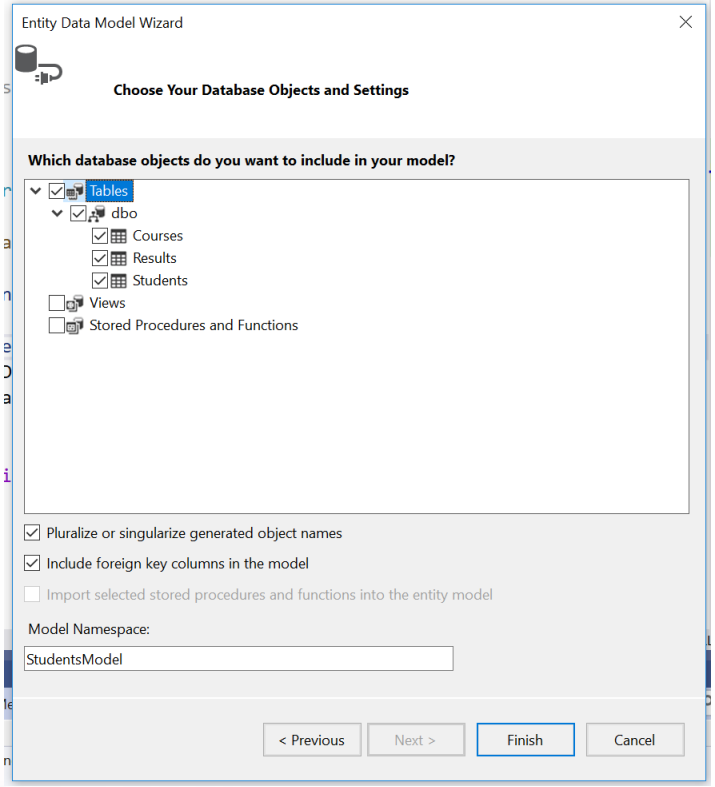
2- Creating Entity Data Model (VS 2019)

- Project Add New Item
- Data ➔ ADO.Net Entity Data Model
- Specify ➔ Model name (e.g. StudentsModel)
- Choose Model component as EF Designer from Database
- Choose Database Connection
  - E.g. Students.mdf
- Choose Save Connection String in WebConfig as
  - StudentsEntities



2- Creating Entity Data Model (VS 2019)

- Choose version of Entity Framework as
  - Entity Framework 6.x (or newer version)
- Choose the database objects and setting to be included in the model
  - Tables (include others if applicable views, stored procedures, functions) and other constraints such as foreign keys
- Choose the model Namespace
  - E.g., StudentsModel



Using Scaffold Statement (VS 2022)

- You can create the Db Conext and the corresponding classes for DB tables
  - From Tools ➔ NuGet Package Manager ➔Package Manager Console
  - Type the following command

scaffold-dbcontext “DB_Connection_String” MicroSoft.EntityFrameWorkCore.SqlServer -OutputDir folderName

E.g.:

scaffold-dbcontext "Data Source=(localdb)\\MSSQLLocalDB;Initial Catalog=squ;Integrated Security=True;Pooling=False" MicroSoft.EntityFrameWorkCore.SqlServer -OutputDir Data

Using Configuration Manager

- Database path can be maintained in appsetting.json file

```
{
 "Logging": { "LogLevel": { "Default": "Information",
                  "Microsoft.AspNetCore": "Warning"}},
 "AllowedHosts": "*",
 "ConnectionStrings": {
  "DB1": "Data Source=(localdb)\\MSSQLLocalDB;Initial Catalog=squ;Integrated……"
 }
}
```

- Use the Scaffold statement from the configuration manager as follows

scaffold-dbcontext -connection name=DB1
MicroSoft.EntityFrameWorkCore.SqlServer -OutputDir Data

- From C# and Web Pages using access to connection string from the appsetting.json as follows

```
var builder = WebApplication.CreateBuilder();
var app = builder.Build();
string db1 = app.Configuration.GetConnectionString("DB1");
```

DBContext Exception (VS 2022)

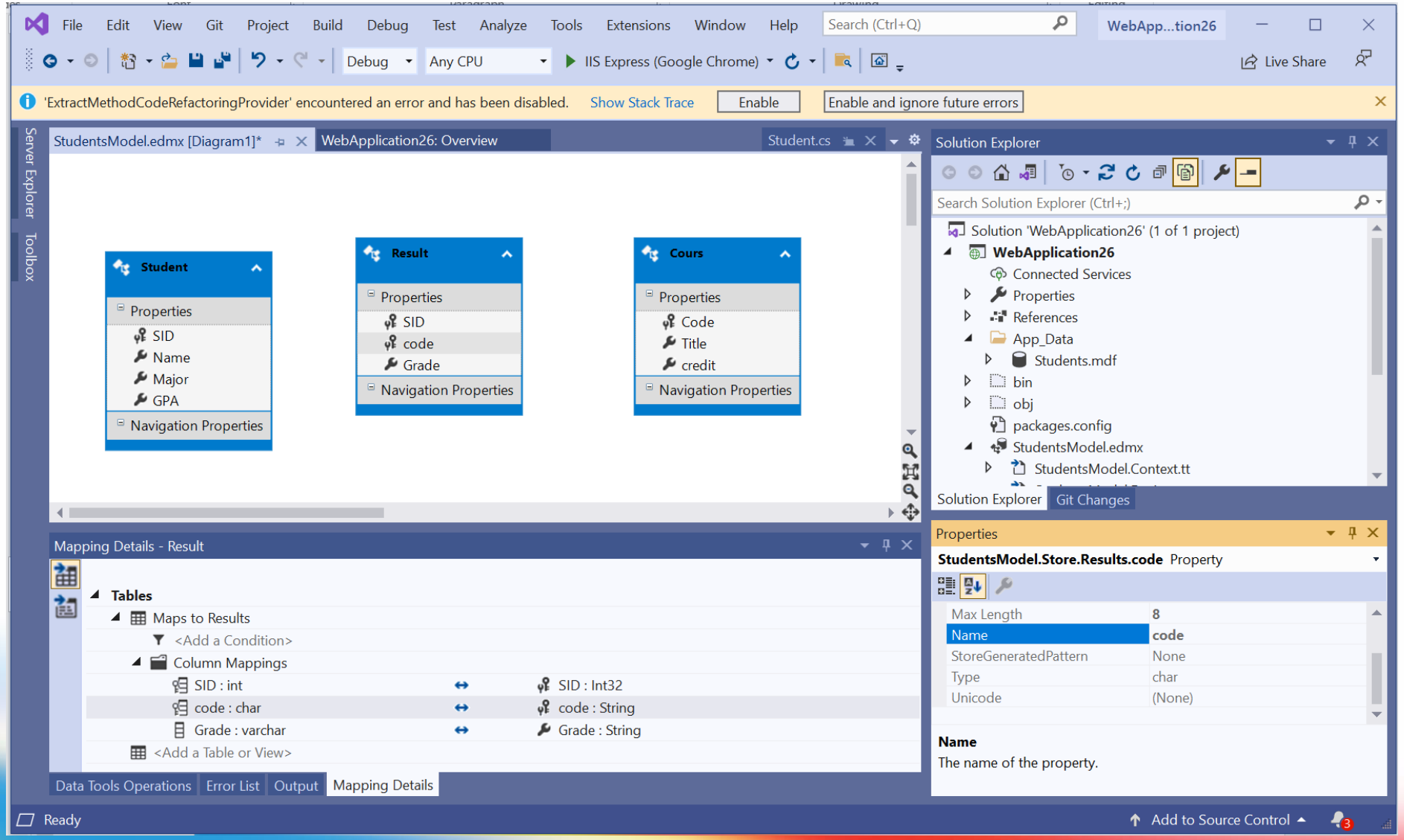
- Exception in the DB connection string "name=DB1” in the following statement

```
protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)
    => optionsBuilder.UseSqlServer("name=DB1");
```

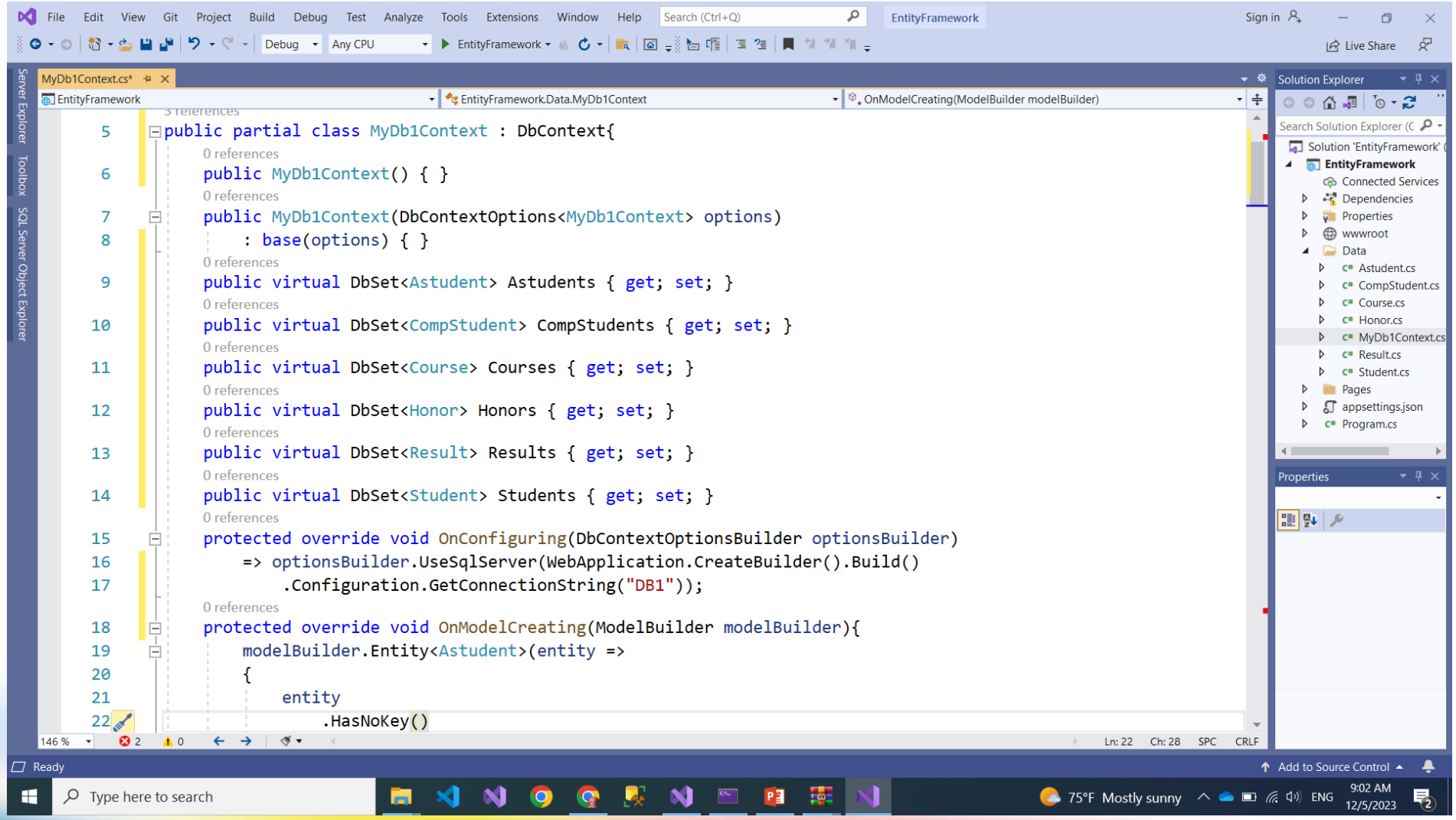
- Replace with With the following statement

```
protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)
    => optionsBuilder.UseSqlServer(
        WebApplication.CreateBuilder().Configuration.GetConnectionString("DB1")
);
```

Entity Data Model (2019) for Students Database



Entity Data Model (2022)



Mapping Classes

- Class is generated for each DB table

```
public partial class Course {
    public string Code { get; set; } = null!;
    public string? Title { get; set; }
    public int? Credit { get; set; }
}
public partial class Result {
    public int Sid { get; set; }
    public string Code { get; set; } = null!;
    public string? Grade { get; set; }
}
public partial class Student {
    public int Sid { get; set; }
    public string? Sname { get; set; }
    public DateTime? Bdate { get; set; }
    public double? Gpa { get; set; }
    public string? Major { get; set; }
}
```

Accessing DB using Linq and Entity Data Framework

- Create a Web page with a Label element and use the Entity model to access to DB as follows:

```
MyDb1Context en = new MyDb1Context();

var w= from s in en.Students
       where s.SID >=100
       select s.Name;

ViewData["message"] = "";
foreach (string s in w)
       ViewData["message"] += $"{s}<br/>";
```

Example

- Display student names, enrolled courses and grades for all students.

```
MyDb1Context en = new MyDb1Context();
var w= from x in en.Students
       from c in en.Courses
       from r in en.Results
       where x.SID >=100 && x.SID==r.SID && c.Code==r.code
       select new { x.Name, c.Title, r.Grade};

ViewData["message"] = "";
foreach (var s in w)
       ViewData["message"] += $"{s.Name}, {s.Title},
{s.Grade}<br/>";
```

Adding new Data

- Create new object as class then add it to the database
- Example

```
//adding new student
MyDb1Context en = new MyDb1Context();
//Create new object from the table class
Student t = new Student {SID = 919, Name = “Khamis",
                         Major ="CS", GPA=4};
en.Students.Add(t); //Add new student to DB entity model
en.SaveChanges(); //Save in DB
```

Deleting data from Database

- Remove student with SID=919 from entity model then save it to the database

```
MyDb1Context en = new MyDb1Context();
//find student with SID 911
Student ss = (from s in en.Students
              where s.SID == 919
              select s).FirstOrDefault();
en.Students.Remove(ss); //remove student from list
en.SaveChanges();  //Save in DB
```

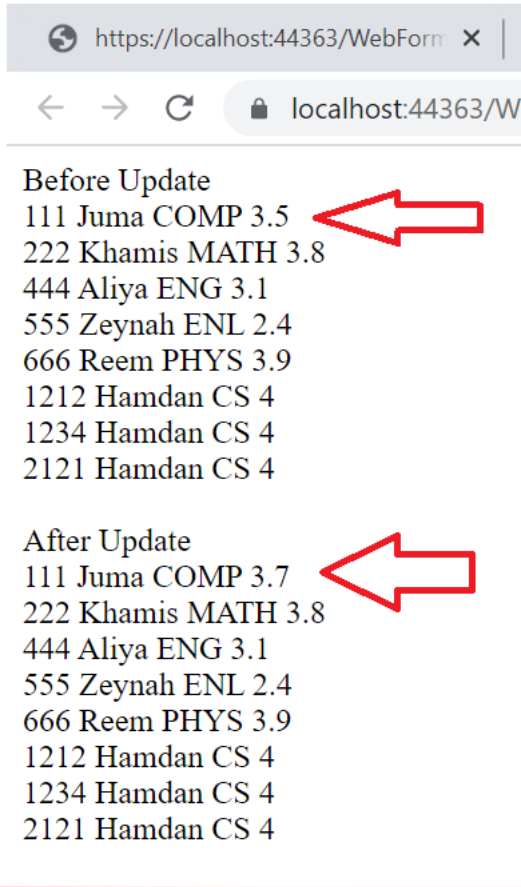
Updating information in Database using Entity Data Model

- Add 0.2 to GPA for the student with SID of 111

```
MyDb1Context en = new MyDb1Context();
ViewData["message"] = "Before Update <br/>";
foreach (var s in en.Students)
   ViewData["message"] += $"{s.SID} {s.Name} {s.Major} {s.GPA}<
```

```
//find the student with SID of 111
var ss = (from mm in en.Students
          where mm.SID == 111
          select mm).FirstOrDefault();

//Update and save student information in DB
ss.GPA += 0.2;
en.SaveChanges();
```

```
ViewData["message"] += "<br/>After Update <br/>";
foreach (var s in en.Students)
    ViewData["message"] += $"{s.SID} {s.Name} {s.Major} {s.GPA}
```

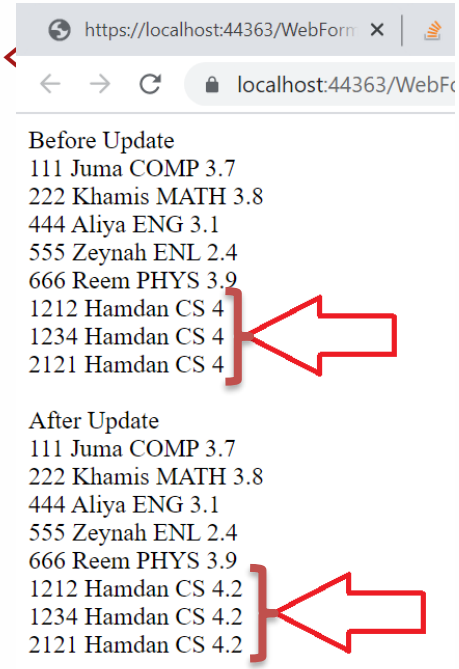


Updating more than one row in Database using Entity Data Model

- Add 0.2 to GPA for CS students

```
MyDb1Context en = new MyDb1Context();
ViewData["message"] = "Before Update <br/>";
foreach (var s in en.Students)
   ViewData["message"] += $"{s.SID} {s.Name} {s.Major} {s.GPA}<
```

```
//find all students from CS major as List
var ss = (from mm in en.Students
          where mm.Major=="CS"
          select mm).ToList();
//Update and save CS student information in DB
foreach (Student x in ss) x.GPA += 0.2;
en.SaveChanges();
```




```
ViewData["message"] += "<br/>After Update <br/>";
foreach (var s in en.Students)
    ViewData["message"] += $"{s.SID} {s.Name} {s.Major} {s.GPA}<br/>";
```

More Examples

- Retrieve COMP students sorted by sid

```
//Entity Framework
MyDb1Context db = new MyDb1Context();

var COMP = from s in db.Students
           where s.Major=="COMP"
           orderby s.Sid
           select s;
//Display the students
foreach(Student s in COMP)
{
    <p>Sid=@s.Sid name=@s.Sname GPA=@s.Gpa Major=@s.Major</p>
}
```

More Examples

- Count number of students per each grade ordered by the grade

```
MyDb1Context db = new MyDb1Context();
var R = from s in db.Results
        group s by s.Grade into g
        orderby g.Key
        select new { Grade=g.Key, noS=g.Count() };

foreach(var g in R)
{
    <p>Grade=@g.Grade No of Students=@g.noS</p>
}
```

More Examples

- Retrieve the student name, course title, and grade for all students and their enrollments

```
MyDb1Context db = new MyDb1Context();
var R_Full = from s in db.Students
             from c in db.Courses
             from r in db.Results
             orderby s.Sname, c.Title
             where s.Sid == r.Sid && r.Code == c.Code
             select new { s.Sname, c.Title, r.Grade };

foreach(var x in R_Full)
{
    <p>student=@x.Sname, Course=@x.Title, Grade=@x.Grade</p>
}
```

More Examples

- Adding a new student to data (using the current number of students +2

```
MyDb1Context db = new MyDb1Context();
int n = db.Students.Count()+2;
//We need to get student info from a form
Student newS = new Student {Sid=n,Gpa=3,
          Sname=n.ToString(), Major="COMP"};
db.Students.Add(newS);
db.SaveChanges();
```

More Examples

- Update student GPA by adding 0.1 in the GPA for COMP students with GPA < 3.5

```
MyDb1Context db = new MyDb1Context();
var updateGPA = from s in db.Students
                where s.Major == "COMP" && s.Gpa<3.5
                select s;

foreach (Student x in updateGPA) x.Gpa += 0.1;
db.SaveChanges();
```

More Examples

- Delete one student with sid of 666

```
MyDb1Context db = new MyDb1Context();
var delCOMP = (from x in db.Students
               where x.Sid == 666
               select x).FirstOrDefault();

if(delCOMP!=null) db.Students.Remove(delCOMP);
db.SaveChanges();
```

More Examples

- Delete all COMP students

```
MyDb1Context db = new MyDb1Context();
var CS = from s in db.Students
         where s.Major == "COMP"
         select s;

 if (CS != null) db.Students.RemoveRange(CS);
 db.SaveChanges();
```