**LINQ(**Language Integrated Query)

<http://dotnetpattern.com/top-linq-interview-questions>

<http://www.dotnetfunda.com/interviews/cat/110/linq>

<http://dotnetpattern.com/linq-query-operators>

1. LINQ
2. LINQ Namespace
3. First or firstordefault
4. Joins
5. Single or SingleORDefault
6. 3rd Highest salary
7. GroupBy
8. Left outer Joins
9. Contains/start with
10. Ienumerable vs Iquerable
11. What are Anonymous Types?
12. What is Anonymous function?
13. What is Lambda Expression?
14. What is Action in LINQ?
15. What is Predicate delegate in LINQ?
16. What are Func delegates in LINQ?
17. What is LINQ Deferred Execution?

* System.Data.Dlinq.dll provides the functionality to work with LINQ to SQL

**1.** using System.Linq;

LINQ is introduced in .NET 3.5. Before LINQ, developers need to use different languages for retrieving and saving different data sources. For example if you want to retrieve data from SQL or Oracle database you must learn some basics of SQL query language, and for retrieving data from XML files you need to learn XML parsers.

LINQ provides an unified query language for query different data sources. You can retrieve and save data into SQL/Oracle database with exactly same code. In addition LINQ provides on extension methods which helps us to write in-line queries on typed object to filter, group, join results.

LINQ variations

* LINQ to objects
* LINQ to SQL
* LINQ to XML
* LINQ to Entities

LINQ Advantages

* LINQ helps in writing faster queries which help in faster development.
* LINQ queries are easier to debug.
* Allow same query syntax for different data sources.
* Compile-time type checking and dynamic query composition.
* LINQ is extensible allow you to create query providers for any new data source.

**Operator List Linq:**

| **Operator Category** | **LINQ Query Operators Names** |
| --- | --- |
| Filtering | Where, OfType |
| Sorting | OrderBy, OrderByDescending, ThenBy, ThenByDescending |
| Set | Except, Intersect, Union, Distinct |
| Quantifier | All, Any, Contains |
| Projection | Select, SelectMany |
| Partitioning | Skip, SkipWhile, Take, TakeWhile |
| Join | Join, GroupJoin |
| Grouping | GroupBy, ToLookup |
| Sequencing | DefaultIfEmpty, Empty, Range, Repeat |
| Equality | SequenceEqual |
| Element | ElementAt, ElementAtOrDefault, First, FirstOrDefault, Last, LastOrDefault, Single, SingleOrDefault |
| Conversion | AsEnumerable, AsQueryable, Cast, OfType, ToArray, ToDictionary, ToList, ToLookup |
| Concatenation | Concat |
| Aggregation | Aggregate, Average, Count, LongCount, Max, Min, Sum |

**Q. First** = Return First element of sequence and throws exception(*InvalidOperationException)* when there is no elements present in table

Example: var result = (from a in obj.Employesses where a.NAME == "sujeet" select a) .First();

=>(from a in obj.Employesses where a.NAME == "tt" select a) .First(); =>it will throw exception because there is no elements in the list with name “tt”.

**FirstORDefault():**returns First Element Of Sequence and does not throws Exception when There IS No element Present in Table and it return null irrespective of data type.

var result = (from a in obj.Employesses where a.ID ==50000 select a) .FirstOrDefault();

**\*Single or SingleORDefault**

**Single :**

* 1. Returns single element in a sequence or element that satisfy a condition. If asequence has more than one elements or If no elements found then throws InvalidOperationException exception.

EmpModel obj = new EmpModel();

var result = (from a in obj.Employesses where a.age == 20 select a).Single();

Console.WriteLine(result.ID);

**SingleORDefault : 1.** Returns single element in a sequence or element that satisfy a condition.and If no elements found then return default value null but If a sequence has more than one elements then throw exception InvalidOperationException :

EmpModel obj = new EmpModel();

var result = (from a in obj.Employesses where a.age == 20 select a).SingleOrDefault();

Console.WriteLine(result.ID);

Q. Nth Highest salary:

1. 2nd highest : EmpModel obj = new EmpModel();

var result = (from a in obj.Employesses select a).OrderByDescending(e => e.salary).Skip(1).First();

If multiple employees may have equal salary and you wish to return an IEnumerable of all the employees with the second-highest salary you could do.

EmpModel obj = new EmpModel();

var result = (from temp in obj.Employesses select temp).GroupBy(e => e.salary).OrderByDescending(g => g.Key).Skip(2).First();

**here g.Key is system defined.**

**Joins:**

What are Anonymous Types=>

Anonymous types are types that are generated by compiler at run time. When we create a anonymous type we do not specify a name. We just write properties names and their values. Compiler at runtime create these properties and assign values to them.

Below is the example of Anonymous type:

|  |  |
| --- | --- |
| 1  2 | var k = new { FirstProperty = "value1", SecondProperty = "value2" };  Console.WriteLine(k.FirstProperty); |

Anonymous class is useful in LINQ queries to save our intermediate results.

There are some restrictions on Anonymous types.

Anonymous types cannot implement interfaces.

* Anonymous types can not specify any methods.
* We can not define static members.
* All defined properties must be initialized.
* We can only define public fields.
* Anonymous type is a reference type and all the properties are read-only.
* The implicitly typed variable- var, is used to hold an anonymous type.
* Anonymous type can be defined using the new keyword and object initializer syntax.

***What is Anonymous function****: A*n Anonymous function is a special function which does not have any name. We just define their parameters and define the code into the curly braces.

*It has been introduced in c# 2.0*

* Anonymous method is a block of code, which is used as a parameter for the delegate.
* An anonymous method can be used anywhere. A delegate is used and is defined in line, without a method name with the optional parameters and a method body.
* The scope of the parameters of an anonymous method is the anonymous-method-block.
* An anonymous method can use generic parameter types like any other method.

delegate int func(int a, int b);

static void Main(string[] args)

{

func f1 = delegate (int a, int b)

{

return a + b;

};

Console.WriteLine(f1(10, 20));

Console.ReadLine();

**What is Lambda Expression= >**

A lambda expression is an [anonymous function](https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/statements-expressions-operators/anonymous-methods) that you can use to create [delegates](https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/delegates/using-delegates) or [expression tree](https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/concepts/expression-trees/index) types. By using lambda expressions, you can write local functions that can be passed as arguments or returned as the value of function calls.

Lambda Expression is a shortcut way to writing delegates. By using Lambda Expression we can write inline functions that can be used to pass as arguments to a function or returned as value from functions.

To create a lambda expression, you specify input parameters (if any) on the left side of the lambda operator [=>](https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/operators/lambda-operator), and you put the expression or statement block on the other side. For example, the lambda expression x => x \* x specifies a parameter that’s named x and returns the value of x squared. You can assign this expression to a delegate type, as the following example shows:

delegate int del(int i);

static void Main(string[] args)

{

del myDelegate = x => x \* x;

int j = myDelegate(5); //j = 25

}

For multiple parameter

delegate int func(int a, int b);

static void Main(string[] args)

{

    func f1 = (a, b) => a + b;

    Console.WriteLine(f1(1, 2));

}

**For expression tree example**:

using System.Linq.Expressions;

namespace ConsoleApplication1

{

class Program

{

static void Main(string[] args)

{

Expression<del> myET = x => x \* x;

}

}

}

For Empty parameters:

() => SomeMethod()

Input with datatype:

(int x, string s) => s.Length > x

***Func Delegate:*****Func** is a delegate (pointer) to a method, that takes zero, one or more input parameters, and returns a value (or reference). The last parameter is always a return type.

Func<int, int, int> obj = (a, b) => { return a + b; };

Console.WriteLine(obj(10, 1000));

Example2:

Func<string, bool> isStringStartWithMChar = (a) => a.StartsWith("m");

Console.WriteLine(isStringStartWithMChar("msmd"))

**Action** is a delegate (pointer) to a method, that takes zero, one or more input parameters, but does not return anything. We can specify upto 16 parameters.

Action<int, int> obj = (a, b) => Console.WriteLine(a + b);

obj(100, 100);

//or

Action<int ,int> obj1 = delegate (int n1, int n2)

{

Console.WriteLine(n1 + n2);

};

obj1(200, 200);

**Predicate delegate :** A predicate delegate methods must take one input parameter and it then returns a boolean value - true or false.

Predicate is a delegated provided by base class library of NET. In Predicate delegate we can only store those method which have one input parameter and a bool return type.

Predicate delegates are mainly used in filtering scenarios in LINQ where we have to filter some list. Predicate only pass one input parameter and not need to pass return type because by default it returns boolenan

Predicate<string> isStringStartWithMChar = (a) => a.StartsWith("m");

Console.WriteLine(isStringStartWithMChar("msmd"));

**What is LINQ Deferred Execution :**

In deferred execution, a LINQ query is not executed when we declare our query. Query is executed when query variable is executed in a loop like for, foreach.

EmpModel empobj = new EmpModel();

var emp = (from a in empobj.Employesses select a);

foreach(var a in emp)

{

Console.WriteLine(a.NAME);

}

Console.ReadLine();

Deferred Execution returns the Latest Data

# Immediate Execution of LINQ Query

Immediate execution is the reverse of deferred execution. It forces the LINQ query to execute and gets the result immediately. The 'To' [conversion operators](http://www.tutorialsteacher.com/linq/linq-conversion-operators) execute the given query and give the result immediately.

Query Syntax doesn't support 'To' operators but can use ToList(), ToArray() or ToDictionary() for immediate execution as below:

EmpModel empobj = new EmpModel();

var emp = (from a in empobj.Employesses select a).ToList(); ;

foreach(var a in emp)

{

Console.WriteLine(a.NAME);

}

Console.ReadLine();

**let keyword :** It projects a new range variable, allows re-use of the expression and makes the query more readable.

IList<Student> studentList = new List<Student>() {

new Student() { StudentID = 1, StudentName = "John", Age = 18 } ,

new Student() { StudentID = 2, StudentName = "Steve", Age = 21 } ,

new Student() { StudentID = 3, StudentName = "Bill", Age = 18 } ,

new Student() { StudentID = 4, StudentName = "Ram" , Age = 20 } ,

new Student() { StudentID = 5, StudentName = "Ron" , Age = 21 }

};

var lowercaseStudentNames = from s in studentList

where s.StudentName.ToLower().StartsWith("r")

select s.StudentName.ToLower();

As you can see, the ToLower() method is used multiple times in the above query. The following example use 'let' to introduce new variable 'lowercaseStudentName' that will be then used in every where. Thus, **let** keyword to make the query more readable.

var lowercaseStudentNames = from s in studentList

let lowercaseStudentName = s.StudentName.ToLower()

where lowercaseStudentName.StartsWith("r")

select lowercaseStudentName;

foreach (var name in lowercaseStudentNames)

# Console.WriteLine(name)

Into keyword: You can also use the 'into' keyword to continue a query after a **select** clause and grouping also.

public class Student

{

public int StudentID { get; set; }

public string StudentName { get; set; }

public int Age { get; set; }

}

class Program

{

delegate int funcdel(int a, int b);

static void Main(string[] args)

{

IList<Student> studentList = new List<Student>()

{

new Student() { StudentID = 1, StudentName = "John", Age = 13} ,

new Student() { StudentID = 2, StudentName = "Moin", Age = 21 } ,

new Student() { StudentID = 3, StudentName = "Bill", Age = 18 } ,

new Student() { StudentID = 4, StudentName = "Ram" , Age = 20} ,

new Student() { StudentID = 5, StudentName = "Ron" , Age = 15 }

};

//var teenAgerStudents = from s in studentList where s.Age > 12 && s.Age < 20 && s.StudentName.StartsWith("B") select s;

var teenAgerStudents = from s in studentList where s.Age > 12 && s.Age < 20 select s into teenStudents where teenStudents.StudentName.StartsWith("B")

select teenStudents;

foreach (Student std in teenAgerStudents)

{

Console.WriteLine(std.StudentName);

}

Console.ReadLine();

}

**Joins:**

EmpModel obj = new Model.EmpModel();

var output = (from a in obj.Employesses

join b in obj.Depts on a.deptit equals b.deptIT

select new

{

id = a.ID,

name = a.NAME,

deptid = b.deptIT

}).ToList(); ;

foreach (var a in output)

{

Console.WriteLine("{0}{1}{2}", a.id, a.name, a.deptid);

}

**LEFT outer JOIN :** returns all records or rows from left table and from right table returns only matched records. If there are no columns matching in the right table, it returns NULL values.

In LINQ to achieve LEFT JOIN behavior, it is mandatory to use "INTO" keyword and "DefaultIfEmpty()" method. We can apply LEFT JOIN in LINQ like as

EmpModel obj = new Model.EmpModel();

var output = (from a in obj.Employesses

join b in obj.Depts on a.deptit equals b.deptIT

into t from rt in t.DefaultIfEmpty()

select new

{

a.ID,

a.NAME,

a.age,

a.deptit,

deptIT= (int?) rt.deptIT,

rt.deptname

}).ToList();

foreach (var a in output)

{

Console.WriteLine("{0}{1}{2}{3}{4}", a.age, a.ID, a.deptit,a.deptIT,a.deptname);

}

Note: holding value in t and then right table value holding value in rt.

Group By :

EmpModel obj = new Model.EmpModel();

var output = obj.Employesses.GroupBy(e => e.NAME)

.Select (g => new

{

name=g.Key,

salarymax=g.Sum(d=>d.salary)

}).ToList();

foreach (var a in output)

{

Console.WriteLine("{0}{1}",a.salarymax,a.name);

}

Note: g.Key means name columns because applied group by name column name.

Cross join : EmpModel obj = new Model.EmpModel();

var output = (from a in obj.Employesses from b in obj.Depts

select new

{

a.NAME,

b.deptname

}).ToList();

Console.WriteLine(output.Count());

**Output : 10\*8 = 80 ans**

**IEnumerable vs IQueryable :**

**IQueryable**

I .Queryable exists in the System.Linq Namespace.

IQueryable is suitable for querying data from out-memory (like remote database, service) collections.

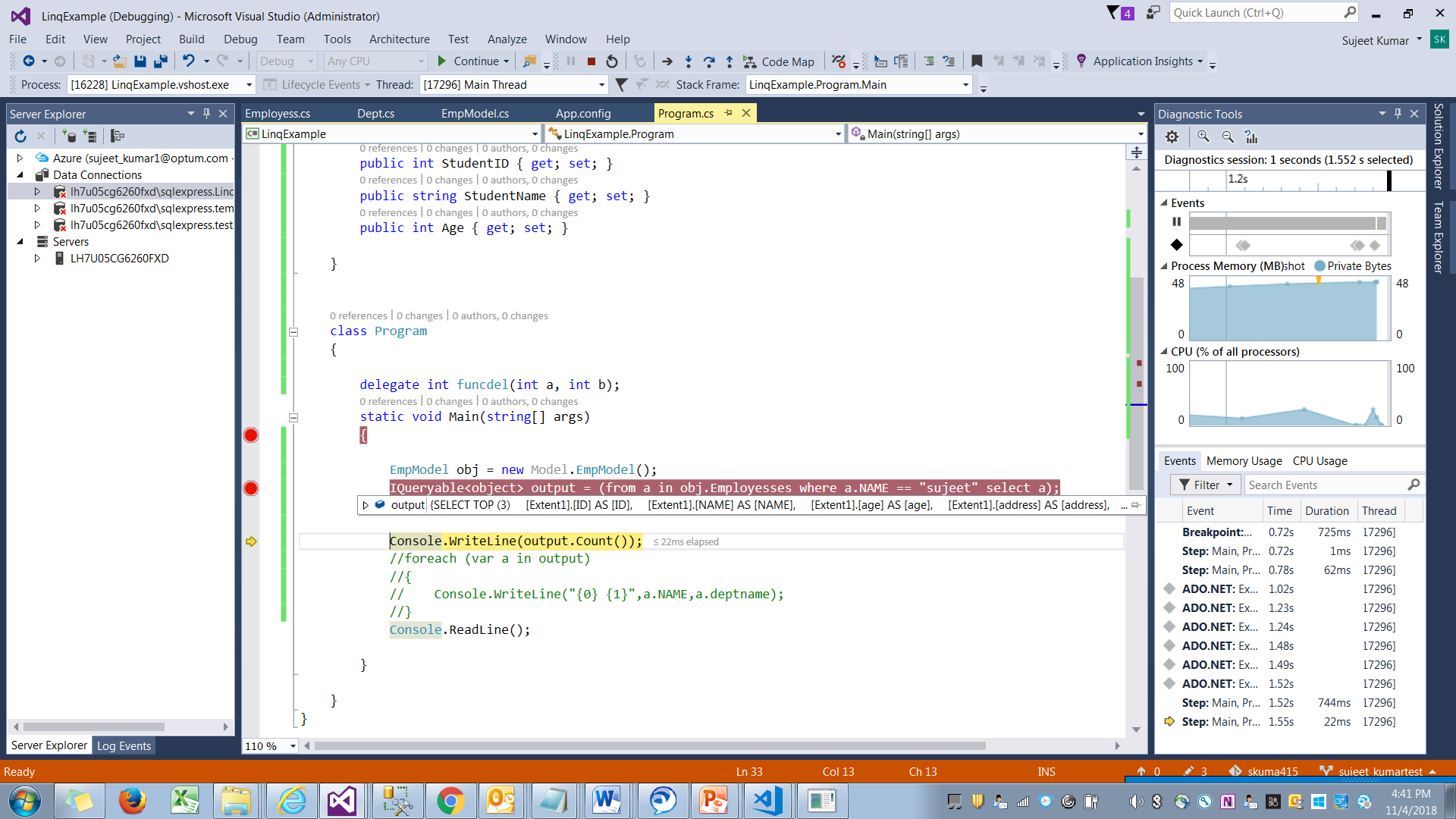
While querying data from a database, IQueryable executes a "select query" on server-side with all filters.

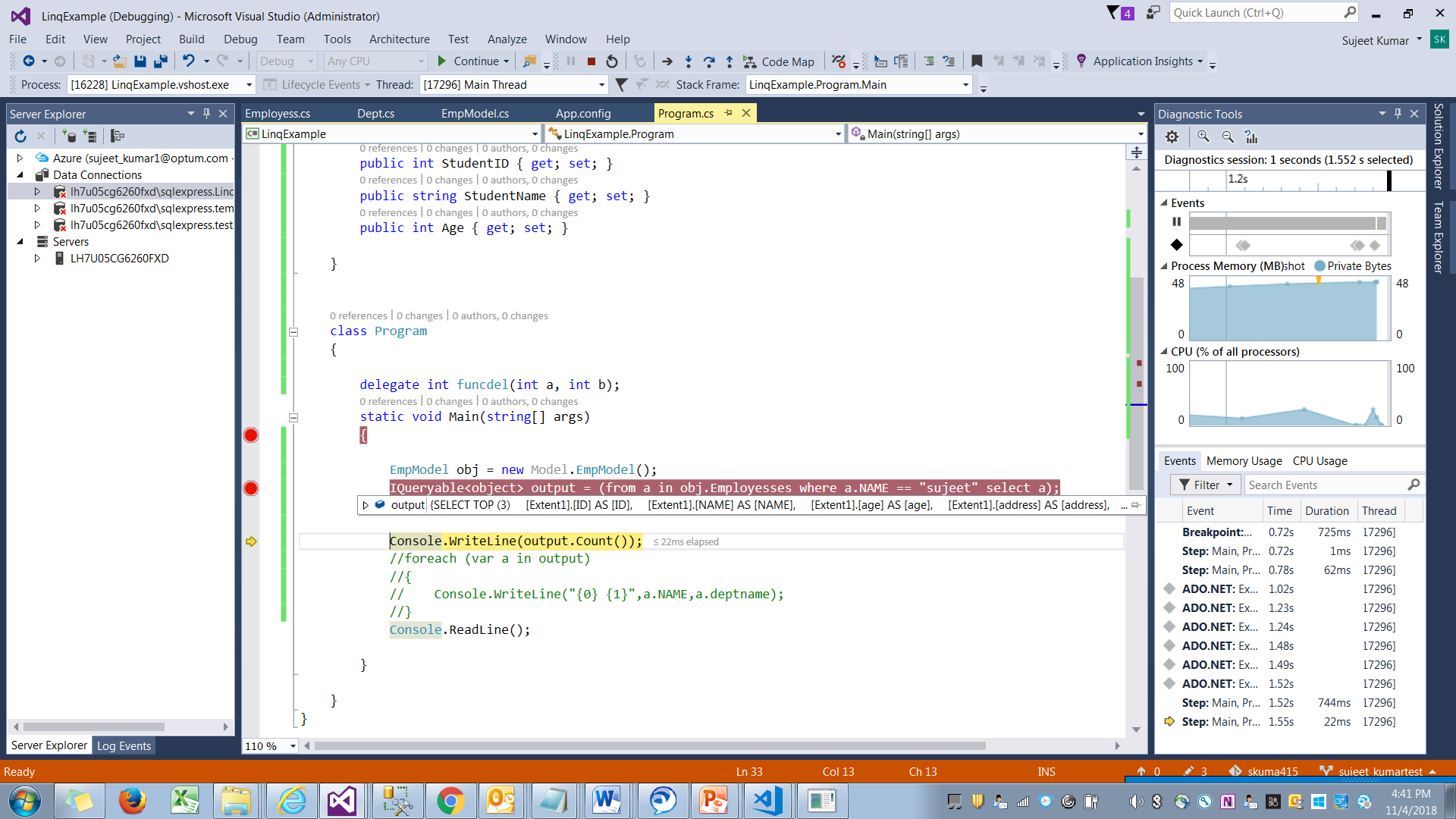
IQueryable is beneficial for LINQ to SQL queries.

EmpModel obj = new Model.EmpModel();

IQueryable<object> output = (from a in obj.Employesses where a.NAME == "sujeet" select a);

output = output.Take<object>(3);





**IEnumerable**

1. IEnumerable exists in the System.Collections namespace.
2. IEnumerable is suitable for querying data from in-memory collections like List, Array and so on.
3. While querying data from the database, IEnumerable executes "select query" on the server-side, loads data in-memory on the client-side and then filters the data.

EmpModel obj = new Model.EmpModel();

IEnumerable<object> output = (from a in obj.Employesses where a.NAME == "sujeet" select a);

output = output.Take<object>(3);

**x**