

# LINQ in practice



# Lesson Objectives

- <Mandatory slide>
- <Brief the objectives of the course: What trainees learn after the course>
- <Remember to update the course version in Notes part>

## Section 1


# STANDARD QUERY OPERATORS

# Standard Query Operators 1-2

- Standard Query Operators in LINQ are actually extension methods for the `IEnumerable<T>` and `IQueryable<T>` types.
- They are defined in the `System.Linq.Enumerable` and `System.Linq.Queryable` classes.
- There are over 50 standard query operators available in LINQ that provide different functionalities like filtering, sorting, grouping, aggregation, concatenation, etc.
- Standard Query Operators in Query Syntax

```
var students = from s in studentList
                where s.age > 20
                select s;
```

Standard Query Operators



- Standard Query Operators in Method Syntax

```
var students = studentList.Where(s => s.age > 20).ToList<Student>();
```

Extension Methods



# Standard Query Operators 2-2

- Standard Query Operators can be classified based on the functionality they provide. The following table lists all the classification of Standard Query Operators:

Classification	Standard Query Operators
Filtering	Where, OfType
Sorting	OrderBy, OrderByDescending, ThenBy, ThenByDescending, Reverse
Grouping	GroupBy, ToLookup
Join	GroupJoin, Join
Projection	Select, SelectMany
Aggregation	Aggregate, Average, Count, LongCount, Max, Min, Sum
Quantifiers	All, Any, Contains
Elements	ElementAt, ElementAtOrDefault, First, FirstOrDefault, Last, LastOrDefault, Single, SingleOrDefault
Set	Distinct, Except, Intersect, Union
Partitioning	Skip, SkipWhile, Take, TakeWhile
Concatenation	Concat
Equality	SequenceEqual
Generation	DefaultEmpty, Empty, Range, Repeat
Conversion	AsEnumerable, AsQueryable, Cast, ToArray, ToDictionary, ToList

# Filtering Operators – Where 1- 2

- Filtering operators in LINQ filter the sequence (collection) based on some given criteria.
- **Where**: Returns values from the collection based on a predicate function
- The **Where** extension method has following two overloads. Both overload methods accepts a Func delegate type parameter.

## Where method Overloads:

```
public static IEnumerable<TSource> Where<TSource>(this IEnumerable<TSource> source,  
                                                Func<TSource, bool> predicate);  
  
public static IEnumerable<TSource> Where<TSource>(this IEnumerable<TSource> source,  
                                                Func<TSource, int, bool> predicate);
```

# Filtering Operators – Where 2- 2

- Example: Where clause - LINQ query syntax C#.

```
ICollection<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 filteredResult = from s in studentList  
    where s.Age > 12 && s.Age < 20  
    select s.StudentName;
```

- Where in method syntax in C#

```
var filteredResult = studentList.Where(s => s.Age > 12 && s.Age < 20);
```

# Filtering Operator - OfType

- The OfType operator filters the collection based on the ability to cast an element in a collection to a specified type.
- OfType in Query Syntax

```
ICollection mixedList = new ArrayList();  
mixedList.Add(0);  
mixedList.Add("One");  
mixedList.Add("Two");  
mixedList.Add(3);  
mixedList.Add(new Student() { StudentId = 1, StudentName = "Bill" });  
  
var stringResult = from s in mixedList.OfType<string>()  
select s;  
  
var intResult = from s in mixedList.OfType<int>()  
select s;
```

- OfType in Method Syntax

```
var stringResult = mixedList.OfType<string>();
```



- A sorting operator arranges the elements of the collection in ascending or descending order. LINQ includes following sorting operators.

Sorting Operator	Description
<a href="#">OrderBy</a>	Sorts the elements in the collection based on specified fields in ascending or descending order.
<a href="#">OrderByDescending</a>	Sorts the collection based on specified fields in descending order. Only valid in method syntax.
<a href="#">ThenBy</a>	Only valid in method syntax. Used for second level sorting in ascending order.
<a href="#">ThenByDescending</a>	Only valid in method syntax. Used for second level sorting in descending order.
<a href="#">Reverse</a>	Only valid in method syntax. Sorts the collection in reverse order.

# Sorting Operators: OrderBy

- OrderBy in Query Syntax:

```
ICollection<Student> studentList = new List<Student>() {  
    new Student() { StudentId = 1, StudentName = "John", Age = 18 },  
    new Student() { StudentId = 2, StudentName = "Steve", Age = 15 },  
    new Student() { StudentId = 3, StudentName = "Bill", Age = 25 },  
    new Student() { StudentId = 4, StudentName = "Ram", Age = 20 },  
    new Student() { StudentId = 5, StudentName = "Ron", Age = 19 }  
};  
  
var orderByResult = from s in studentList  
    orderby s.StudentName  
    select s;  
  
var orderByDescendingResult = from s in studentList  
    orderby s.StudentName descending  
    select s;
```

- OrderBy in Method Syntax:

```
var studentsInAscOrder = studentList.OrderBy(s => s.StudentName);  
var studentsInDescOrder = studentList.OrderByDescending(s => s.StudentName);
```

# Sorting Operators: ThenBy & ThenByDescending

- Multiple sorting in method syntax is supported by using ThenBy and ThenByDescending extension methods.

```

IList<Student> studentList = new List<Student>() {
    new Student() { StudentId = 1, StudentName = "John", Age = 18 },
    new Student() { StudentId = 2, StudentName = "Steve", Age = 15 },
    new Student() { StudentId = 3, StudentName = "Bill", Age = 25 },
    new Student() { StudentId = 4, StudentName = "Ram", Age = 20 },
    new Student() { StudentId = 5, StudentName = "Ron", Age = 19 }
};

var thenByResult = studentList.OrderBy(s => s.StudentName).ThenBy(s => s.Age);

var thenByDescResult = studentList.OrderBy(s => s.StudentName).ThenByDescending(s => s.Age);

```

# Grouping Operators: GroupBy & ToLookup

- The grouping operators do the same thing as the GroupBy clause of SQL query.
- This group is contained in a special type of collection that implements an `IGrouping<TKey,TSource>` interface where TKey is a key value, on which the group has been formed and TSource is the collection of elements that matches with the grouping key value.

Grouping Operators	Description
<a href="#">GroupBy</a>	The GroupBy operator returns groups of elements based on some key value. Each group is represented by <code>IGrouping&lt;TKey, TElement&gt;</code> object.
<a href="#">ToLookup</a>	ToLookup is the same as GroupBy; the only difference is the execution of GroupBy is deferred whereas ToLookup execution is immediate.

# Grouping Operators: GroupBy 1 - 2

- GroupBy in Query Syntax

```
ICollection<Student> studentList = new List<Student>() {  
    new Student() { StudentId = 1, StudentName = "John", Age = 18 },  
    new Student() { StudentId = 2, StudentName = "Steve", Age = 15 },  
    new Student() { StudentId = 3, StudentName = "Bill", Age = 25 },  
    new Student() { StudentId = 4, StudentName = "Ram", Age = 20 },  
    new Student() { StudentId = 5, StudentName = "Ron", Age = 19 }  
};  
  
var groupedResult = from s in studentList  
    group s by s.Age;  
  
//iterate each group  
foreach (var ageGroup in groupedResult)  
{  
    Console.WriteLine("Age Group: {0}", ageGroup.Key); //Each group has a key  
  
    foreach (Student s in ageGroup) // Each group has inner collection  
        Console.WriteLine("Student Name: {0}", s.StudentName);  
}
```

- GroupBy in Method Syntax

```
ICollection<Student> studentList = new List<Student>() {  
    new Student() { StudentId = 1, StudentName = "John", Age = 18 },  
    new Student() { StudentId = 2, StudentName = "Steve", Age = 15 },  
    new Student() { StudentId = 3, StudentName = "Bill", Age = 25 },  
    new Student() { StudentId = 4, StudentName = "Ram", Age = 20 },  
    new Student() { StudentId = 5, StudentName = "Ron", Age = 19 }  
};  
  
var groupedResult = studentList.GroupBy(s => s.Age);  
  
foreach (var ageGroup in groupedResult)  
{  
    Console.WriteLine("Age Group: {0}", ageGroup.Key); //Each group has a key  
  
    foreach (Student s in ageGroup) //Each group has a inner collection  
    {  
        Console.WriteLine("Student Name: {0}", s.StudentName);  
    }  
}
```

# Grouping Operators: ToLookup

- ToLookup is the same as GroupBy; the only difference is GroupBy execution is deferred, whereas ToLookup execution is immediate. Also, ToLookup is only applicable in Method syntax. **ToLookup is not supported in the query syntax.**
- ToLookup in method syntax:

```
ICollection<Student> studentList = new List<Student>() {  
    new Student() { StudentId = 1, StudentName = "John", Age = 18 },  
    new Student() { StudentId = 2, StudentName = "Steve", Age = 15 },  
    new Student() { StudentId = 3, StudentName = "Bill", Age = 25 },  
    new Student() { StudentId = 4, StudentName = "Ram", Age = 20 },  
    new Student() { StudentId = 5, StudentName = "Ron", Age = 19 }  
};  
  
var lookupResult = studentList.ToLookup(s => s.Age);  
  
foreach (var group in lookupResult)  
{  
    Console.WriteLine("Age Group: {0}", group.Key); //Each group has a key  
  
    foreach (Student s in group) //Each group has a inner collection  
    {  
        Console.WriteLine("Student Name: {0}", s.StudentName);  
    }  
}
```

# Joining Operator: Join, GroupJoin

- The joining operators joins the two sequences (collections) and produce a result.

Joining Operators	Usage
Join	The Join operator joins two sequences (collections) based on a key and returns a resulted sequence.
GroupJoin	The GroupJoin operator joins two sequences based on keys and returns groups of sequences. It is like Left Outer Join of SQL.



# Joining Operator: Join 1 - 3

- The Join operator operates on two collections, inner collection & outer collection. It returns a new collection that contains elements from both the collections which satisfies specified expression. It is the same as **inner join** of SQL.

```
IList<string> strList1 = new List<string>() {"One","Two","Three","Four"};  
  
IList<string> strList2 = new List<string>() {"One","Two","Five","Six"};  
  
var innerJoin = strList1.Join(strList2,  
    str1 => str1,  
    str2 => str2,  
    (str1, str2) => str1);
```

# Joining Operator: Join 2 - 3

- Now, let's understand join method using following Student and Standard class where Student class includes StandardId that matches with StandardId of Standard class.

```
public class Student
{
    public int StudentId { get; set; }
    public string StudentName { get; set; }
    public int StandardId { get; set; }
}

public class Standard
{
    public int StandardId { get; set; }
    public string StandardName { get; set; }
}
```

# Joining Operator: Join 3 - 3

- Example: Join Query

```

IList<Student> studentList = new List<Student>() {
    new Student() { StudentId = 1, StudentName = "John", StandardId = 1 },
    new Student() { StudentId = 2, StudentName = "Moin", StandardId = 1 },
    new Student() { StudentId = 3, StudentName = "Bill", StandardId = 2 },
    new Student() { StudentId = 4, StudentName = "Ram", StandardId = 2 },
    new Student() { StudentId = 5, StudentName = "Ron" }
};

IList<Standard> standardList = new List<Standard>() {
    new Standard() { StandardId = 1, StandardName = "Standard 1" },
    new Standard() { StandardId = 2, StandardName = "Standard 2" },
    new Standard() { StandardId = 3, StandardName = "Standard 3" }
};

var innerJoin = studentList.Join(// outer sequence
    standardList, // inner sequence
    student => student.StandardId, // outerKeySelector
    standard => standard.StandardId, // innerKeySelector
    (student, standard) => new // result selector
    {
        StudentName = student.StudentName,
        StandardName = standard.StandardName
    });

```

# Joining Operator: GroupJoin

- The GroupJoin operator performs the same task as Join operator except that GroupJoin returns a result in group based on specified group key. The GroupJoin operator joins two sequences based on key and groups the result by matching key and then returns the collection of grouped result and key.

```
var groupJoin = standardList.GroupJoin(studentList, //inner sequence
    std => std.StandardId, //outerKeySelector
    s => s.StandardId, //innerKeySelector
    (std, studentsGroup) => new // resultSelector
    {
        Students = studentsGroup,
        StandarFullIdName = std.StandardName
    });

foreach (var item in groupJoin)
{
    Console.WriteLine(item.StandarFullIdName);

    foreach (var stud in item.Students)
        Console.WriteLine(stud.StudentName);
}
```

- The Select operator always returns an IEnumerable collection which contains elements based on a transformation function. It is similar to the Select clause of SQL that produces a flat result set.
- LINQ query syntax must end with a **Select** or **GroupBy** clause. The following example demonstrates select operator that returns a string collection of StudentName.
- Select in Query Syntax:

```
ICollection<Student> studentList = new List<Student>() {  
    new Student() { StudentId = 1, StudentName = "John" },  
    new Student() { StudentId = 2, StudentName = "Moin" },  
    new Student() { StudentId = 3, StudentName = "Bill" },  
    new Student() { StudentId = 4, StudentName = "Ram" },  
    new Student() { StudentId = 5, StudentName = "Ron" }  
};  
  
var selectResult = from s in studentList  
    select s.StudentName;
```

- Select in Method Syntax:

```
var selectResult = studentList.Select(s => new {Name = s.StudentName});
```

- The SelectMany operator projects sequences of values that are based on a transform function and then flattens them into one sequence.

```
List<string> phrases = new List<string>() { "an apple a day", "the quick brown fox" };  
  
var query = from phrase in phrases  
            from word in phrase.Split(' ')  
            select word;  
  
foreach (string s in query)  
    Console.WriteLine(s);
```

# Quantifier Operators

- The quantifier operators evaluate elements of the sequence on some condition and return a boolean value to indicate that some or all elements satisfy the condition.

Operator	Description
All	Checks if all the elements in a sequence satisfies the specified condition
Any	Checks if any of the elements in a sequence satisfies the specified condition
Contains	Checks if the sequence contains a specific element

# Quantifier Operators: All, Any

- The All operator evaluates each elements in the given collection on a specified condition and returns True if all the elements satisfy a condition.

```
ICollection<Student> studentList = new List<Student>() {  
    new Student() { StudentId = 1, StudentName = "John", Age = 18 },  
    new Student() { StudentId = 2, StudentName = "Steve", Age = 15 },  
    new Student() { StudentId = 3, StudentName = "Bill", Age = 25 },  
    new Student() { StudentId = 4, StudentName = "Ram", Age = 20 },  
    new Student() { StudentId = 5, StudentName = "Ron", Age = 19 }  
};  
  
// checks whether all the students are teenagers  
bool areAllStudentsTeenAger = studentList.All(s => s.Age > 12 && s.Age < 20);  
  
Console.WriteLine(areAllStudentsTeenAger);
```

- Any checks whether any element satisfy given condition or not? In the following example, Any operation is used to check whether any student is teen ager or not.

```
bool isAnyStudentTeenAger = studentList.Any(s => s.Age > 12 && s.Age < 20);
```



# Quantifier Operator: Contains

- The Contains operator checks whether a specified element exists in the collection or not and returns a boolean.

```
ICollection<int> intList = new List<int>() { 1, 2, 3, 4, 5 };  
bool result = intList.Contains(10); // returns false
```

# Aggregation Operators: Aggregate

- The aggregation operators perform mathematical operations like Average, Aggregate, Count, Max, Min and Sum, on the numeric property of the elements in the collection.

Method	Description
Aggregate	Performs a custom aggregation operation on the values in the collection.
Average	calculates the average of the numeric items in the collection.
Count	Counts the elements in a collection.
LongCount	Counts the elements in a collection.
Max	Finds the largest value in the collection.
Min	Finds the smallest value in the collection.
Sum	Calculates sum of the values in the collection.

- Element operators return a particular element from a sequence (collection).
- The following table lists all the Element operators in LINQ.

Element Operators (Methods)	Description
ElementAt	Returns the element at a specified index in a collection
ElementAtOrDefault	Returns the element at a specified index in a collection or a default value if the index is out of range.
First	Returns the first element of a collection, or the first element that satisfies a condition.
FirstOrDefault	Returns the first element of a collection, or the first element that satisfies a condition. Returns a default value if index is out of range.
Last	Returns the last element of a collection, or the last element that satisfies a condition
LastOrDefault	Returns the last element of a collection, or the last element that satisfies a condition. Returns a default value if no such element exists.
Single	Returns the only element of a collection, or the only element that satisfies a condition.
SingleOrDefault	Returns the only element of a collection, or the only element that satisfies a condition. Returns a default value if no such element exists or the collection does not contain exactly one element.

# Equality Operator: SequenceEqual

- There is only one equality operator: SequenceEqual. The SequenceEqual method checks whether the number of elements, value of each element and order of elements in two collections are equal or not.

```
Student std = new Student() { StudentID = 1, StudentName = "Bill" };

IList<Student> studentList1 = new List<Student>(){ std };

IList<Student> studentList2 = new List<Student>(){ std };

bool isEqual = studentList1.SequenceEqual(studentList2); // returns true

Student std1 = new Student() { StudentID = 1, StudentName = "Bill" };
Student std2 = new Student() { StudentID = 1, StudentName = "Bill" };

IList<Student> studentList3 = new List<Student>(){ std1};

IList<Student> studentList4 = new List<Student>(){ std2 };

isEqual = studentList3.SequenceEqual(studentList4); // returns false
```

# Concatenation Operator: Concat

- The Concat() method appends two sequences of the same type and returns a new sequence (collection).

```
IList<string> collection1 = new List<string>() { "One", "Two", "Three" };  
IList<string> collection2 = new List<string>() { "Five", "Six"};  
  
var collection3 = collection1.Concat(collection2);  
  
foreach (string str in collection3)  
    Console.WriteLine(str);
```

# Generation Operator: DefaultIfEmpty

- The DefaultIfEmpty() method returns a new collection with the default value if the given collection on which DefaultIfEmpty() is invoked is empty.

```
IList<string> emptyList = new List<string>();  
  
var newList1 = emptyList.DefaultIfEmpty();  
var newList2 = emptyList.DefaultIfEmpty("None");  
  
Console.WriteLine("Count: {0}" , newList1.Count());  
Console.WriteLine("Value: {0}" , newList1.ElementAt(0));  
  
Console.WriteLine("Count: {0}" , newList2.Count());  
Console.WriteLine("Value: {0}" , newList2.ElementAt(0));
```

# Generation Operators: Empty, Range, Repeat

- LINQ includes generation operators DefaultIfEmpty, Empty, Range & Repeat.

Method	Description
Empty	Returns an empty collection
Range	Generates collection of IEnumerable<T> type with specified number of elements with sequential values, starting from first element.
Repeat	Generates a collection of IEnumerable<T> type with specified number of elements and each element contains same specified value.

```
var emptyCollection1 = Enumerable.Empty<string>();  
var emptyCollection2 = Enumerable.Empty<Student>();  
  
Console.WriteLine("Count: {0} ", emptyCollection1.Count());  
Console.WriteLine("Type: {0} ", emptyCollection1.GetType().Name );  
  
Console.WriteLine("Count: {0} ", emptyCollection2.Count());  
Console.WriteLine("Type: {0} ", emptyCollection2.GetType().Name );
```

# Set Operator: Distinct 1 - 2

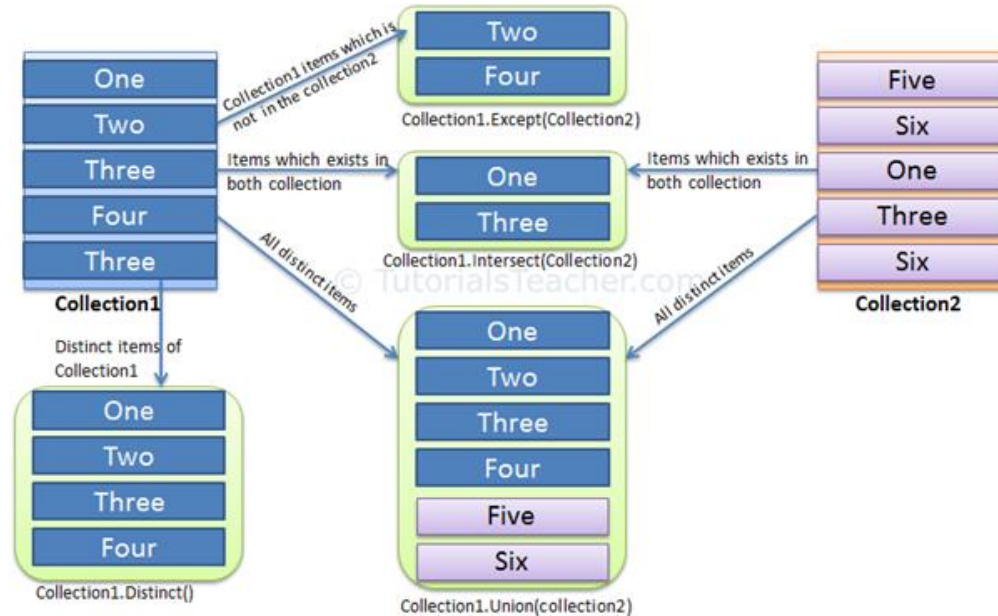
- The following table lists all Set operators available in LINQ.

Set Operators	Usage
<a href="#">Distinct</a>	Returns distinct values from a collection.
<a href="#">Except</a>	Returns the difference between two sequences, which means the elements of one collection that do not appear in the second collection.
<a href="#">Intersect</a>	Returns the intersection of two sequences, which means elements that appear in both the collections.
<a href="#">Union</a>	Returns unique elements from two sequences, which means unique elements that appear in either of the two sequences.



# Set Operator: Distinct 2 - 2

- The following figure shows how each set operators works on the collections:



# Partitioning Operators: Skip & SkipWhile

- Partitioning operators split the sequence (collection) into two parts and return one of the parts.

Method	Description
Skip	Skips elements up to a specified position starting from the first element in a sequence.
SkipWhile	Skips elements based on a condition until an element does not satisfy the condition. If the first element itself doesn't satisfy the condition, it then skips 0 elements and returns all the elements in the sequence.
Take	Takes elements up to a specified position starting from the first element in a sequence.
TakeWhile	Returns elements from the first element until an element does not satisfy the condition. If the first element itself doesn't satisfy the condition then returns an empty collection.

- The Skip() method skips the specified number of element starting from first element and returns rest of the elements.

```
IList<string> strList = new List<string>(){ "One", "Two", "Three", "Four", "Five" };  
  
var newList = strList.Skip(2);  
  
foreach(var str in newList)  
    Console.WriteLine(str);
```

- As the name suggests, the SkipWhile() extension method in LINQ skip elements in the collection till the specified condition is true. It returns a new collection that includes all the remaining elements once the specified condition becomes false for any element.

```
IList<string> strList = new List<string>() {  
    "One",  
    "Two",  
    "Three",  
    "Four",  
    "Five",  
    "Six" };  
  
var resultList = strList.SkipWhile(s => s.Length < 4);  
  
foreach(string str in resultList)  
    Console.WriteLine(str);
```

# Partitioning Operators: Take & TakeWhile

- Partitioning operators split the sequence (collection) into two parts and returns one of the parts.
- The Take() extension method returns the specified number of elements starting from the first element.

```
IList<string> strList = new List<string>(){ "One", "Two", "Three", "Four", "Five" };  
  
var newList = strList.Take(2);  
  
foreach(var str in newList)  
    Console.WriteLine(str);
```

- The TakeWhile() extension method returns elements from the given collection until the specified condition is true. If the first element itself doesn't satisfy the condition then returns an empty collection.

```
IList<string> strList = new List<string>() {  
    "Three",  
    "Four",  
    "Five",  
    "Hundred" };  
  
var result = strList.TakeWhile(s => s.Length > 4);  
  
foreach(string str in result)  
    Console.WriteLine(str);
```

# Conversion Operators

- The Conversion operators in LINQ are useful in converting the type of the elements in a sequence (collection). There are three types of conversion operators: **As** operators (AsEnumerable and AsQueryable), **To** operators (ToArray, ToDictionary, ToList and ToLookup), and **Casting** operators (Cast and OfType).
- The following table lists all the conversion operators.

Method	Description
AsEnumerable	Returns the input sequence as IEnumerable<t>
AsQueryable	Converts IEnumerable to IQueryable, to simulate a remote query provider
Cast	Coverts a non-generic collection to a generic collection (IEnumerable to IEnumerable<T>)
OfType	Filters a collection based on a specified type
ToArray	Converts a collection to an array
ToDictionary	Puts elements into a Dictionary based on key selector function
ToList	Converts collection to List
ToLookup	Groups elements into an Lookup<TKey,TElement>

# Standard Query Operators - Summary

- <Q&A>



## Section 2

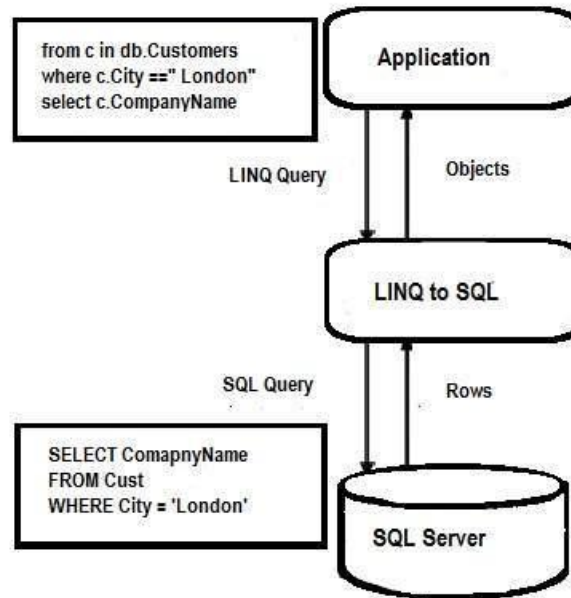
# LINQ TO SQL

# Introduction of LINQ To SQL 1 - 2

- LINQ to SQL offers an infrastructure (run-time) for the management of relational data as objects.
- LINQ to SQL (also known as DLINQ) is an electrifying part of Language Integrated Query as this allows querying data in SQL server database by using usual LINQ expressions.
- It also allows to update, delete, and insert data, but the only drawback from which it suffers is its limitation to the SQL server database.
- However, there are many benefits of LINQ to SQL over ADO.NET like reduced complexity, few lines of coding and many more.

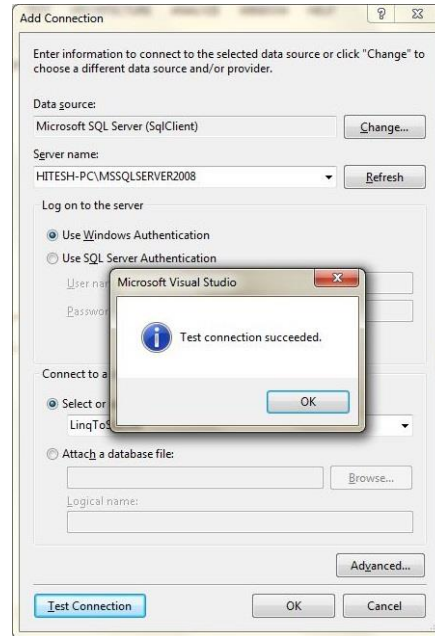
# Introduction of LINQ To SQL 2 - 2

- Below is a diagram showing the execution architecture of LINQ to SQL.



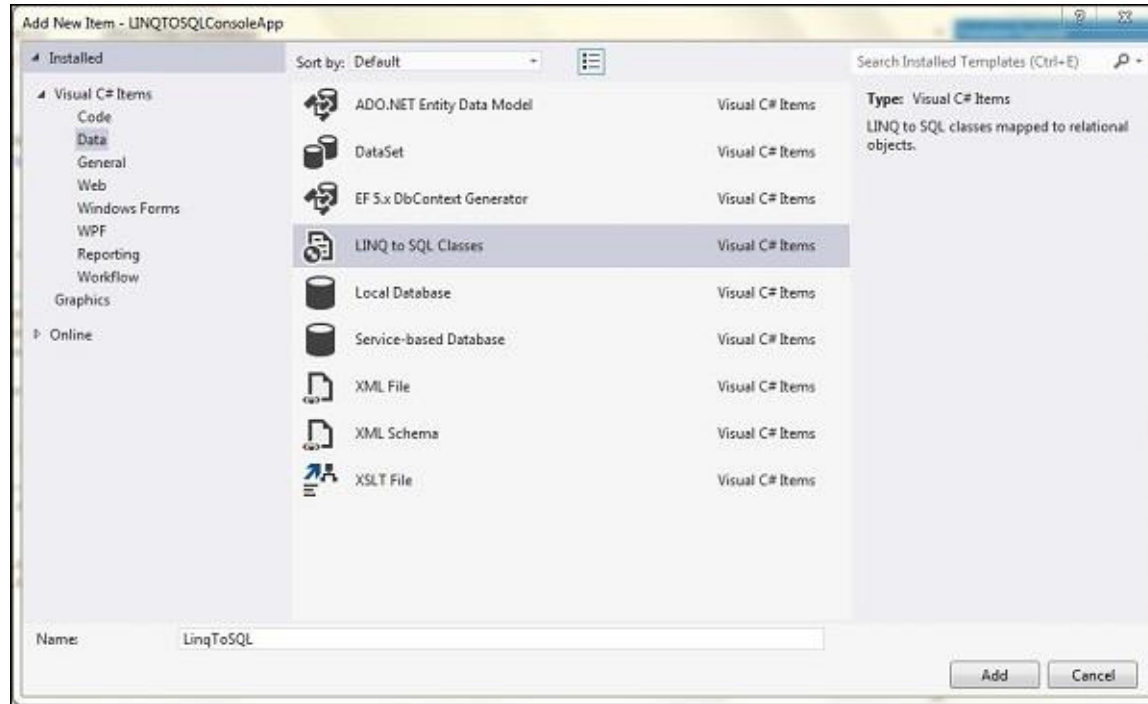
# How to Use LINQ to SQL? 1 - 4

- **Step 1** – Make a new “Data Connection” with database server. View &arr; Server Explorer &arr; Data Connections &arr; Add Connection



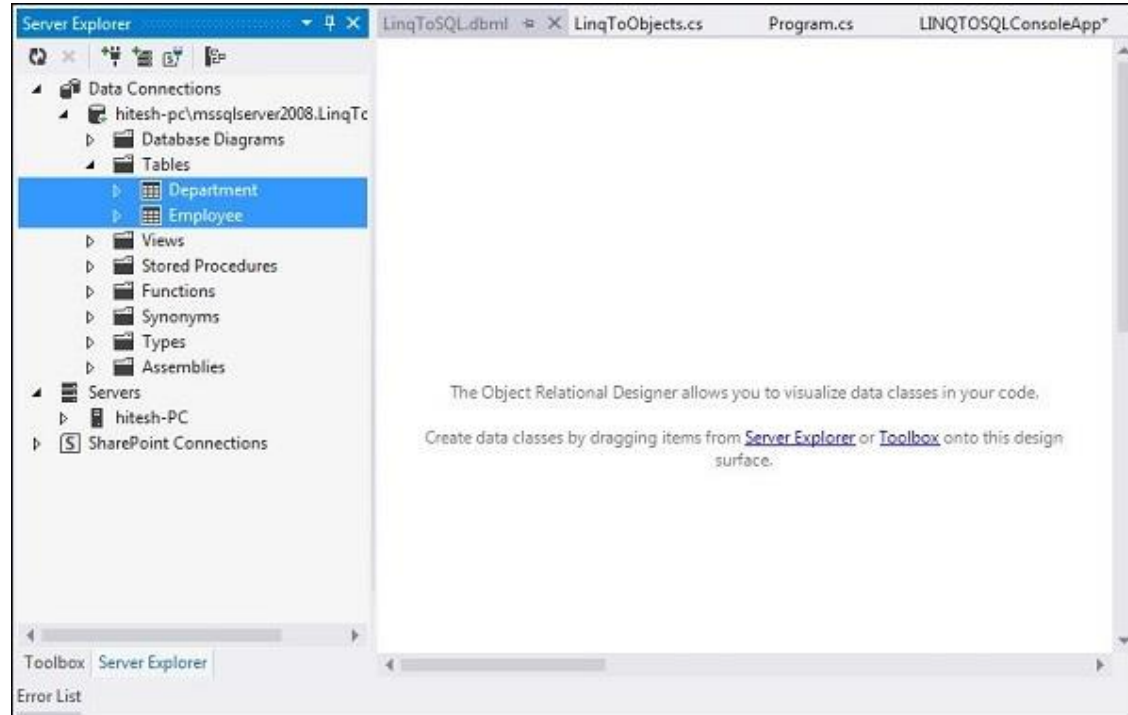
# How to Use LINQ to SQL? 2 - 4

- **Step 2** – Add LINQ To SQL class file



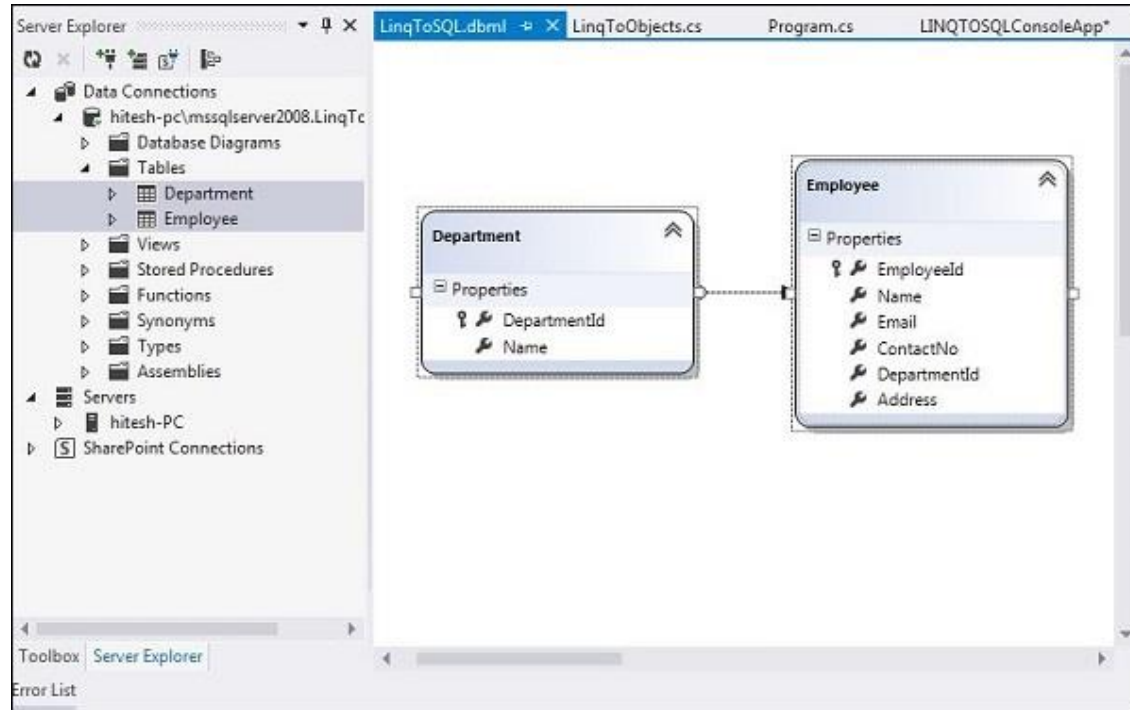
# How to Use LINQ to SQL? 3 - 4

- **Step 3** – Select tables from database and drag and drop into the new LINQ to SQL class file.



# How to Use LINQ to SQL? 4 - 4

- **Step 4** – Added tables to class file.



- The rules for executing a query with LINQ to SQL is similar to that of a standard LINQ query i.e. query is executed either deferred or immediate
  - ✓ **LINQ to SQL API** – requests query execution on behalf of an application and sent it to LINQ to SQL Provider.
  - ✓ **LINQ to SQL Provider** – converts query to Transact SQL(T-SQL) and sends the new query to the ADO Provider for execution.
  - ✓ **ADO Provider** – After execution of the query, send the results in the form of a DataReader to LINQ to SQL Provider which in turn converts it into a form of user object.



# Insert using LINQ To SQL

```
string connectionString = System.Configuration.ConfigurationManager.ConnectionStrings["LinqToSQLDBConnectionString"].ToString();

LinqToSQLDataContext db = new LinqToSQLDataContext(connectionString);

//Create new Employee

Employee newEmployee = new Employee();
newEmployee.Name = "Michael";
newEmployee.Email = "yourname@companyname.com";
newEmployee.ContactNo = "343434343";
newEmployee.DepartmentId = 3;
newEmployee.Address = "Michael - USA";

//Add new Employee to database
db.Employees.InsertOnSubmit(newEmployee);

//Save changes to Database.
db.SubmitChanges();

//Get new Inserted Employee
Employee insertedEmployee = db.Employees.FirstOrDefault(e => e.Name.Equals("Michael"));

Console.WriteLine("Employee Id = {0} , Name = {1}, Email = {2}, ContactNo = {3}, Address = {4}",
    insertedEmployee.EmployeeId, insertedEmployee.Name, insertedEmployee.Email,
    insertedEmployee.ContactNo, insertedEmployee.Address);

Console.WriteLine("\nPress any key to continue.");
Console.ReadKey();
```

# Update using LINQ To SQL

```
string connectionString = System.Configuration.ConfigurationManager.ConnectionStrings["LinqToSQLDBConnectionString"].ToString();

LinqToSQLDataContext db = new LinqToSQLDataContext(connectionString);

//Get Employee for update
Employee employee = db.Employees.FirstOrDefault(e => e.Name.Equals("Michael"));

employee.Name = "George Michael";
employee.Email = "yourname@companyname.com";
employee.ContactNo = "99999999";
employee.DepartmentId = 2;
employee.Address = "Michael George - UK";

//Save changes to Database.
db.SubmitChanges();

//Get Updated Employee
Employee updatedEmployee = db.Employees.FirstOrDefault(e => e.Name.Equals("George Michael"));

Console.WriteLine("Employee Id = {0} , Name = {1}, Email = {2}, ContactNo = {3}, Address = {4}",
    updatedEmployee.EmployeeId, updatedEmployee.Name, updatedEmployee.Email,
    updatedEmployee.ContactNo, updatedEmployee.Address);

Console.WriteLine("\nPress any key to continue.");
Console.ReadKey();
```

# Delete using LINQ To SQL

```
string connectionString = System.Configuration.ConfigurationManager.ConnectionStrings["LinqToSQLDBConnectionString"].ToString();

LinqToSQLDataContext db = new LinqToSQLDataContext(connectionString);

//Get Employee to Delete
Employee deleteEmployee = db.Employees.FirstOrDefault(e => e.Name.Equals("George Michael"));

//Delete Employee
db.Employees.DeleteOnSubmit(deleteEmployee);

//Save changes to Database.
db.SubmitChanges();

//Get All Employee from Database
var employeeList = db.Employees;
foreach (Employee employee in employeeList) {
    Console.WriteLine("Employee Id = {0} , Name = {1}, Email = {2}, ContactNo = {3}",
        employee.EmployeeId, employee.Name, employee.Email, employee.ContactNo);
}

Console.WriteLine("\nPress any key to continue.");
Console.ReadKey();
```

# Thank you

