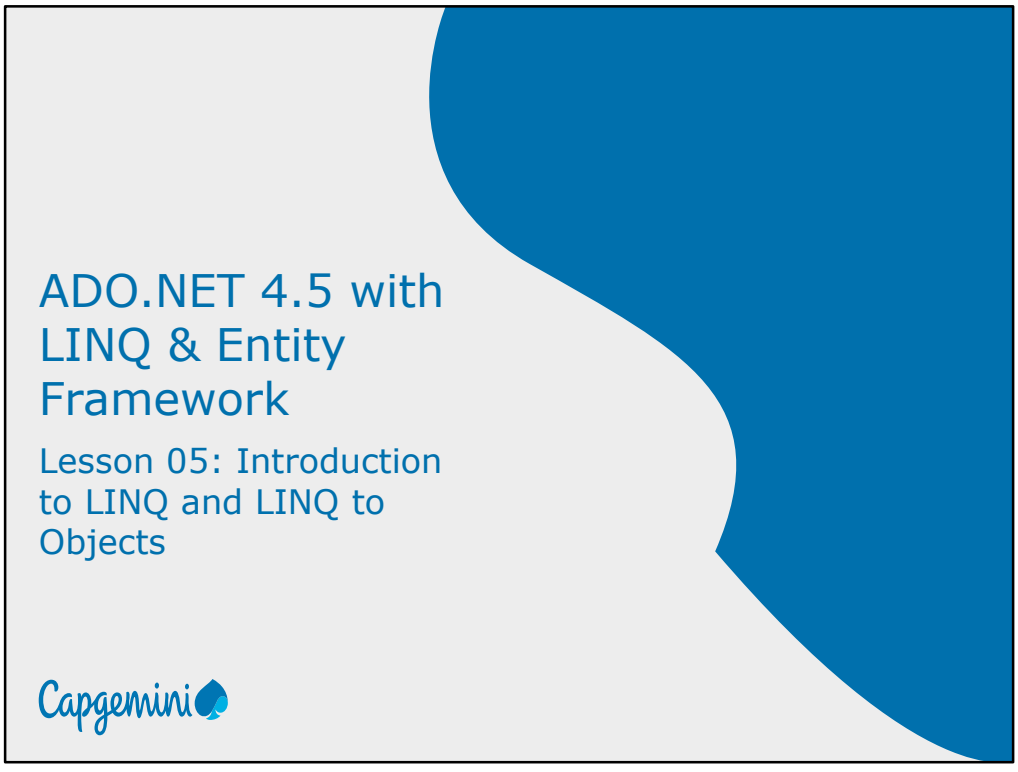

ADO.NET 4.5 with LINQ & Entity Framework
Lesson 05: Introduction to LINQ and LINQ to Objects
Capgemini

Lesson Objectives

- In this lesson we will cover the following
  - Overview of LINQ
  - LINQ Architecture and Components
  - Basic Query Operations with LINQ

Overview

- LINQ stands for Language Integrated Query
- It was introduced from .NET Framework 3.5 and continued as a part of .NET Framework 4.0 and 4.5
- It bridges the gap between the world of objects and the world of data
- It adds Query capabilities to the Programming Language
- LINQ enables you to query data from within the .NET Programming Language in the same way the SQL enables your to query data from a database.

Traditionally ,Developer use to write down code for accessing database object ADO.NET where they use to pass SQL query as string to the database which does not have a compile time due which if the query if formed incorrect it use to result in Runtime exception and abruptly putting a stop to Program execution flow .
To over come this problem LINQ was introduced as a part of .NET framework 3.5 which allowed the developer to write query to handle data using programming language like C#.
With the introduction of **L**anguage **In**tegrated **Q**uery (LINQ), the developers now can:

- deal with data by using a consistent programmatic approach.
- perform data access with new data design surfaces.

LINQ aims to reduce the complexity for developers. It helps boost their productivity through a set of extensions to the C# and Visual Basic programming languages, as well as the Microsoft .NET Framework, which provides integrated querying for objects, databases, and XML data.
Using LINQ, developers are able to write queries natively in C# or Visual Basic without using specialized languages, such as Structured Query Language (SQL) and Xpath.
With Visual Studio , you can work with data in the way you want, namely:

- You can create entire collections of objects from a database backend, if required.
- You can interact with data as rows or columns – whatever makes sense to your application.

Language Integrated Query or “LINQ” dramatically changes the way we work with and program against data. By creating a common querying language in LINQ, the developer is free
to focus on things that matter most. LINQ provides the ease of use that you have come to expect with Visual Studio offering both “IntelliSense” and “Autocompletion” right in the IDE.

1.1: LINQ

Overview

- Traditionally , queries against data was a simple string without type checking at compile time and IntelliSense support , which used to result in runtime exceptions
- Unlike the traditional query LINQ query uses language construct and uses features like compile time check and IntelliSense

LINQ provides native querying syntax in C# and VB.Net. This frees the developer from having to master independent data programmability technologies (for example: Xpath, Xquery, T/SQL). Instead, LINQ offers the developer a consistent way to query data.

The best part is that the LINQ code that you write is consistent. This is irrespective of whether the data store is one of the following:

- a SQL Server
- a data store contained in a ADO.NET Dataset
- an XML document
- an EDM that you create, or
- an object that you create in memory

1.1: LINQ

Overview

- Benefit of using LINQ is that one can:
  - work with data in a consistent way, regardless of the type of data
  - interact with data as objects
  - integrate better with programming languages
  - improve productivity through IntelliSense in Visual Studio

1.1: LINQ

Overview

- The design goals for LINQ are:
  - to integrate objects, relational data, and XML
  - to provide SQL and XQuery-like power in C# and VB
  - to provide Extensibility model for languages
  - to provide Extensibility model for multiple data sources
  - to provide Type safety
  - to provide Extensive IntelliSense support (enabled by strong-typing)
  - to provide Debugger support

LINQ- Design Goals:

Design goals for LINQ are:

- **To integrate objects, relational data, and XML**
  - To provide unified query syntax across data sources to avoid different languages for different data sources.
  - To provide Single model for crunching all types of data regardless of source or in-memory representation.
- **To provide SQL and XQuery-like power in C# and VB**
  - To integrate querying abilities right into the programming languages.
- **To provide Extensibility model for languages**
  - To enable implementation for other programming languages.
- **To provide Extensibility model for multiple data sources**
  - To be able to access other data sources than relational databases or XML documents.
  - To allow other frameworks to implement LINQ for their own needs.
- **To provide Type safety**
  - To provide “compile-time type checking” to avoid problems that were previously discovered only at run-time.
  - To enable the compiler to catch errors in your queries.

1.1: LINQ

Overview

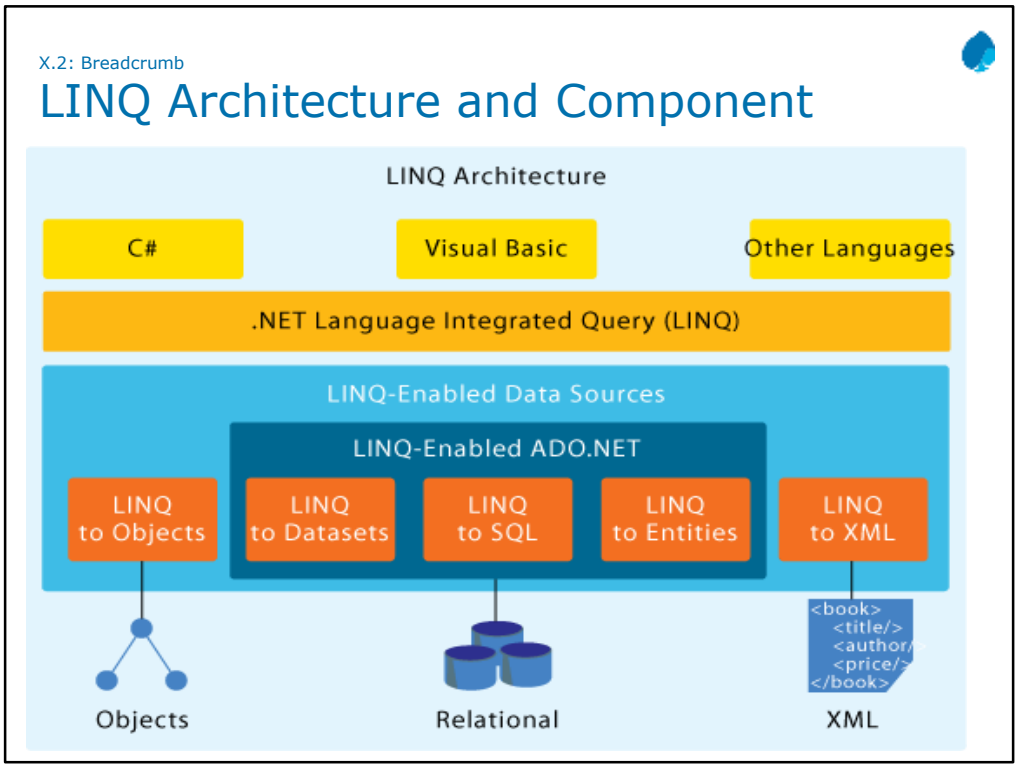
- Hidden Slide with Notes

Design Goals of LINQ (contd.):

- **To provide extensive IntelliSense support (enabled by strong-typing)**
  - To assist the developers when writing queries to improve productivity
  - To help the developers get up to speed with the new syntax. (The editor guides when writing queries.)
- **To provide Debugger support:**
  - To provide Debugger support that allows the developers to debug LINQ queries in a step by step fashion along with rich debugging information.

- The number one LINQ feature is the ability to deal with several data types and sources. LINQ ships with implementations that support querying against regular object collections, databases, entities, and XML sources. Since LINQ supports "rich extensibility", the developers can easily integrate LINQ with other data sources and providers, as well.

- Another essential feature of LINQ is that it is "strongly-typed". This means that:
  - You get compile-time checking for all queries, unlike contemporary SQL statements, where you typically find out only at run-time in case something is wrong. This implies that you will be able to check during development that your code is correct. The direct benefit is a reduction in the number of problems discovered late in production. Usually, bugs arise due to human factors. Strongly-typed queries allow early detection of typos, and mistakes done by the developer in charge of the keyboard.
  - You get IntelliSense within Visual Studio when writing LINQ queries. This makes typing faster. It also makes working easier - against both simple and complex collection and data source object models.



LINQ Architecture and Components:-

LINQ is currently integrated directly in the native syntax for C# 5.0 and VB 10 which are part of .Net Framework 4.5 and Visual Studio

The above diagram show the architecture of the LINQ and all the components of LINQ which are as follows

The top most layer consist of language extension which show all the languages which can use LINQ including C# , VB and other langugage which are the part of .Net Framework

The second layer consist of .NET LINQ which

The third layer consist of LINQ Enabled Data source which specifies different data source on which LINQ can be used . LINQ can be use with Objects, SQL, Dataset and XML.

LINQ comes in different flavors based on data source being used as follows

i) LINQ to Objects
ii) LINQ to ADO.Net which includes LINQ to Datasets, LINQ to SQL, LINQ to Entities
iii) LINQ to XML

X.X: [Topic]

LINQ Architecture and Components

- Different Flavors of LINQ
  - LINQ to Objects
  - LINQ to ADO.NET
    - LINQ to SQL
    - LINQ to Datasets
    - LINQ to Entities
  - LINQ to XML
  - Parallel LINQ(PLINQ)

LINQ to Objects:

- SQL-like queries for any .NET collection (or anything that implements Ienumerable)
- The LINQ to Objects API supports queries over any .NET collection, such as arrays and generic lists.
- This API is defined in the System.Linq namespaces inside System.Core.dll
- LINQ to objects is enabled by including the System.Linq namespace.
- Manipulating collections of objects, which can be related to each other to form a hierary or a graph. From a certain point of view, LINQ to Objects is the default implementation used by a LINQ query.

LINQ to ADO.NET:

- Query enabled data access framework
- Allows to perform query operation on Sql, DataSet and Entities
- API is defined in System.Data.Linq namespace

LINQ to XML:

- Query enabled, smaller, faster XML DOM
- API is defined in System.XML.Linq namespace

- One important thing to remember is that querying syntax use in LINQ remains consistent regardless of the type of data you are working with

Basic Query Operations with LINQ

- LINQ Queries are written using the LINQ declarative query syntax.
- These queries uses a set of query keywords built into the .NET Framework that allows the developer to write SQL like commands in programming language.
- Commonly used Keywords are
  - from / in
  - where
  - orderby
  - select
  - groupby

Basic Query Operations with LINQ

- All LINQ query operations require the following three distinct actions:
  - Obtain the data source
  - Create the query
  - Execute the query

Basic Operations in LINQ:
A query is an expression that retrieves data from a data source. Queries are usually expressed in a specialized query language. Different languages have been developed over time for the various types of data sources.
For example: SQL for relational databases and XQuery for XML.

Therefore developers have had to learn a new query language for each type of data source or data format that they must support. LINQ simplifies this situation. It offers a consistent model for working with data across various kinds of data sources and formats. In a LINQ query, you are always working with objects. You use the same basic coding patterns to query and transform data in XML documents, SQL databases, ADO.NET Datasets, .NET collections, and any other format for which a LINQ provider is available.

All LINQ query operations consist of three distinct actions:
1. Obtain the data source.
2. Create the query.
3. Execute the query.

Example

```
class IntrotoLINQ
{
   static void Main()
   {
         //1.Data Source
         int [] numbers = new int[7]{0,1,2,3,4,5,6};

         //2.Query Creation
         var numQuery= from num in numbers
                       where (num%2)==0
                       select num;
         //3.Query Execution
         foreach(int num in numQuery)
         {Console.Write("{0,1}",num);}
   }
}
```

X.X: [Topic]

Basic Query Operations with LINQ

- Syntax

```
from <range variable> in <collection>

<filter, joining, grouping, aggregate operators etc.> <lambda expression>

<select or groupBy operator> <formulate the result>
```

In the above diagram we can see the syntax of a Basic LINQ Query

Every LINQ query begins with a from clause ends with a select query or group. In between can be use filtering , joining , grouping etc on the data.

Query looks similar to SQL and will provide intellisense support.

X.X: [Topic]

Basic Query Operations with LINQ

- Code Snippet

```
static void Main(string[] args)
{
    int[] numbers = { 10, 22, 98, 76, 81, 99, 181, 71, 31 };

    var result = from number in numbers
                 where number % 2 == 0
                 select number;

}
```

Add the notes here.
The Code Snippet in the Slide returns even number from the Integer array and stores them in variable name result

X.X: [Topic]

Demo

- Demo of Implementing Simple LINQ Queries

Add the notes here.
Demo of Simple LINQ Queries.

Basic Query Operators LINQ

- LINQ provide some standard query operator that can be used to query data
- Following is the categories of operator in LINQ
  - Filtering Operators
  - Projection Operators
  - Sorting Operators
  - Aggregation
  - Grouping Operators
  - Conversions

Basic Query Opertors

Filtering Operators

i) Where :- Filter value based on a predicate function

Projection Operators

i) Select :- The operator projects value on basis of a transform function

Sorting Operators

i) OrderBy :- The operator sort values in an ascending order
ii) OrderByDescending :- The operator sort values in a descending order
iii) ThenBy:- Executes a secondary sorting in an ascending order
iv) ThenByDescending :- Executes a secondary sorting in a descending order
v) Reverse :- Performs a reversal of the order of the elements in a collection

Aggregation

i) Average :- Average value of collection of value is calculated
ii) Count :- Counts the number of entries
iii) Max :- Find out the maximum value within a collection
iv) Min :- Find out the Minimum value within a collection
v) Sum :- Find out the sum of value within a collection

Filtering Operator

- Where Operator:-
  - It is filteriing opertor
  - It filter a sequence based on a predicate function

The Where Operator:

Suppose that you have to list the names and cities of customers from Italy. To filter a set of items, you can use the Where operator, which is also called a “restriction operator” because it restricts a set of items.

Projection Operator

- Select operator:
  - It is a projection operator
  - It enumerates the source sequence, and yields the results of evaluating the selector function for each element
- SelectMany:
  - It is a projection operator
  - It performs a one-to-many element projection over a sequence

Projection Operators:

Select:

- The Select operator performs a projection over a sequence.
- When the object returned by Select is enumerated, it enumerates the source sequence. Subsequently, it yields the results of evaluating the select or function for each element.
  - The first argument of the selector function represents the element to process.
  - The second argument, if present, represents the zero-based index of the element within the source sequence.

SelectMany:

The SelectMany operator performs a projection over a sequence. This operator is similar to Select because it allows us to define the elements that have to be picked up from a sequence. The difference is in the return type.

With the IEnumerable<S> type returned by the selector parameter of SelectMany, it is possible to concatenate many projection operations together. This can be done either on different sequences or starting from the result of a previous query.

Sorting Operator

- OrderBy and OrderByDescending:
  - The OrderBy and OrderByDescending operators order elements of a sequence according to a given key
  - The OrderByDescending operator inverts the ordering.
- ThenBy and ThenByDescending:
  - These operators are useful for specifying additional ordering keys after the first one is specified either by the OrderBy or OrderByDescending operator
  - ThenByDescending is similar to ThenBy. However, it inversely sorts the sequence
- Reverse:
  - This operator returns a new sequence having elements in a reverse ordering of the source sequence

Sorting Operators:

OrderBy and OrderByDescending:

- Similar to the ORDER BY and ORDER BY DESC in SQL, the OrderBy and OrderByDescending operators order elements of a sequence according to a given key. The OrderByDescending operator inverts the ordering.

ThenBy and ThenByDescending:

- OrderBy allows us to specify only one ordering key. Hence we have to use either ThenBy or ThenByDescending to concatenate ordering-key values.

Reverse:

- This method simply returns a new sequence with elements in a reverse ordering of the source sequence.

Grouping Operators

- GroupBy:
  - This operator groups the elements of a sequence.

```
int[] numbers = { 1, 2, 3, 4, 5, 6, 7, 8, 9 };

var query = numbers.ToLookup(i => i % 2);

foreach (IGrouping<int, int> group in query)
{
    Console.WriteLine("Key:  {0}", group.Key);
    foreach (int number in group)
    {
        Console.WriteLine(number);
    }
}
```

Grouping Operators:
Similar to the GROUP BY clause of SQL, the GroupBy operator groups elements of a sequence based on a given selector function.

Concatenation Operator

- Concat:
  - This operator concatenates two sequences

```
int[] numbers = { 1, 2, 3, 4, 5, 6, 7, 8, 9 };
int[] moreNumbers = { 10, 11, 12, 13 };
var query = numbers.Concat(moreNumbers);
foreach(var item in query)
Console.WriteLine(item);
```

Concatenation Operators:

- This operator concatenates two sequences.
- The resulting IEnumerable<T> type is the concatenation of the first and second sequences specified as a parameter.

Set Operator

- Distinct:
  - This operator eliminates duplicate elements from a sequence
- Except:
  - This operator enumerates the first sequence, collecting all distinct elements. Subsequently, it enumerates the second sequence, thus removing elements contained in the first sequence
- Intersect:
  - The operator enumerates the first sequence, collecting all distinct elements
  - It then enumerates the second sequence, yielding elements that occur in both sequences
- Union:
  - The operator produces a set union of two sequences

Set Operators:

Distinct:

- This operator is similar to the DISTINCT keyword used in SQL. It eliminates the duplicates from a sequence.
- When the code processes the query, it enumerates the element of the sequence, and stores into an IEnumerable<T> type. It is done for each element that has not been previously stored.
- The Distinct operator selects unique values from the sequence.

Except:

- This Operator Produces the set difference of two sequences by using the default equality comparer to compare values.

Intersect:

- This operator returns a sequence made by common elements of two different sequences.
- The Intersect Method is used to retrieve common elements in two sequences as shown below:

Union:

- This operator returns a new sequence formed by uniting the two different sequences.

Conversion Operators

- AsEnumerable:
  - This operator returns its argument typed as IEnumerable<T>
- OfType:
  - This operator filters the elements of a sequence based on a type
- ToArray:
  - This operator creates an array from a sequence

Conversion Operators:

AsEnumerable
- This operator produces a new IEnumerable<T> type composed of only the element of the specified type

OfType:
- The OfType Searches for the specified type T in the Sequence

ToArray:
- This operator returns an array composed of the elements of the source sequence.

Aggregate Operators

- Aggregate:
  - The operator applies a function over a sequence

```
int[] numbers = { 1, 2, 3, 4, 5, 6, 7, 8, 9 };
var query = numbers.Aggregate((a, b) => a * b);
Console.WriteLine(query);
```

- Average:
  - The operator computes the average of a sequence of numeric values Count & LongCount
  - It counts the number of elements in a sequence

Aggregate Operator (Contd..)

- Max:
  - The operator finds the maximum value from a sequence of numeric values
- Min:
  - The operator finds the minimum value from a sequence of numeric values
- Sum:
  - The operator computes the sum of a sequence of numeric values

Element Operators

- First:
  - The operator returns the first element of a sequence
- FirstOrDefault:
  - The operator returns the first element of a sequence, or a default value if no element is found
- Last:
  - The operator returns the last element of a sequence
- LastOrDefault:
  - The operator returns the last element of a sequence, or a default value if no element is found

Element Operators (Contd..)

- Single:
  - The operator returns the single element of a sequence. An exception is thrown if the source sequence contains no match or more than one match
- SingleOrDefault:
  - The operator returns the single element of a sequence, or a default value if no element is found. The default value is for reference and nullable types

Element Operators (Contd..)

- DefaultIfEmpty:
  - The operator supplies a default element for an empty sequence. It can be combined with a grouping join to produce a left outer join
- ElementAt:
  - It returns the element at a given index in a sequence
- ElementAtOrDefault:
  - The operator returns the element at a given index in a sequence, or a default value if the index is out of range

Example

- Example 1:

```
int[] numbers = { 1, 2, 3, 4, 5, 6, 7, 8, 9 };
var query = numbers.First();
Console.WriteLine("The first element in the sequence");
Console.WriteLine(query);
query = numbers.Last();
Console.WriteLine("The last element in the sequence");
Console.WriteLine(query);
Console.WriteLine("The first even element in the sequence");
query = numbers.First(n => n % 2 == 0);
Console.WriteLine(query);
Console.WriteLine("The last even element in the sequence");
query = numbers.Last(n => n % 2 == 0);
Console.WriteLine(query);
```

Example

- Example 2: FirstOrDefault and LastOrDefault

```
int[] numbers = {1, 3, 5, 7, 9};
var query = numbers.FirstOrDefault(n => n % 2 == 0);
Console.WriteLine("The first even element in the sequence");
Console.WriteLine(query);
Console.WriteLine("The last odd element in the sequence");
query = numbers.LastOrDefault(n => n % 2 == 1);
Console.WriteLine(query);
```

- Using the FirstOrDefault/LastOrDefault methods we obtain the same results. However, when we use those methods, and a predicate does not find an element satisfying the specified condition, a default value is returned (thereby avoiding retrieval of an exception).
- In the above example , Since no even numbers are in the sequence, FirstOrDefault returns the zero default value. On the other hand, the LastOrDefault operator looks for the last odd number in the sequence, and finds the number 9.

Example

- Example 3: Single and SingleOrDefault

```
int[] numbers = {1, 2, 3, 4, 5, 6, 7, 8, 9};
var query = numbers.Single(n => n > 8);
Console.WriteLine(query);

int[] numbers = {1, 2, 3, 4, 5, 6, 7, 8, 9};
var query = numbers.SingleOrDefault(n => n > 9);
Console.WriteLine(query
```

- Using the Single method, if no element satisfies the predicate condition, an exception is thrown.
- Using the SingleOrDefault method, when no element satisfies the predicate function, either a null or zero value is returned.
- The difference between the null and zero value depends on the source type: null for reference types (i.e., strings) and zero for value types (i.e., integers).

Example

- Example 4: ElementAt and ElementAtOrDefault

```
int[] numbers = {1, 2, 3, 4, 5, 6, 7, 8, 9};
var query = numbers.ElementAt(4);
Console.WriteLine(query);

int[] numbers = {1, 2, 3, 4, 5, 6, 7, 8, 9};
var query = numbers.ElementAtOrDefault(9);
Console.WriteLine(query);
```

X.X: [Topic]

Demo

- Demo of Implementing standard operator in LINQ

Add the notes here.
Demo of Simple LINQ Queries.

X.X: [Topic]

Lab

- Lab Topic

Add the notes here.

Summary

- In this lesson we have learnt the following topic
  - Overview of LINQ
  - LINQ Architecture and Components
  - Basic Query Operation using LINQ

Add the notes here.

Review Question

- Which of the following version of .NET framework introduced LINQ?
  - .NET Framework 2.0
  - .NET Framework 3.5
  - .NET Framework 3.5 SP1
  - .NET Framework 3.0
- LINQ uses programming language syntax to define queries
  - True
  - False

Add the notes here.

Review Question

- LINQ stands for _____
  - Language in Query
  - Language Integrated Query
  - Language Independent Query
  - Language Include Query

Add the notes here.