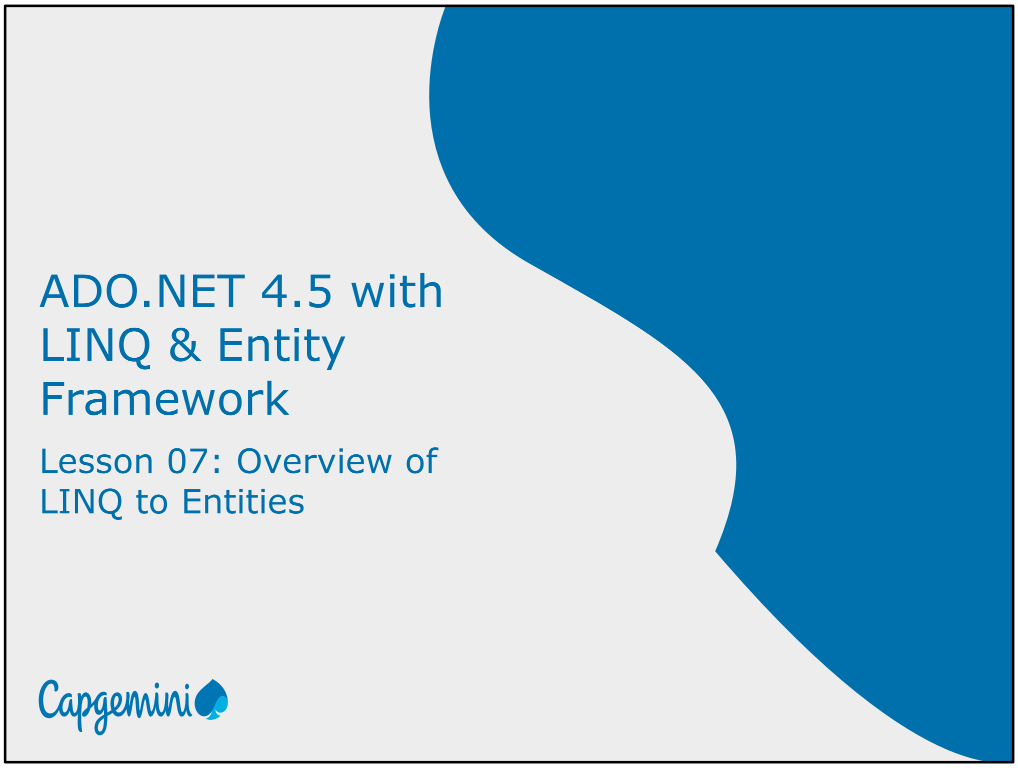
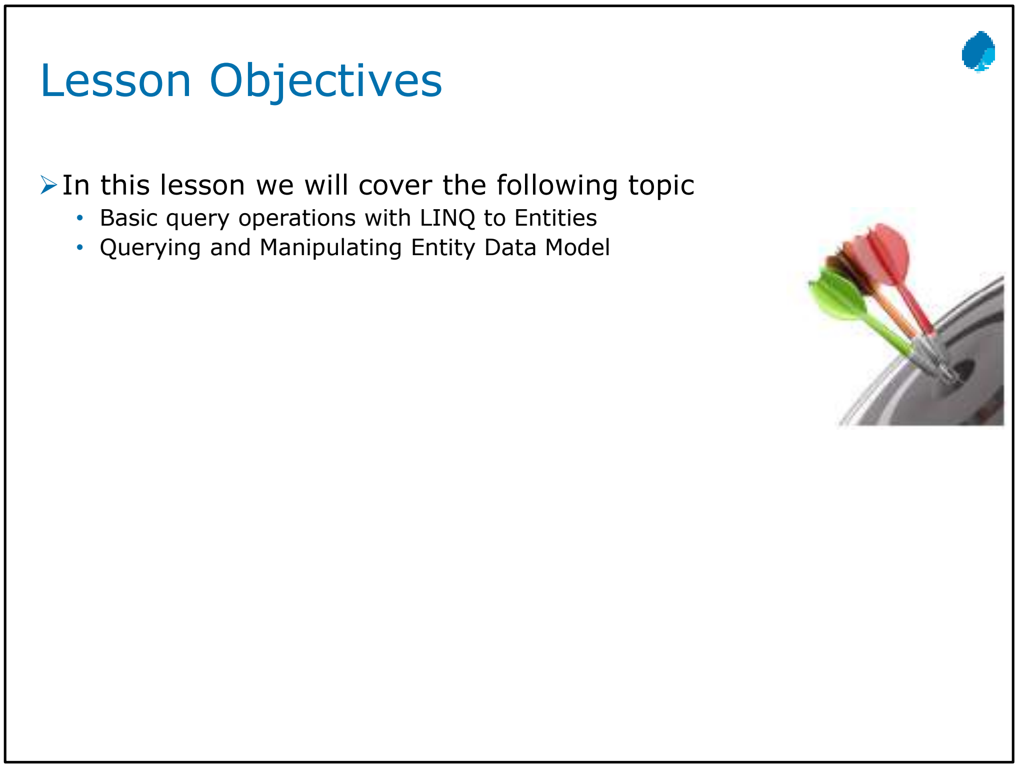


ADO.NET 4.5 with LINQ & Entity Framework

Lesson 07: Overview of LINQ to Entities

Capgemini



Lesson Objectives

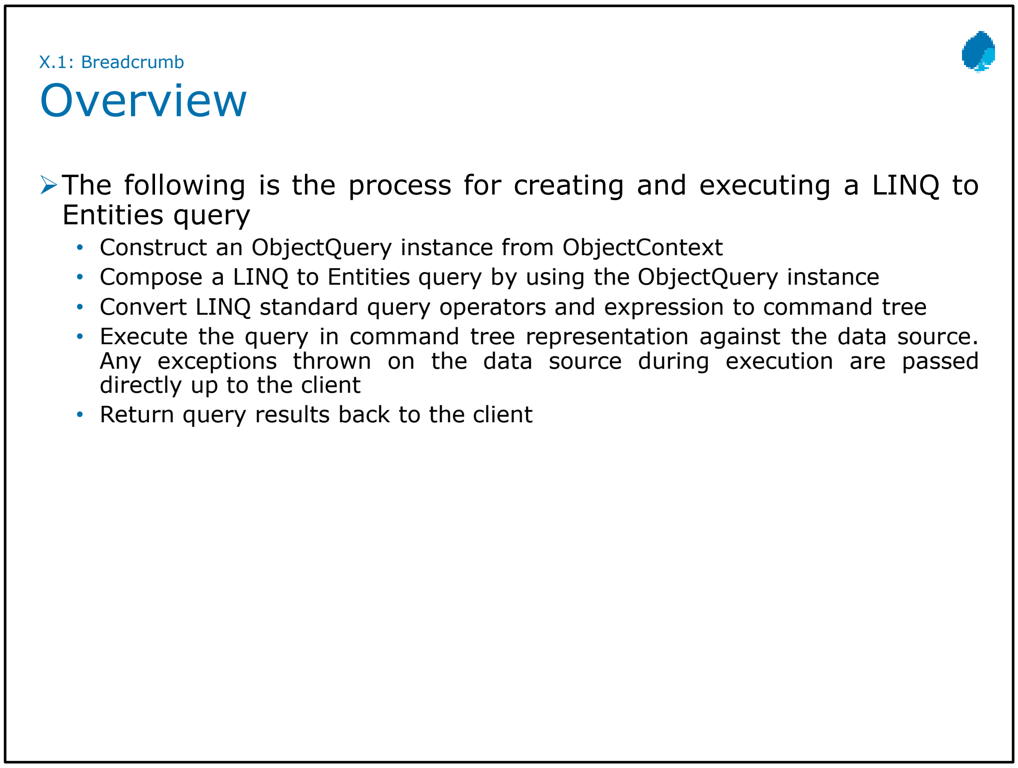
- In this lesson we will cover the following topic
  - Basic query operations with LINQ to Entities
  - Querying and Manipulating Entity Data Model

Overview

- LINQ to Entities provides LINQ support that enables developer to write queries against the Entity Framework Conceptual model
- Queries against the Entity Framework are represented by command tree queries which against the object context
- LINQ to Entities converts LINQ queries to command tree queries, executes the queries against the Entity Framework and returns the object that can be used by both the Entity Framework and LINQ

LINQ to Entities :-

LINQ to Entities provides Language-Integrated Query (LINQ) support that enables developers to write queries against the Entity Framework conceptual model using Visual Basic or Visual C#.
Queries against the Entity Framework are represented by command tree queries, which execute against the object context.
LINQ to Entities converts Language-Integrated Queries (LINQ) queries to command tree queries, executes the queries against the Entity Framework, and returns objects that can be used by both the Entity Framework and LINQ.



X.1: Breadcrumb

Overview

- The following is the process for creating and executing a LINQ to Entities query
  - Construct an ObjectQuery instance from ObjectContext
  - Compose a LINQ to Entities query by using the ObjectQuery instance
  - Convert LINQ standard query operators and expression to command tree
  - Execute the query in command tree representation against the data source. Any exceptions thrown on the data source during execution are passed directly up to the client
  - Return query results back to the client

ObjectContext :- Provides facilities for querying and working with entity data as objects.

ObjectQuery

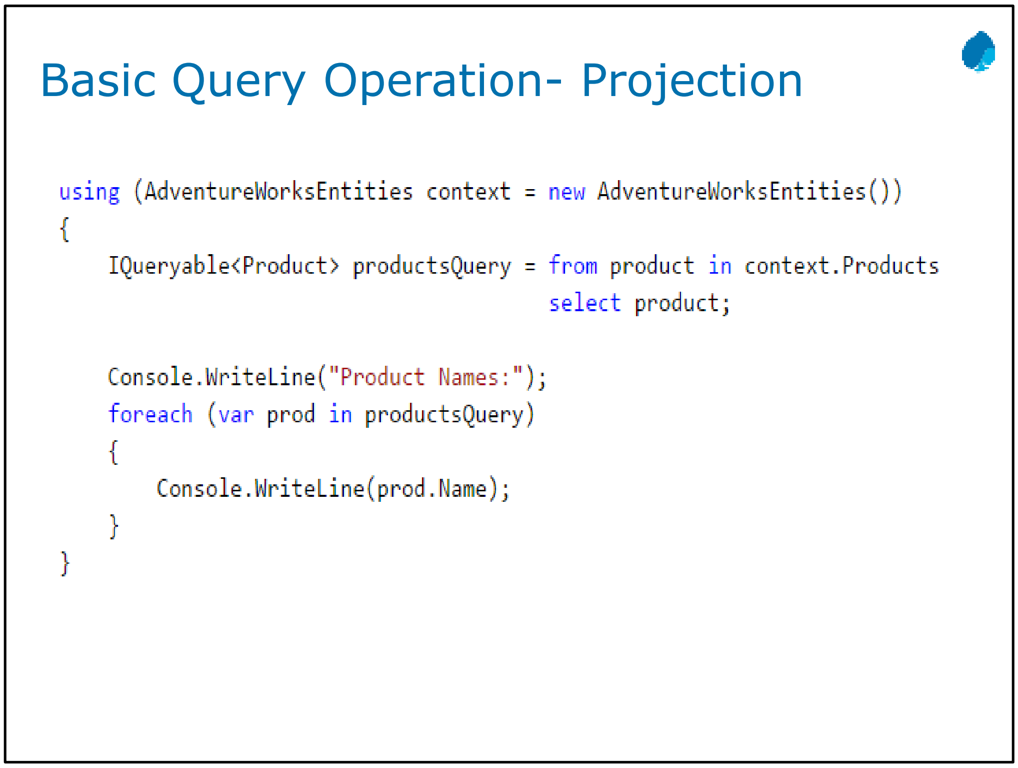
IQueryable

X.2: Breadcrumb

Basic Query Operations

- LINQ to Entities enables you to define LINQ queries on the entities that are returned by an object query.
- In a typical scenario, an application defines an object query to retrieve data from the persistent data store into a collection of entities in memory, and then it uses LINQ to Entities to query the entities without requiring additional roundtrips to the database.
- We can perform basic operation like projection,filtering ,ordering and aggregation on the Entities

Add the notes here.



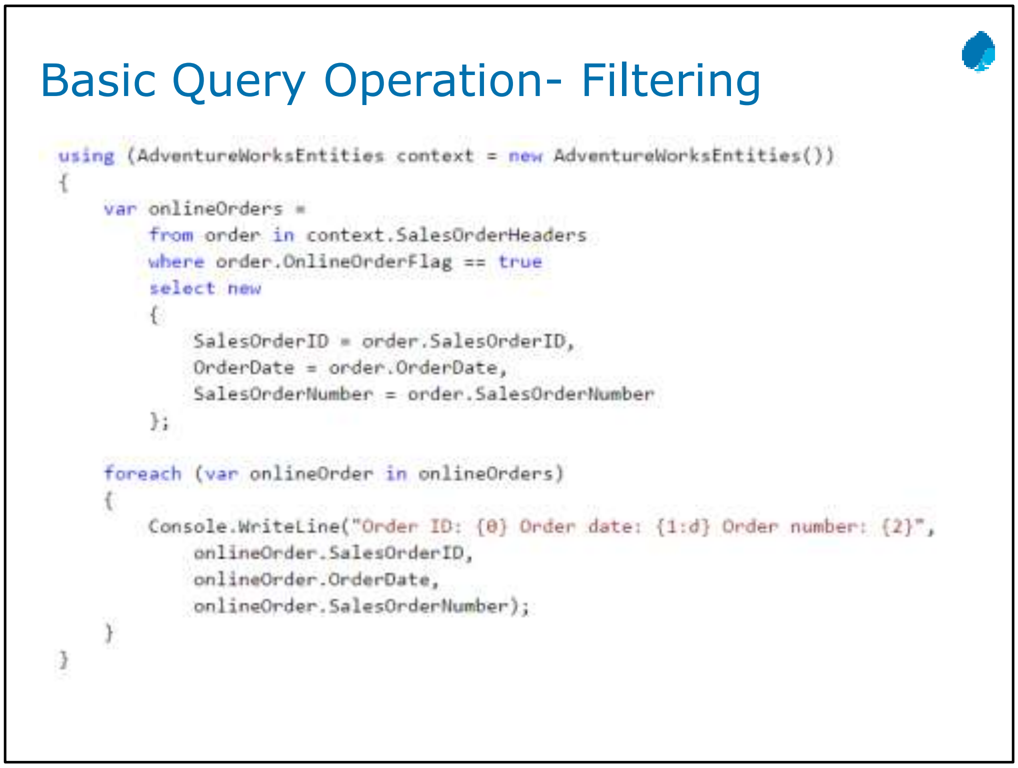
Basic Query Operation- Projection

```
using (AdventureWorksEntities context = new AdventureWorksEntities())
{
    IQueryable<Product> productsQuery = from product in context.Products
                                        select product;

    Console.WriteLine("Product Names:");
    foreach (var prod in productsQuery)
    {
        Console.WriteLine(prod.Name);
    }
}
```

The slide show a simple query using LINQ to Entities . It show use of select

The above query returns name of all the products from the Product table



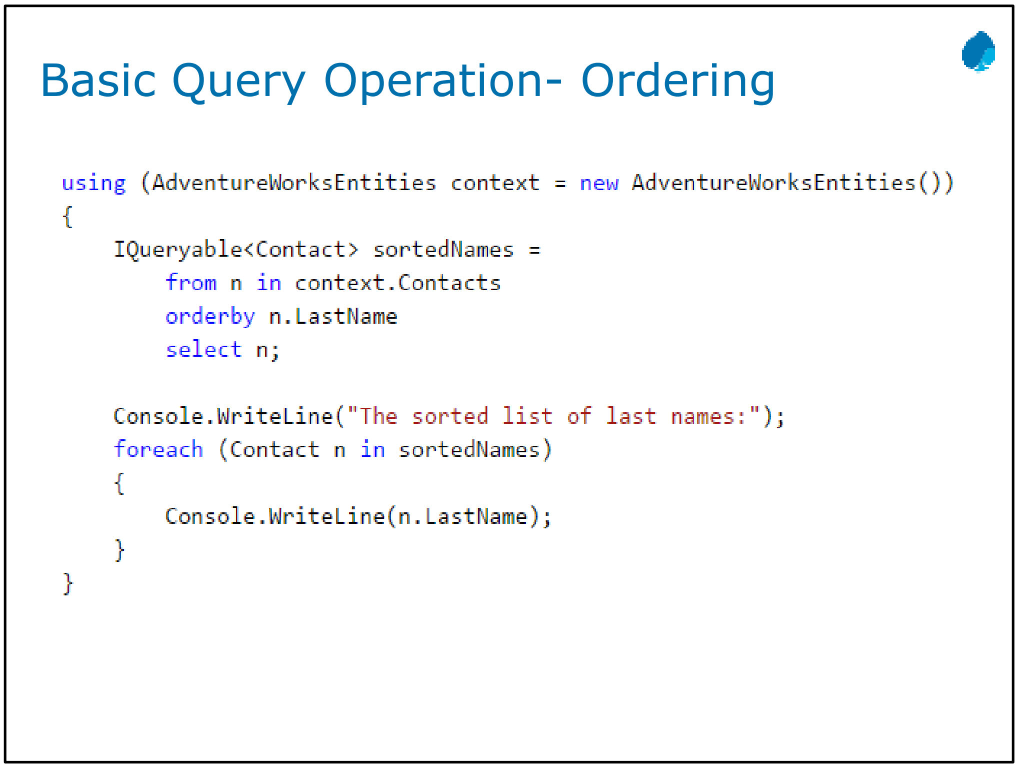
Basic Query Operation- Filtering

```
using (AdventureWorksEntities context = new AdventureWorksEntities())
{
    var onlineOrders =
        from order in context.SalesOrderHeaders
        where order.OnlineOrderFlag == true
        select new
        {
            SalesOrderID = order.SalesOrderID,
            OrderDate = order.OrderDate,
            SalesOrderNumber = order.SalesOrderNumber
        };

    foreach (var onlineOrder in onlineOrders)
    {
        Console.WriteLine("Order ID: {0} Order date: {1:d} Order number: {2}",
            onlineOrder.SalesOrderID,
            onlineOrder.OrderDate,
            onlineOrder.SalesOrderNumber);
    }
}
```

The slide show example of Filtering of data using LINQ to Entites. In this example we are using Where for filtering

The query will returns all the order which was placed online



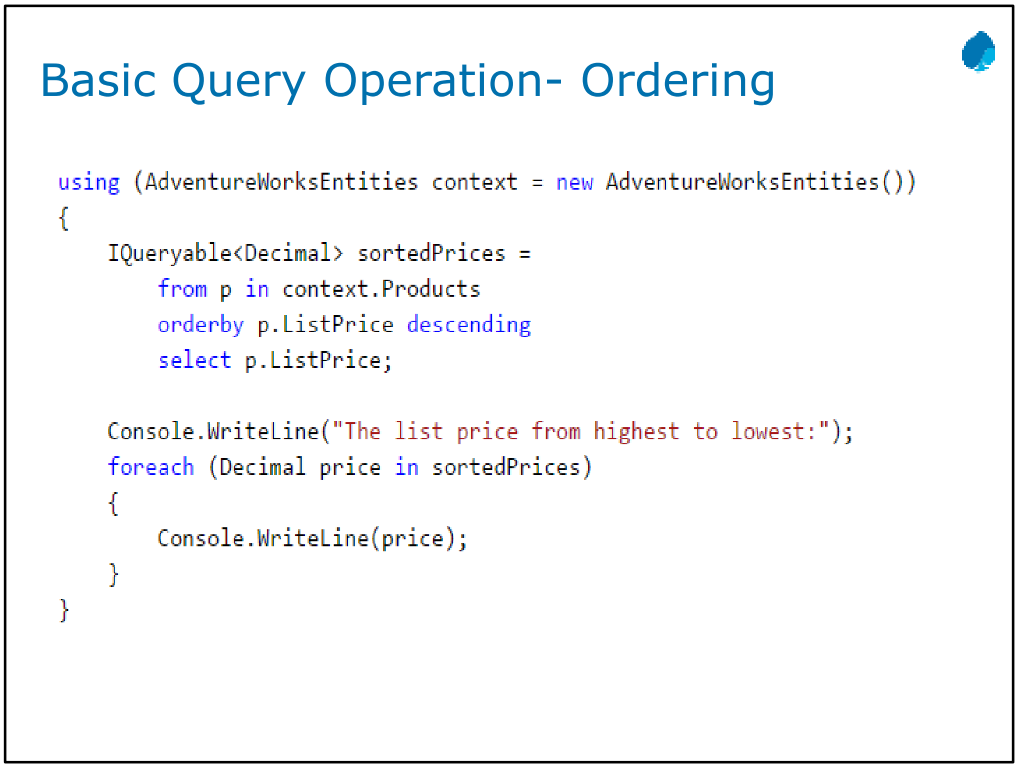
Basic Query Operation- Ordering

```
using (AdventureWorksEntities context = new AdventureWorksEntities())
{
    IQueryable<Contact> sortedNames =
        from n in context.Contacts
        orderby n.LastName
        select n;

    Console.WriteLine("The sorted list of last names:");
    foreach (Contact n in sortedNames)
    {
        Console.WriteLine(n.LastName);
    }
}
```

The above example shows use of OrderBy.

The query will return a list of contacts ordered by last name.



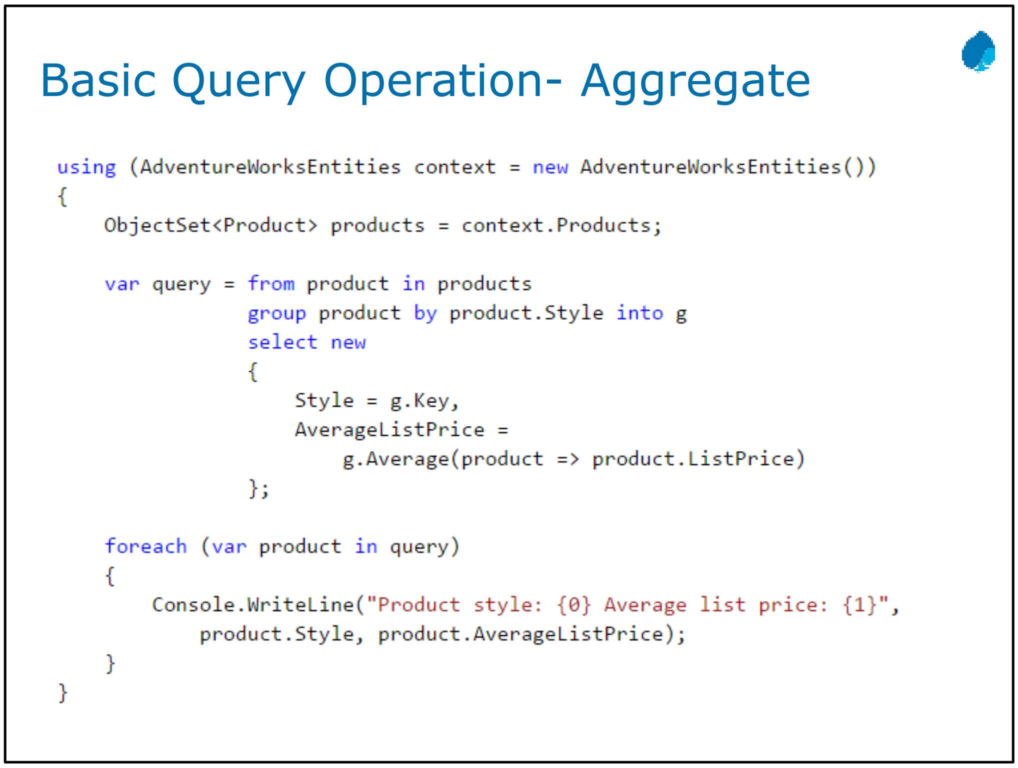
Basic Query Operation- Ordering

```
using (AdventureWorksEntities context = new AdventureWorksEntities())
{
    IQueryable<Decimal> sortedPrices =
        from p in context.Products
        orderby p.ListPrice descending
        select p.ListPrice;

    Console.WriteLine("The list price from highest to lowest:");
    foreach (Decimal price in sortedPrices)
    {
        Console.WriteLine(price);
    }
}
```

The above example shows use of OrderByDescending.

The query will return sorted price list of the product from highest to lowest.



Basic Query Operation- Aggregate

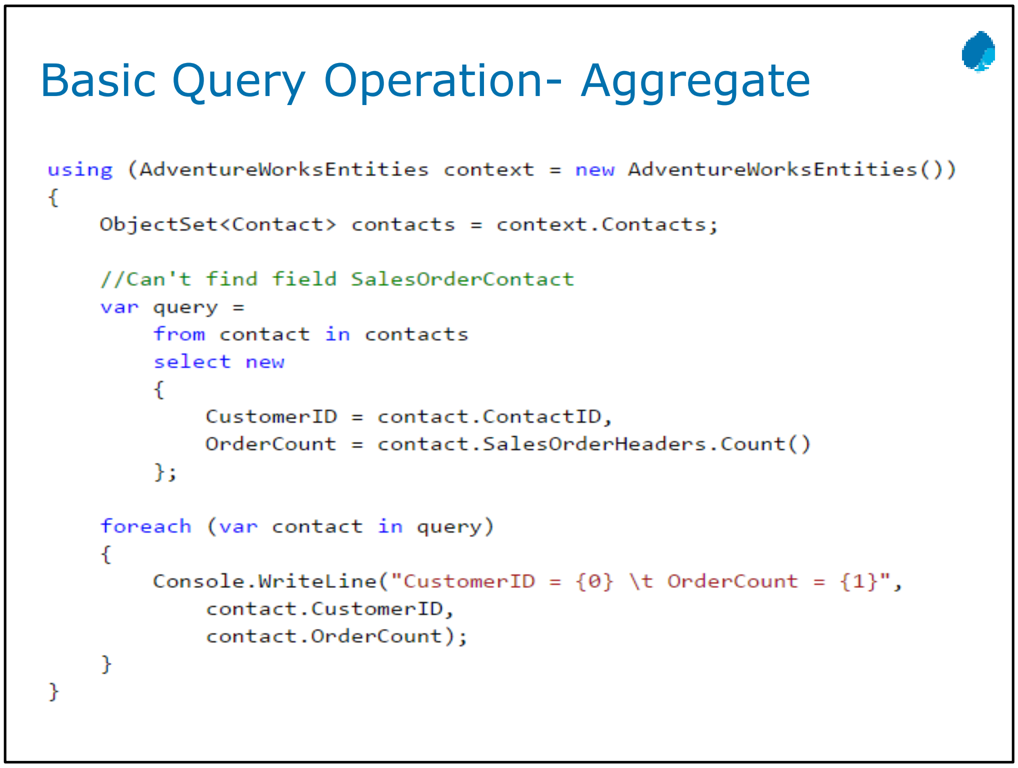
```
using (AdventureWorksEntities context = new AdventureWorksEntities())
{
    ObjectSet<Product> products = context.Products;

    var query = from product in products
                group product by product.Style into g
                select new
                {
                    Style = g.Key,
                    AverageListPrice =
                        g.Average(product => product.ListPrice)
                };

    foreach (var product in query)
    {
        Console.WriteLine("Product style: {0} Average list price: {1}",
            product.Style, product.AverageListPrice);
    }
}
```

The above example shows use of Average Aggregate function.

The query returns the average list price of the products of each style.



Basic Query Operation- Aggregate

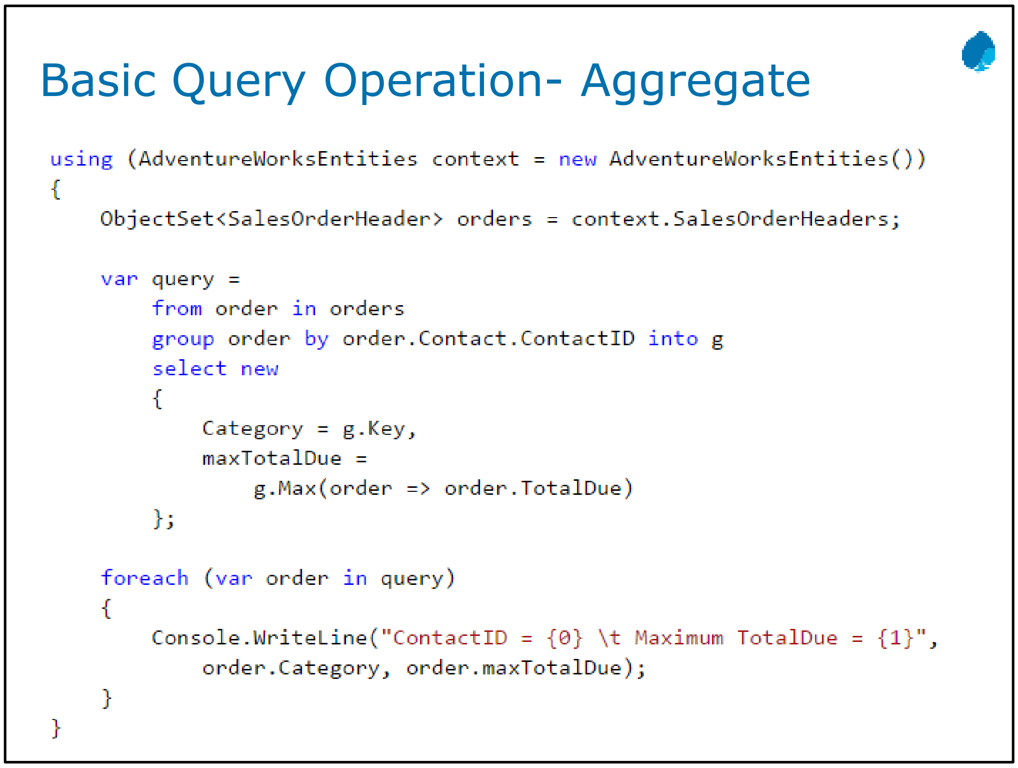
```
using (AdventureWorksEntities context = new AdventureWorksEntities())
{
    ObjectSet<Contact> contacts = context.Contacts;

    //Can't find field SalesOrderContact
    var query =
        from contact in contacts
        select new
        {
            CustomerID = contact.ContactID,
            OrderCount = contact.SalesOrderHeaders.Count()
        };

    foreach (var contact in query)
    {
        Console.WriteLine("CustomerID = {0} \t OrderCount = {1}",
            contact.CustomerID,
            contact.OrderCount);
    }
}
```

The above example shows use of Count Aggregate function.

The query return a list of contact IDs and how many orders each has.



Basic Query Operation- Aggregate

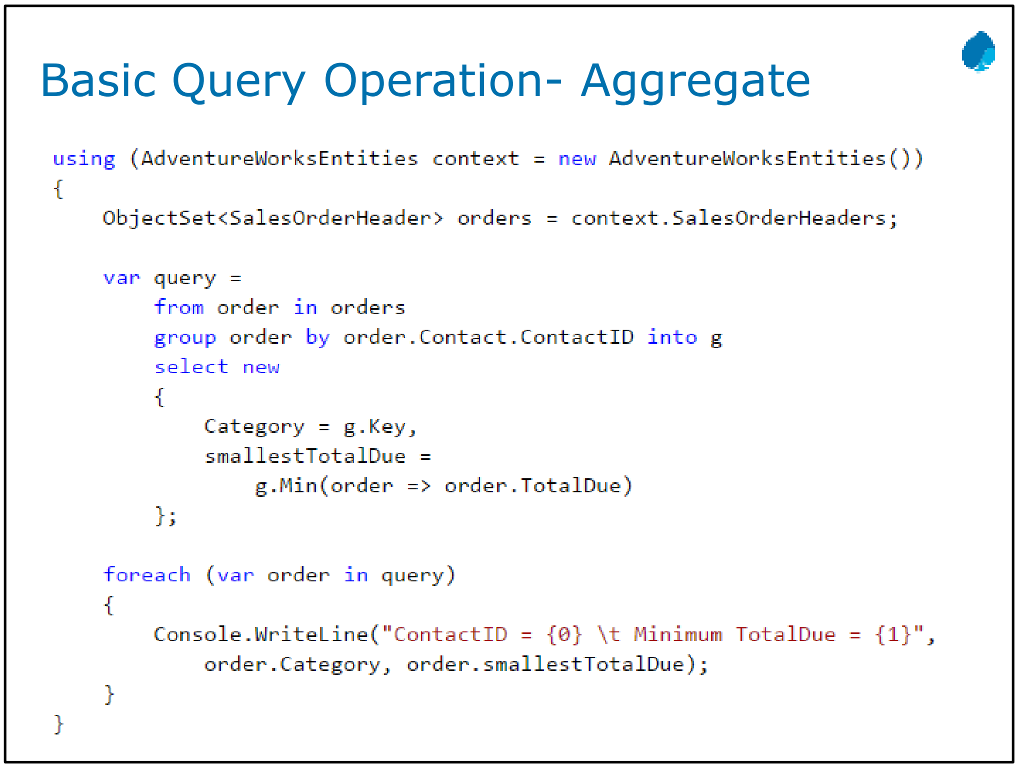
```
using (AdventureWorksEntities context = new AdventureWorksEntities())
{
    ObjectSet<SalesOrderHeader> orders = context.SalesOrderHeaders;

    var query =
        from order in orders
        group order by order.Contact.ContactID into g
        select new
        {
            Category = g.Key,
            maxTotalDue =
                g.Max(order => order.TotalDue)
        };

    foreach (var order in query)
    {
        Console.WriteLine("ContactID = {0} \t Maximum TotalDue = {1}",
            order.Category, order.maxTotalDue);
    }
}
```

The above example shows use of Max Aggregate function.

The query return the largest total due for each contact ID.



Basic Query Operation- Aggregate

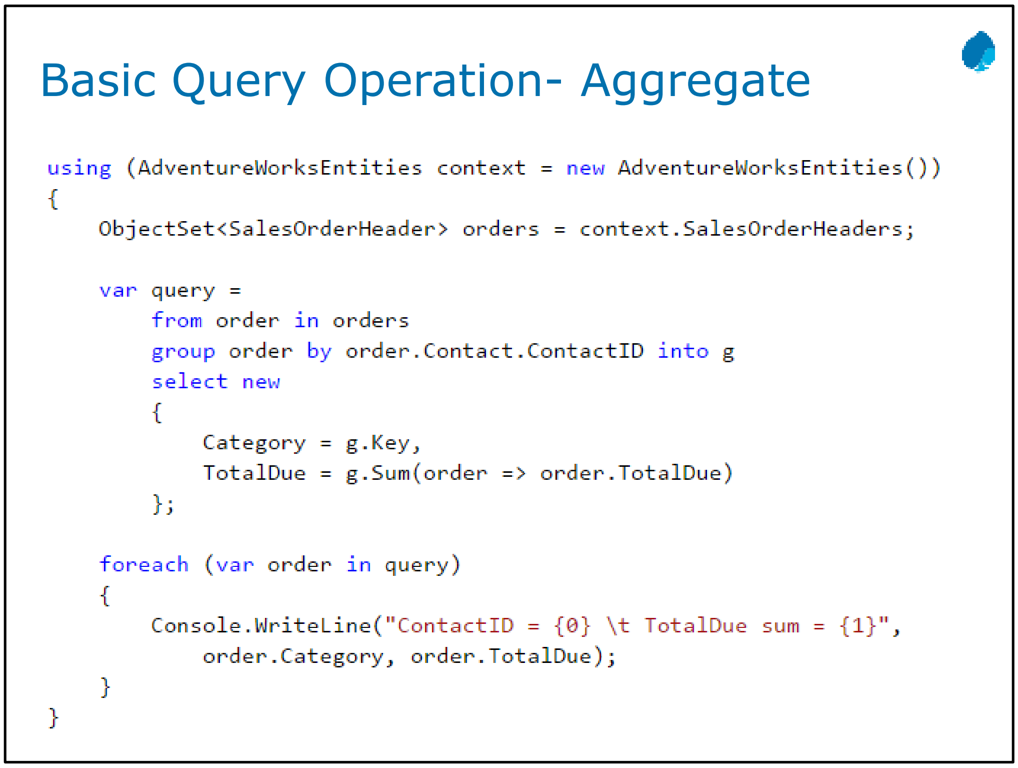
```
using (AdventureWorksEntities context = new AdventureWorksEntities())
{
    ObjectSet<SalesOrderHeader> orders = context.SalesOrderHeaders;

    var query =
        from order in orders
        group order by order.Contact.ContactID into g
        select new
        {
            Category = g.Key,
            smallestTotalDue =
                g.Min(order => order.TotalDue)
        };

    foreach (var order in query)
    {
        Console.WriteLine("ContactID = {0} \t Minimum TotalDue = {1}",
            order.Category, order.smallestTotalDue);
    }
}
```

The above example shows use of Min Aggregate function.

The query returns the smallest total due for each contact ID.



Basic Query Operation- Aggregate

```
using (AdventureWorksEntities context = new AdventureWorksEntities())
{
    ObjectSet<SalesOrderHeader> orders = context.SalesOrderHeaders;

    var query =
        from order in orders
        group order by order.Contact.ContactID into g
        select new
        {
            Category = g.Key,
            TotalDue = g.Sum(order => order.TotalDue)
        };

    foreach (var order in query)
    {
        Console.WriteLine("ContactID = {0} \t TotalDue sum = {1}",
            order.Category, order.TotalDue);
    }
}
```

The above example shows use of Sum Aggregate function.

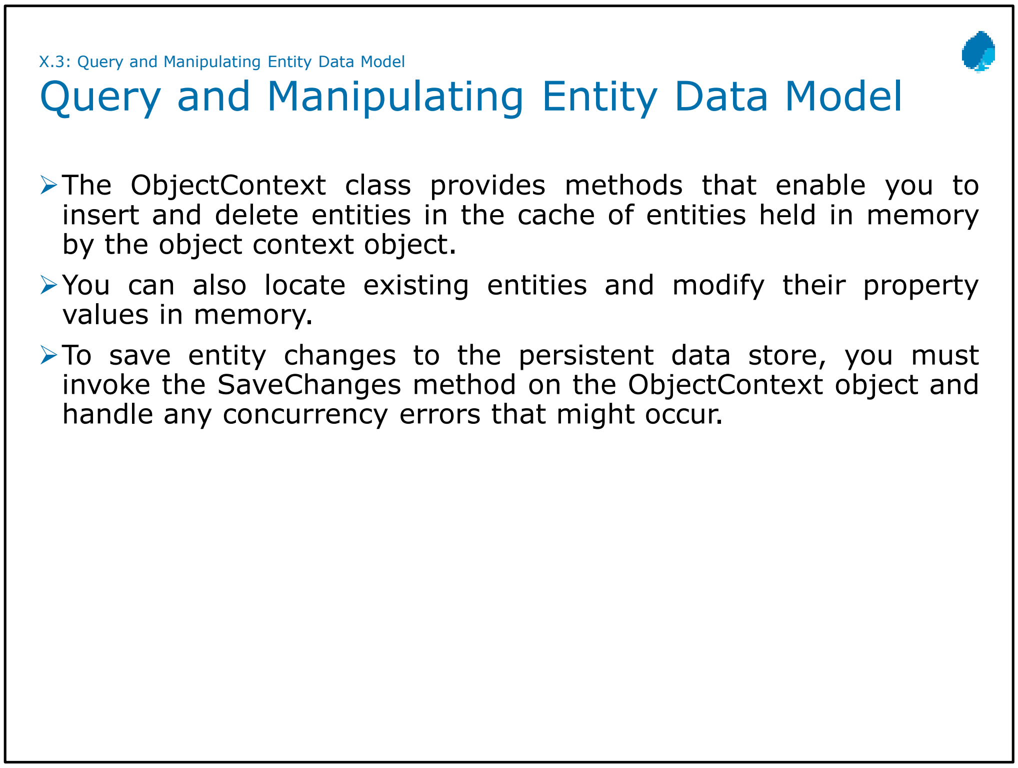
The query returns the total due for each contact ID.

X.X: Basic Query Operations

Demo

- Implementing Basic query operations in LINQ to Entities

Add the notes here.



X.3: Query and Manipulating Entity Data Model

Query and Manipulating Entity Data Model

- The ObjectContext class provides methods that enable you to insert and delete entities in the cache of entities held in memory by the object context object.
- You can also locate existing entities and modify their property values in memory.
- To save entity changes to the persistent data store, you must invoke the SaveChanges method on the ObjectContext object and handle any concurrency errors that might occur.

Query and Manipulating Entity Data Model

To insert an entity into an object context:

1. Create an entity and set its properties
2. Invoke the `Add` method on the `ObjectContext` object

To update an existing entity in an object context:

1. Locate the entity in the object context
2. Modify property values on the entity

To delete an existing entity in an object context:

1. Locate the entity in the object context
2. Invoke the `Remove` method on the `ObjectContext` object

To save the object context changes to the data store:

1.Invoke the `SaveChanges` method on the `ObjectContext` object

2. Handle concurrency exceptions

X.X: [Topic]

Demo

- Implementing Data Manipulation using LINQ to Entities

Add the notes here.

Best Practices for LINQ

- Best practices for LINQ query performance
  - Turn Object Tracking off
  - Turn Optimistic Concurrency off
  - Use Compiled queries judiciously
  - Using Data Context
  - Retrieve selective data
  - Analyze queries

Best Practices of LINQ query performance

<u>Turn Object Tracking off :-</u>

You should turn ObjectTrackingEnabled property off if you don't need it. Note that if don't need to change data but just read it, it is always advisable to turn this property off so as to turn off the unnecessary identity management of objects. The ObjectTrackingEnabled property is set to true by default. This implies that LINQ to SQL would keep track of every change that you make to your data so that it can remember those changes when you need to persist those changes to the underlying database at a later point in time. This will help boost the application's performance to a considerable extent. The following code snippet illustrates how you can turn this property off (set it to false):

```
using (IDGDataContext dataContext = new IDGDataContext())
{
dataContext.ObjectTrackingEnabled = false;
//Usual code
}
```

<u>Turn Optimistic Concurrency off :-</u>

Concurrency handling enables you to detect and resolve conflicts that arise out of concurrent requests to the same resource. Note that there are two types of concurrency - Optimistic and Pessimistic and LINQ follows an optimistic concurrency model by default. You should turn optimistic concurrency off unless it is needed. Note that UpdateCheck is set to Always which implies LINQ to SQL will check the property value, i.e., the value contained in that property against the value stored in the column of the database table that is mapped to that property. You should avoid using optimistic concurrency if not needed. You can use the UpdateCheck property to turn off optimistic concurrency. The following code snippet illustrates how you can set this property in the attribute of your entity class:

```
[Column(Storage="_Address", DbType="NText",
UpdateCheck=UpdateCheck.Never)]
```

Best Practices for LINQ (Contd…)

Use Compiled queries judiciously :-

You can take advantage of Compiled Query to boost query performance in your application. But, remember that compiled query could be costly when used for the first time. So, do ensure you use compiled queries only in situations where you need them, i.e., when you need a query to be used repeatedly.
At the time when a query is to be executed by the LINQ engine, LINQ to SQL translates the LINQ queries to SQL -- this is repeated every time the query is to be executed. This involves traversing the expression tree recursively again and hence it is a performance overhead. No worries at all -- you have the CompiledQuery class for the rescue
You can leverage CompiledQuery to eliminate this performance overhead for queries that need to be executed again and again. A word of caution though: Use CompiledQuery judiciously and only when it is needed.

Using Data Context :-

You should not dump all the database objects into one single DataContext. The DataContext should represent one unit of work. This approach if followed would reduce the identity management and object tracking overhead involved. Also, you should only attach the objects to your data context those have been changed since the time they were read into the memory.

Best Practices for LINQ (Contd…)

Retrieve selective data :-

You should retrieve only the data that is needed and avoid retrieving all of the data. To achieve this, you can take advantage of the Take and Skip methods. You can also filter the data to be retrieved using DataLoadOptions.AssociateWith so that only the required data is returned. The following code snippet illustrates how DataLoadOptions.AssociateWith can be used.

```
using (IDGDataContext dataContext = new IDGDataContext())
{
DataLoadOptions dataLoadOptions = new DataLoadOptions();
dataLoadOptions.AssociateWith<Customer>(customer =>
customer.Address.Where<Address>(address => address.PinCode == 500016));
dataContext.LoadOptions = dataLoadOptions;
}
```

Analyze queries :-

You should always analyze your LINQ queries and take a monitor the generated SQL. You can set the Log property of the data context to see the generated SQL.

```
using (IDGDataContext dataContext = new IDGDataContext())
{
dataContext.Log = Console.Out;
}
```

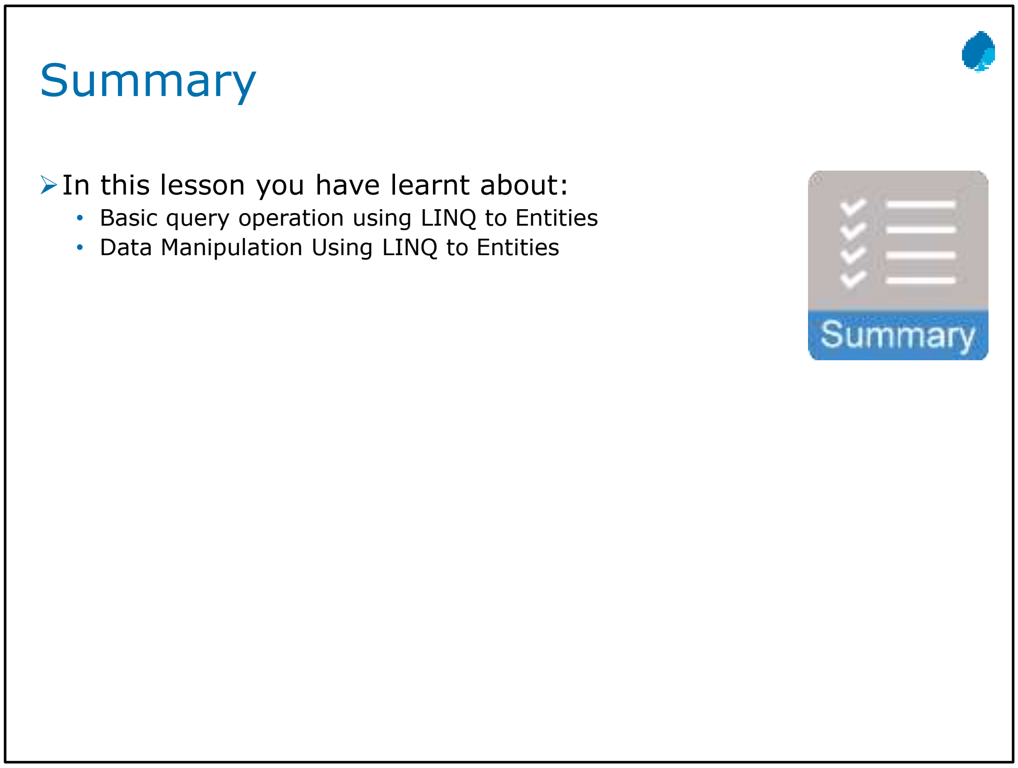
This will help you to understand how your LINQ query has been translated to SQL and any additional columns or extra data that is retrieved when the SQL query is eventually executed.

X.X: [Topic]

Lab

- Lab Topic

Add the notes here.



Summary

➢ In this lesson you have learnt about:
- Basic query operation using LINQ to Entities
- Data Manipulation Using LINQ to Entities

Add the notes here.

Review Question

➢ Which of the following allows to access Entities using declarative query syntax?
- LINQ to Array
- LINQ to SQL
- LINQ to DataSet
- LINQ to Entities

➢ ObjectContext Represents a typed query against a conceptual model in a given object context.
- True
- False

➢ Question 3: Fill in the Blanks

___________ method allow to save change to database
- SafeChanges()
- SaveChanges()
- AcceptChnage()
- ImplementChnages()

Add the notes here.