LINQ & EF Core

Lesson 04 : CRUD Operations in EF CORE

Lesson Objectives

- In this lesson we will cover the following topic
  - LINQ to Entities
  - DbContext Class
  - Migrations in EF Core
  - CRUD Operation in EF Core
  - Transaction in EF & EF Core

LINQ to Entities Overview

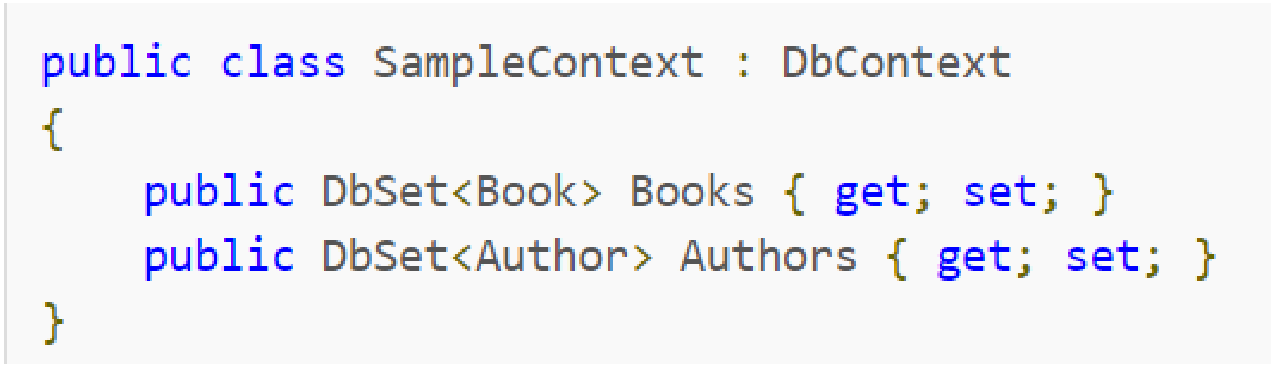
- LINQ to Entities provides LINQ support that enables developer to write queries against the Entity Framework & EF Core
- Queries against the Entity Framework are represented by command tree queries which against the object context
- LINQ to Entities converts LINQ queries to command tree queries, executes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

DbContext Overview:

- DbContext class is the brain of **Entity Framework Core.**
- It is an Integral part of Entity Framework.
- An instance of DbContext represents a session with the database which can be used to query and save instances of your entities to a database.
- DbContext is a combination of the Unit Of Work and Repository patterns.
- DbContext in EF Core allows us to perform following tasks:
  - Manage database connection
  - Configure model & relationship
  - Querying database
  - Saving data to the database
  - Configure change tracking
  - Caching
  - Transaction management

EF Core DbSet:

- DbSet<TEntity> class represents a collection for a given entity within the model and is the gateway to database operations against an entity.
- DbSet<TEntity> classes are added as properties to the
- DbContext and are mapped by default to database tables that take the name of the DbSet<TEntity> property.
- DbContext is a combination of the Unit Of Work and Repository patterns.



```
public class SampleContext : DbContext
{
    public DbSet<Book> Books { get; set; }
    public DbSet<Author> Authors { get; set; }
}
```

Using DBContext in our Class

- In order to use the features of DbContext class, your application context class has to derive from it.

```
public class CompanyContext : DbContext
{

}
```

- Add DbSet for the identified entity class

```
public class CompanyContext : DbContext
{
    public DbSet<Department> Department { get; set; }
    public DbSet<Employee> Employee { get; set; }
}
```

Using DBContext in our Class

- The OnConfiguring() method of the DbContext class is overridden inside the context class.
- It allows to select and configure the data source to be used with a context using DbContextOptionsBuilder which is passed to its parameter.

```
public class CompanyContext : DbContext
{
    public DbSet<Department> Department { get; set; }
    public DbSet<Employee> Employee { get; set; }

    protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)
    {
        if (!optionsBuilder.IsConfigured)
        {
          optionsBuilder.UseSqlServer(@"Server=vaio;Database=Company;Trusted_Connection=True;");
        }
    }
}
```

DbContext Methods:

Method	Description
Add	**Adds a new entity with Added state**
AddRange	**Adds a collection of new entities with Added state**
Attach	**Attaches a new or existing entity with Unchanged state**
AttachRange	**Attaches a collection of new or existing entity with Unchanged state**
Remove	**Attaches an entity with Deleted state**
RemoveRange	**Attaches a collection of entities with Deleted state**
Update	**Attaches disconnected entity with Modified state**
UpdateRange	**Attaches collection of disconnected entity with Modified state**
SaveChanges	**Execute INSERT, UPDATE or DELETE command to the database for the entities with Added, Modified or Deleted state.**

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

Loading Related Data

- Entity Framework Core allows you to use the navigation properties in your model to load related entities.
- There are three common O/RM patterns used to load related data.
  - Eager Loading
  - Explicit Loading
  - Lazy Loading
- **Eager Loading** means that the related data is loaded from the database as part of the initial query. We can use the **Include()** method to specify related data to be included in query results
- **Explicit loading** means that the related data is explicitly loaded from the database at a later time. Here the **Load()** method is used to load related entity explicitly. If you want to filter the related entity before loading then use the **Query()** method.
- **Lazy loading** means that the related data is transparently loaded from the database when the navigation property is accessed. Lazy Loading was introduced in EF Core 2.1 as an opt-in feature. Lazy loading can be enabled in two ways:
  - Using Proxies
  - Using the ILazyLoader service

Querying data via the DbSet

- Data querying in Entity Framework Core is performed against the DbSet properties of the DbContext.
- The DbSet represents a collection of entities of a specific type - the type specified by the type parameter.
- The EF Core provider that you use is responsible for translating the LINQ query into the actual SQL to be executed against the database.

```
var data = from a in Authors select a orderby a.LastName
```

- Using Lambda Expression

```
var data = context.Authors.OrderBy(a => a.LastName);
```

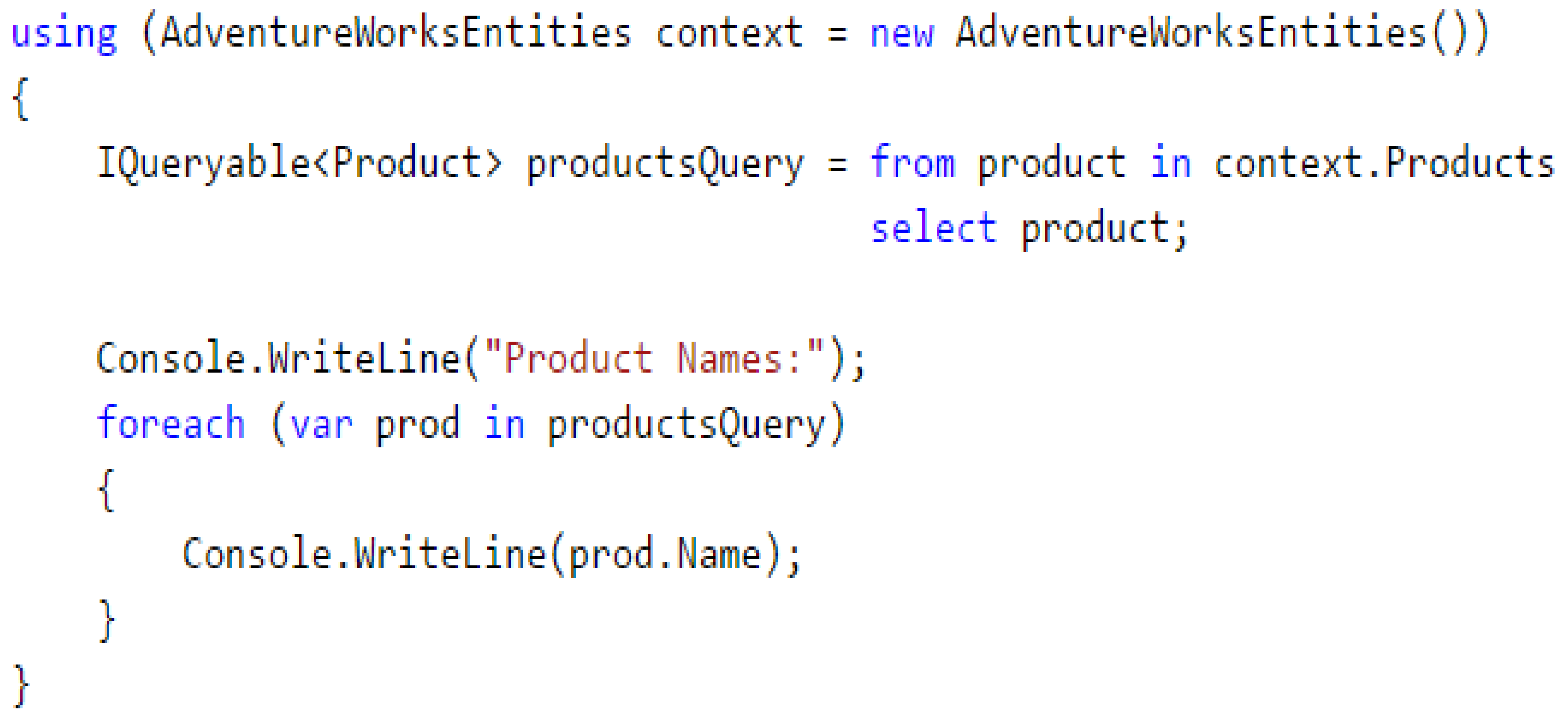
- Using First & FirstOrDefault()

```
var author = context.Authors.First();
```

- Resulting SQL:

```
SELECT TOP(1) [a].[AuthorId], [a].[FirstName], [a].[LastName]
FROM [Authors] AS [a]
```

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);
    }
}
```

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);
    }
}
```

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);
    }
}
```

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);
    }
}
```

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);
    }
}
```

Basic Query Operation- Aggregate

```csharp
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);
    }
}
```

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);
    }
}
```

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);
    }
}
```

Basic Query Operation- Aggregate

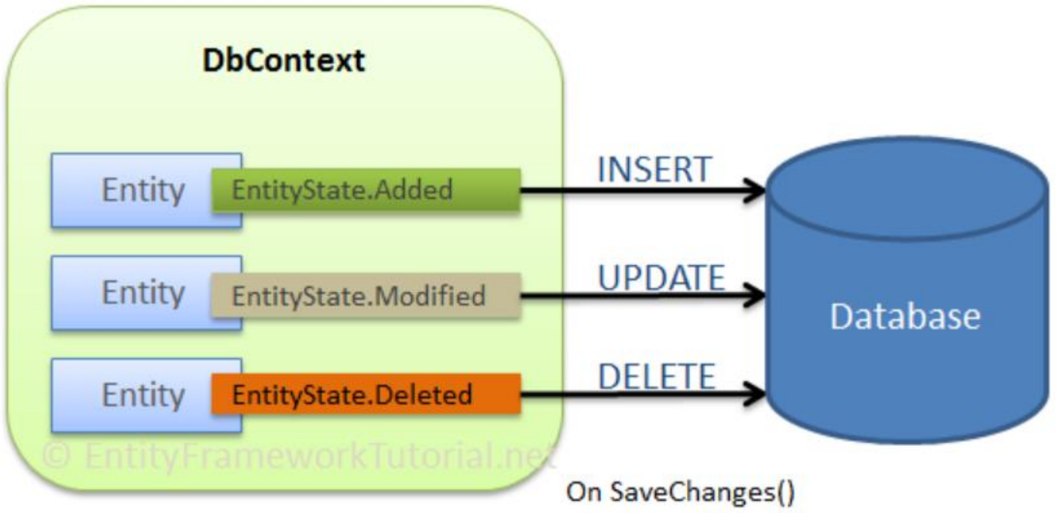
```
using (AdventureWorksEntities context = new AdventureWorksEntities())
{
    ObjectSet<SalesOrderHeader> orders = context.SalesOrderHeaders;

    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);
    }
}
```

Demo

- Implementing Basic query operations in LINQ to Entities

EF Core DML Implementation

- Entity Framework Core provides different ways to add, update, or delete data in the underlying database.
- An entity contains data in its scalar property will be either inserted or updated or deleted based on its EntityState.
- 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 DbContext object and handle any concurrency errors that might occur.



Inserting Record:

- The context begins tracking the entity that was passed in to the Add method and applies an EntityState value of Added to it.
- The context also applies the same EntityState value of Added to all other objects in the object graph that aren't already being tracked by the context.
- Key Methods for adding Record:
  - Add<TEntity>(TEntity entity)
  - AddRange<TEntity>(IEnumerable<TEntity> entities)
  - AddRange<TEntity>(params TEntity[] entities)
- The context also applies the same EntityState value of Added to all other objects in the object graph that aren't already being tracked by the context.

```
var author = new Author{ FirstName = "William", LastName = "Shakespeare" };
context.Authors.Add<Author>(author);
```

```
var books = new List<Book> {
    new Book { Title = "It", Author = author },
    new Book { Title = "Carrie", Author = author },
    new Book { Title = "Misery", Author = author }
};
context.Books.AddRange(books);
context.SaveChanges();
```

Updating Record:

- When we alter property values on a tracked entity, the context changes the EntityState for the entity to Modified and the ChangeTracker records the old property values and the new property values.
- When SaveChanges is called, an UPDATE statement is generated and executed by the database.

```
var author = context.Authors.Find(1);
author.FirstName = "Bill";
context.SaveChanges();
```

- Equivalent SQL Statement:

```
exec sp_executesql N'SET NOCOUNT ON;
UPDATE [Authors] SET [FirstName] = @p0
WHERE [AuthorId] = @p1;
SELECT @@ROWCOUNT;
',N'@p1 int,@p0 nvarchar(4000)',@p1=1,@p0=N'Bill'
```

Deleting Record:

- When we deleting record, the entity to be deleted is obtained by the context, so the context begins tracking it immediately.
- The DbSet<T>.Remove method results in the entity's EntityState being set to Deleted.

```
context.Authors.Remove(context.Authors.Find(1));
context.SaveChanges();
```

- When SaveChanges is called, a DELETE statement is generated and executed by the database.

```
exec sp_executesql N'SET NOCOUNT ON;
DELETE FROM [Authors]
WHERE [AuthorId] = @p0;
SELECT @@ROWCOUNT;
',N'@p0 int',@p0=1
```

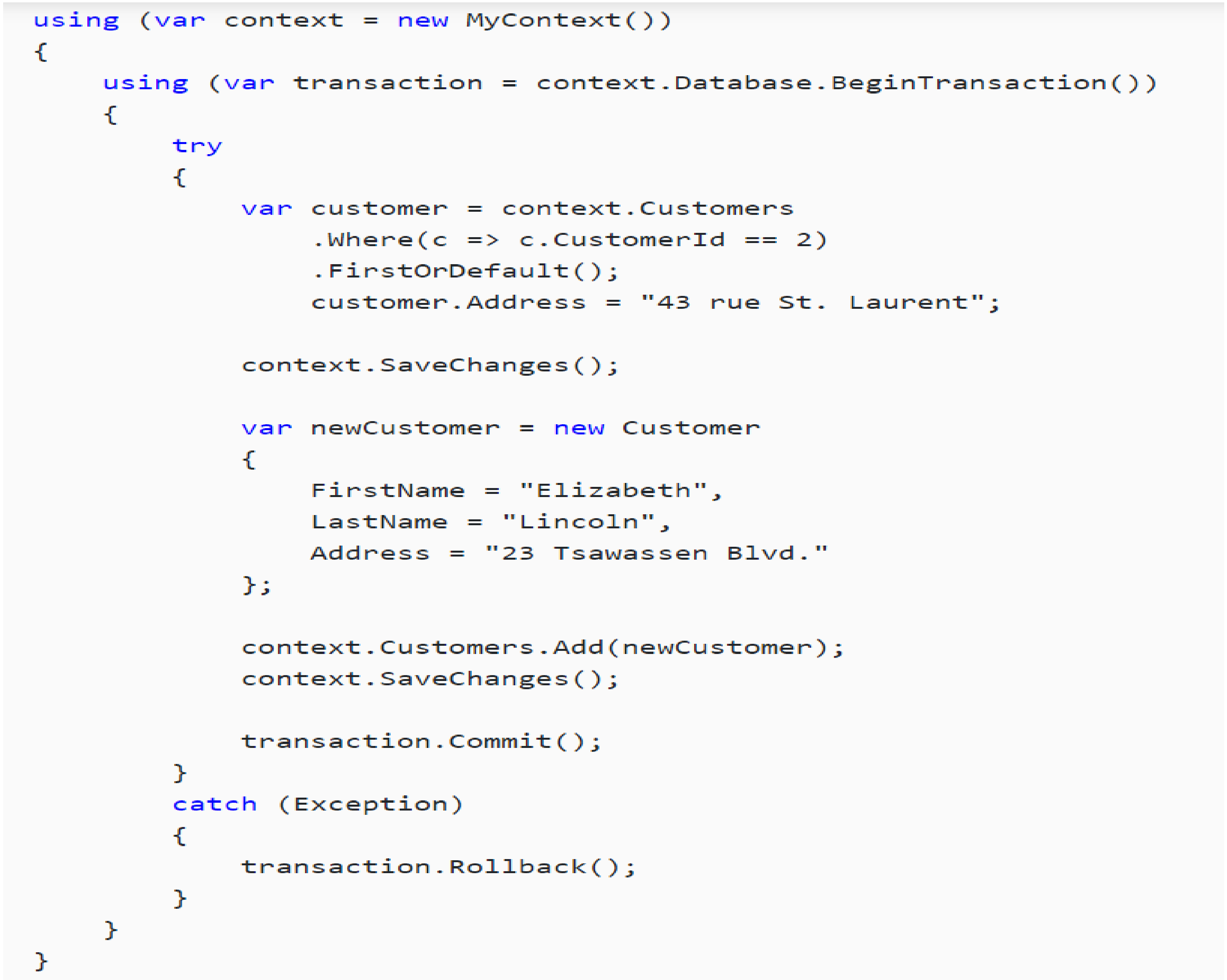
- Example:

```
var context = new SampleContext();
var author = new Author { AuthorId = 1 };
context.Authors.Remove(author);
context.SaveChanges();
```

Transaction in EF6 & EF Core

- Transactions allow several database operations to be processed in an atomic manner.
- If the transaction is committed, all of the operations are successfully applied to the database.
- If the transaction is rolled back, none of the operations are applied to the database.
- Entity Framework internally maintains transactions when the SaveChanges() method is called or when you call SaveChanges() to insert, delete, or update data to the database, the entity framework will wrap that operation in a transaction.
- In EF 6 and EF Core, you can use multiple SaveChanges within a single transaction. You can use the DbContext.Database API to begin, commit, and rollback transactions.

Transaction in EF & EF Core Demo



```
using (var context = new MyContext())
{
    using (var transaction = context.Database.BeginTransaction())
    {
        try
        {
            var customer = context.Customers
                .Where(c => c.CustomerId == 2)
                .FirstOrDefault();
                customer.Address = "43 rue St. Laurent";

            context.SaveChanges();

            var newCustomer = new Customer
            {
                FirstName = "Elizabeth",
                LastName = "Lincoln",
                Address = "23 Tsawassen Blvd."
            };

            context.Customers.Add(newCustomer);
            context.SaveChanges();

            transaction.Commit();
        }
        catch (Exception)
        {
            transaction.Rollback();
        }
    }
}
```

Demo

- Implementing Data Manipulation in EF Core

Lab

- Querying Data using LINQ to Entity
- DML Manipulation in EF Core

Summary

- In this lesson you have learnt about:
  - LINQ to Entities
  - DbContext Class
  - Migrations in EF Core
  - CRUD Operation in EF Core
  - Transaction in EF & EF Core

Review Question

- Which of the following allows to access Entities using declarative query syntax?
  - LINQ to XML
  - LINQ to Table
  - LINQ to DataSet
  - LINQ to Entities

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

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