Acesso a dados .NET Core

Data Access

Net SDK Core 3.0

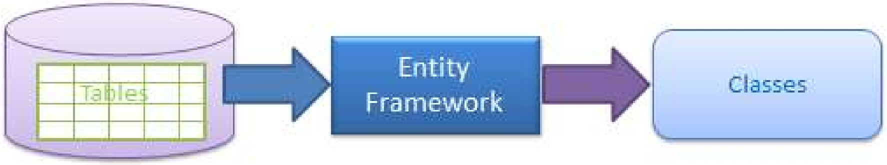
Part 2 - Objectives

- What is Entity Framework (EF Core)
- EF Architecture
- Types of entities
- Major elements

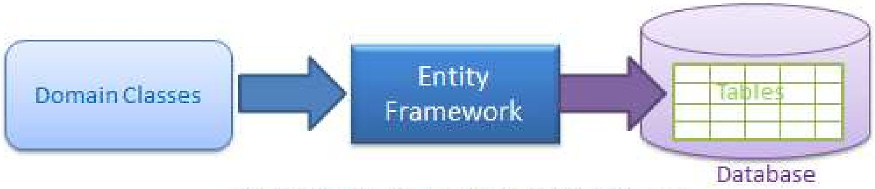
What is Entity Framework

- Entity framework is an Object/Relational Mapping (O/RM) framework. It is an enhancement to ADO.NET that gives developers an automated mechanism for accessing and storing the data in the database.
- The Entity Framework is a full solution that works with any data source, even flat-file and hierarchical databases.
- Entity framework is useful in three scenarios:
  - First, if you already have existing database or you want to design your database ahead of other parts of the application.
  - Second, you want to focus on your domain classes and then create the database from your domain classes.
  - Third, you want to design your database schema on the visual designer and then create the database and classes.

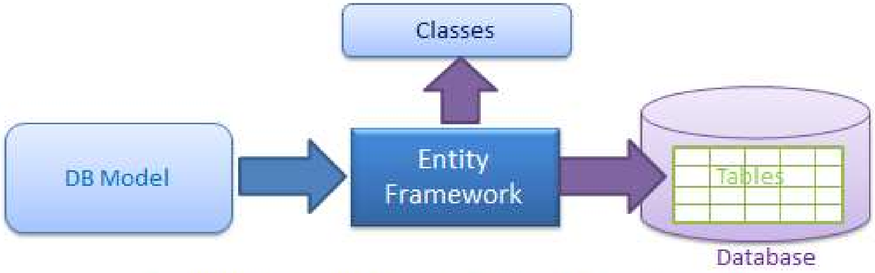
What is Entity Framework



Generate Data Access Classes for Existing Database



Create Database from the Domain Classes



Create Database and Classes from the DB Model design

Defining Entity

An entity is the data associated with a particular object when considered from the perspective of a particular application.

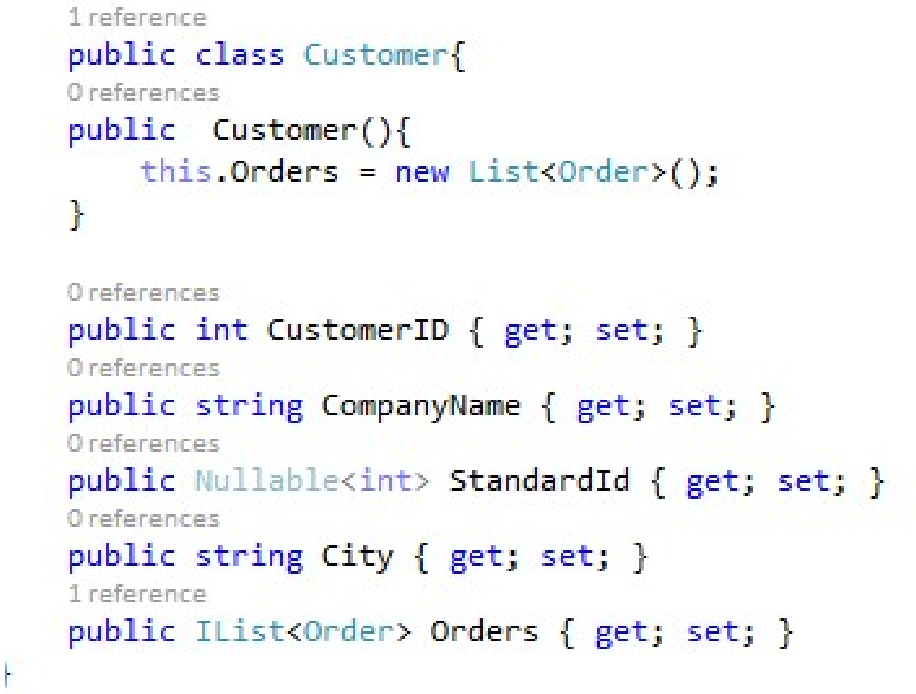
For example, a customer object will include a customer’s name, address, telephone number, company name, and so on.

The actual customer object may have more data than this associated with it, but from the perspective of this particular application, the customer object is complete by knowing these facts.

Types of Entity in Entity Framework

- **There are two types of Entities in Entity Framework 5.0/6.0: POCO entity and dynamic proxy entity.**
- **POCO Entity (Plain Old CLR Object):**
  - POCO class is the class that doesn't depend on any framework specific base class. It is like any other normal .net class which is why it is called "Plain Old CLR Objects".
  - These POCO entities (also known as persistence-ignorant objects) support most of the same query, insert, update, and delete behaviors as entity types that are generated by the Entity Data Model. The following is an example of Customer POCO entity.

POCO Entity

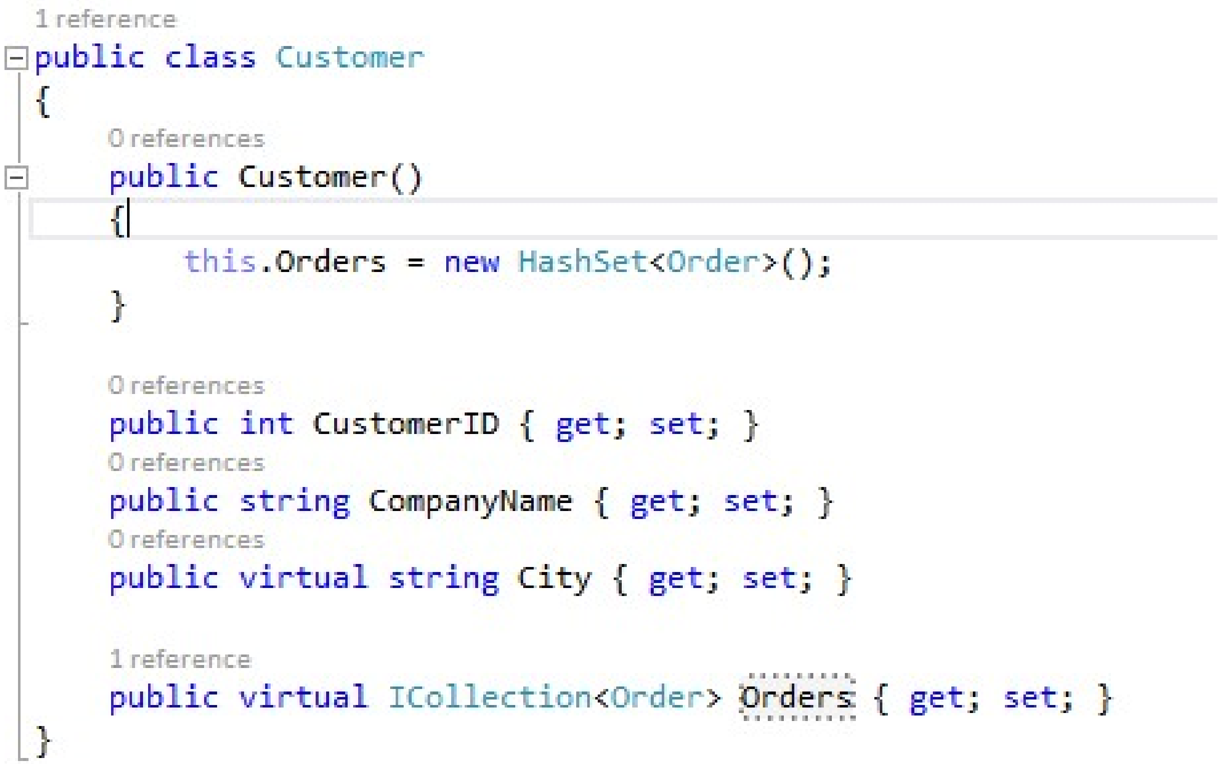


```
 1 reference
 public class Customer{
 0 references
 public  Customer(){
     this.Orders = new List<Order>();
 }

 0 references
 public int CustomerID { get; set; }
 0 references
 public string CompanyName { get; set; }
 0 references
 public Nullable<int> StandardId { get; set; }
 0 references
 public string City { get; set; }
 1 reference
 public IList<Order> Orders { get; set; }
}
```

Types of Entity in Entity Framework

- **Dynamic Proxy (POCO Proxy):**
  - Dynamic Proxy is a runtime proxy class of POCO entity. It is like a wrapper class of POCO entity. Dynamic proxy entities allow lazy loading and automatic change tracking.
  - POCO entity should meet the following requirements to become a POCO proxy:
    - A POCO class must be declared with public access.
    - A POCO class must not be sealed.
    - A POCO class must not be abstract.
  - Each navigation property must be declared as public, virtual
  - Each collection property must be ICollection<T>
  - ProxyCreationEnabled option must NOT be false (default is true) in context class

Dynamic Entity



```
1 reference
public class Customer
{
    0 references
    public Customer()
    {
        this.Orders = new HashSet<Order>();
    }

    0 references
    public int CustomerID { get; set; }
    0 references
    public string CompanyName { get; set; }
    0 references
    public virtual string City { get; set; }

    1 reference
    public virtual ICollection<Order> Orders { get; set; }
}
```

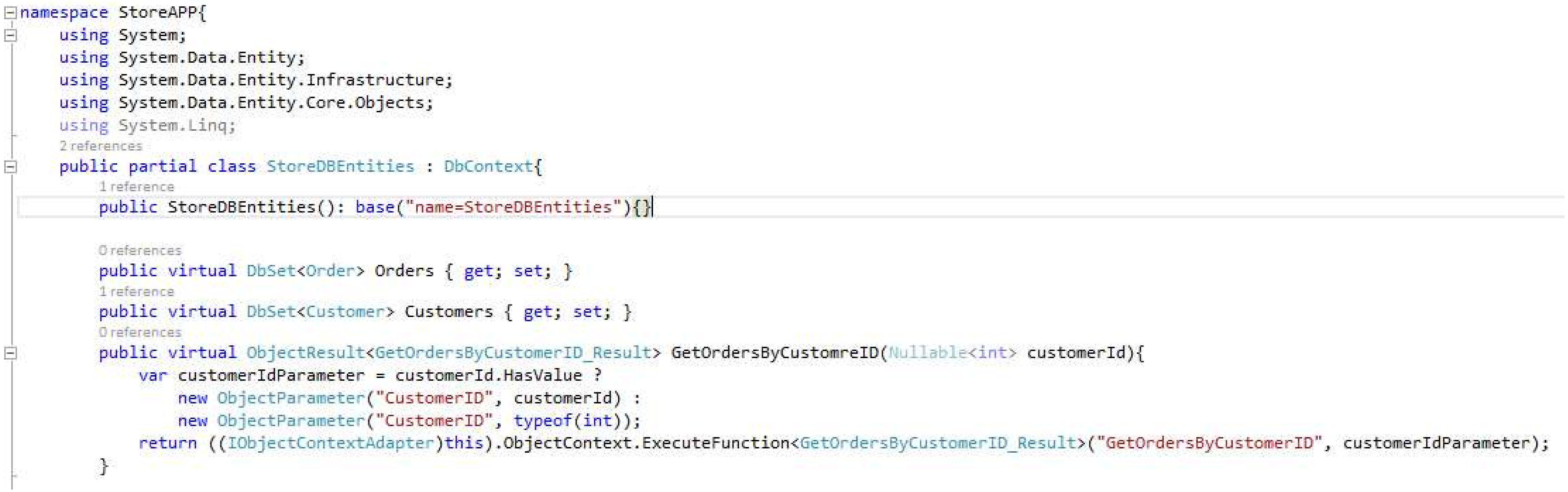
Types of Entity in Entity Framework

- **Entity can have two types of properties, Scalar and Navigation properties.**
  - – Scalar properties:
    - Scalar properties are properties whose actual values are contained in the entity. For example, Customer entity has scalar properties like CustomerID and CompanyName. These correspond with the Customer table columns.
  - – Navigation properties:
    - Navigation properties are pointers to other related entities. The Customer has Orders property as a navigation property that will enable the application to navigate from a Customer to related Orders entity.

DbContext Class

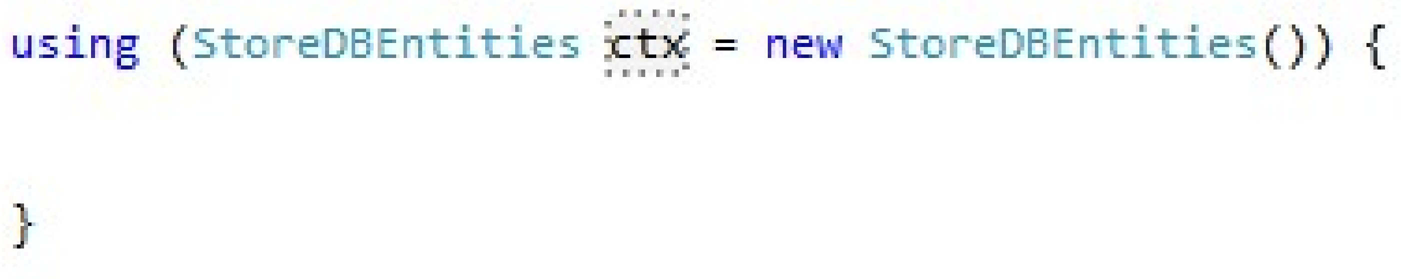
- DbContext is an important part of Entity Framework. It is a bridge between your domain or entity classes and the database.
- DbContext is the primary class that is responsible for interacting with data as object. DbContext is responsible for the following activities:
  - EntitySet: DbContext contains entity set (DbSet<TEntity>) for all the entities which is mapped to DB tables.
  - Querying: DbContext converts LINQ-to-Entities queries to SQL query and send it to the database.
  - Change Tracking: It keeps track of changes that occurred in the entities after it has been querying from the database.
  - Persisting Data: It also performs the Insert, Update and Delete operations to the database, based on what the entity states.
  - Caching: DbContext does first level caching by default. It stores the entities which have been retrieved during the life time of a context class.
  - Manage Relationship: DbContext also manages relationship using CSDL, MSL and SSDL in DB-First or Model-First approach or using fluent API in Code-First approach.
  - Object Materialization: DbContext converts raw table data into entity objects.

DbContext Class



```
namespace StoreAPP{
    using System;
    using System.Data.Entity;
    using System.Data.Entity.Infrastructure;
    using System.Data.Entity.Core.Objects;
    using System.Linq;
    2 references
    public partial class StoreDBEntities : DbContext{
        1 reference
        public StoreDBEntities(): base("name=StoreDBEntities"){}

        0 references
        public virtual DbSet<Order> Orders { get; set; }
        1 reference
        public virtual DbSet<Customer> Customers { get; set; }
        0 references
        public virtual ObjectResult<GetOrdersByCustomerID_Result> GetOrdersByCustomreID(Nullable<int> customerId){
            var customerIdParameter = customerId.HasValue ?
                new ObjectParameter("CustomerID", customerId) :
                new ObjectParameter("CustomerID", typeof(int));
            return ((IObjectContextAdapter)this).ObjectContext.ExecuteFunction<GetOrdersByCustomerID_Result>("GetOrdersByCustomerID", customerIdParameter);
        }
```

Instantiating DbContext

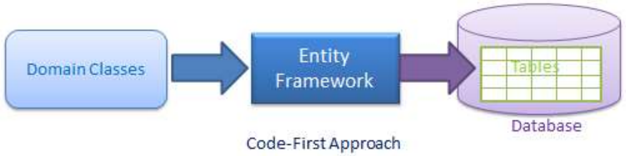


```
using (StoreDBEntities ctx = new StoreDBEntities()) {


}
```

CODE FIRST

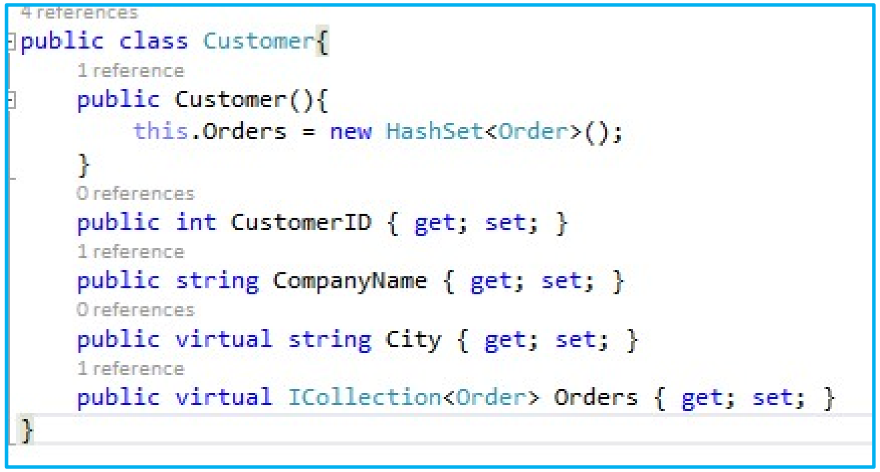
Code First Development

- In the Code First approach, the POCO classes are first created and then create database from these POCO classes.

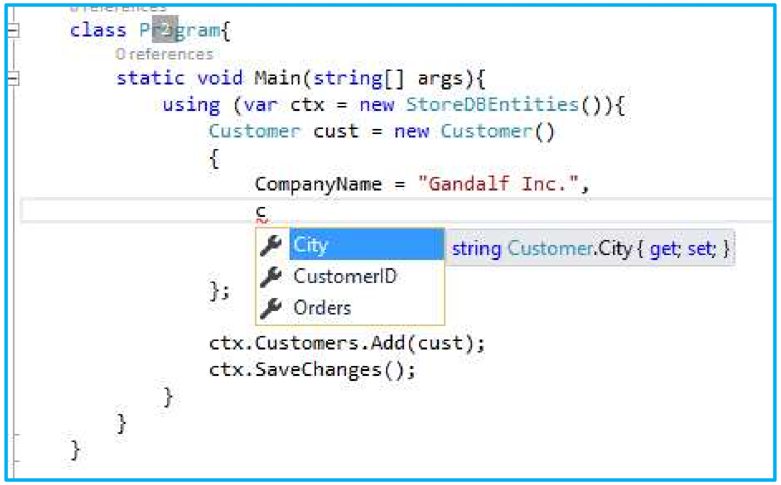


- Write application domain classes and context class→ configure domain classes for additional mapping requirements → Hit F5 to run the application → Code First API creates new database or map existing database with domain classes → Seed default/test data into the database → Finally launches the application

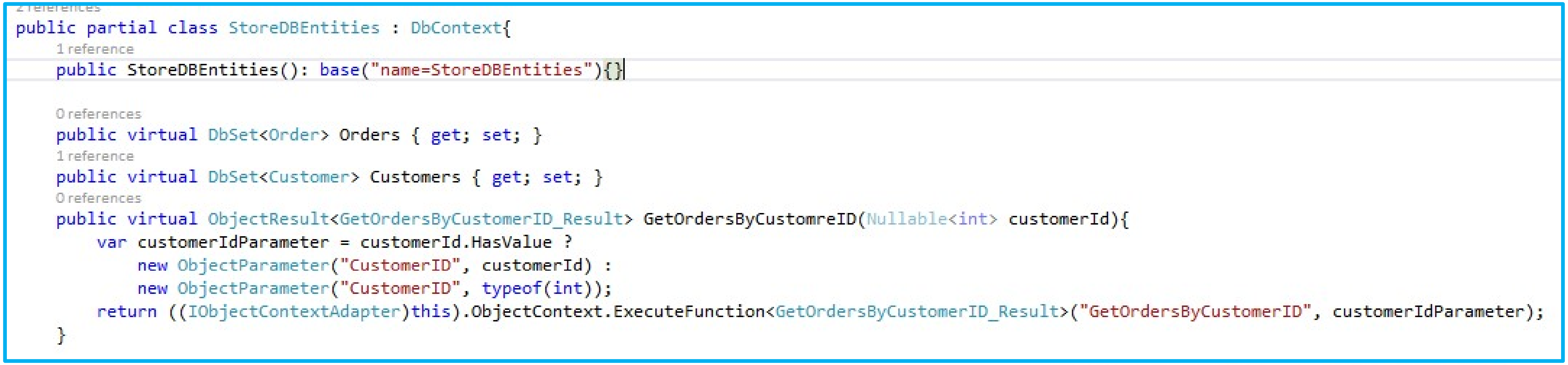
Code First Classes



```
4 references
public class Customer{
    1 reference
    public Customer(){
        this.Orders = new HashSet<Order>();
    }
    0 references
    public int CustomerID { get; set; }
    1 reference
    public string CompanyName { get; set; }
    0 references
    public virtual string City { get; set; }
    1 reference
    public virtual ICollection<Order> Orders { get; set; }
}
```



```
0 references
class Program{
    0 references
    static void Main(string[] args){
        using (var ctx = new StoreDBEntities()){
            Customer cust = new Customer()
            {
                CompanyName = "Gandalf Inc.",
                C
            };
            ctx.Customers.Add(cust);
            ctx.SaveChanges();
        }
    }
}
```

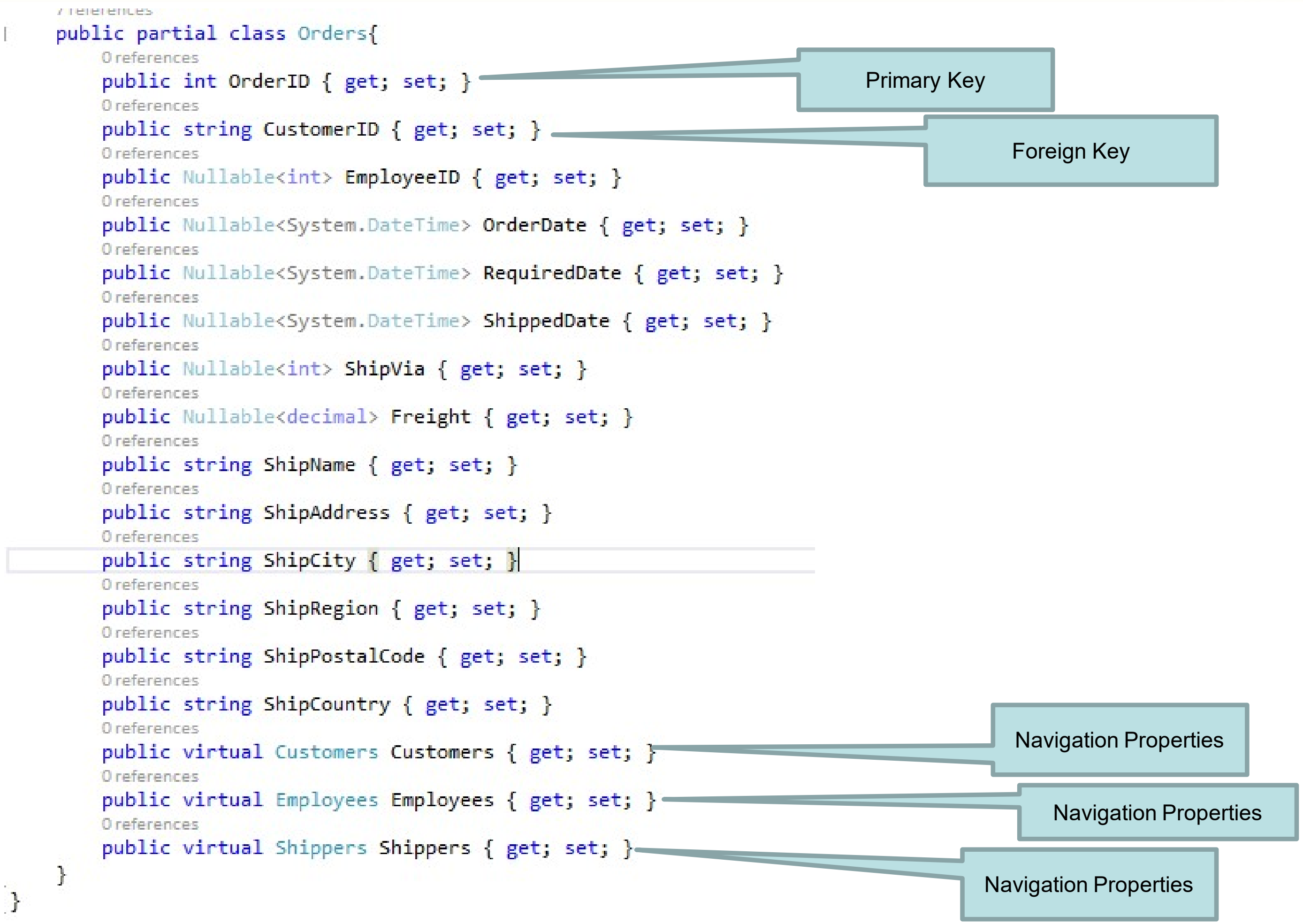


```
2 references
public partial class StoreDBEntities : DbContext{
    1 reference
    public StoreDBEntities(): base("name=StoreDBEntities"){}

    0 references
    public virtual DbSet<Order> Orders { get; set; }
    1 reference
    public virtual DbSet<Customer> Customers { get; set; }
    0 references
    public virtual ObjectResult<GetOrdersByCustomerID_Result> GetOrdersByCustomreID(Nullable<int> customerId){
        var customerIdParameter = customerId.HasValue ?
            new ObjectParameter("CustomerID", customerId) :
            new ObjectParameter("CustomerID", typeof(int));
        return ((IObjectContextAdapter)this).ObjectContext.ExecuteFunction<GetOrdersByCustomerID_Result>("GetOrdersByCustomerID", customerIdParameter);
    }
```

Conventions

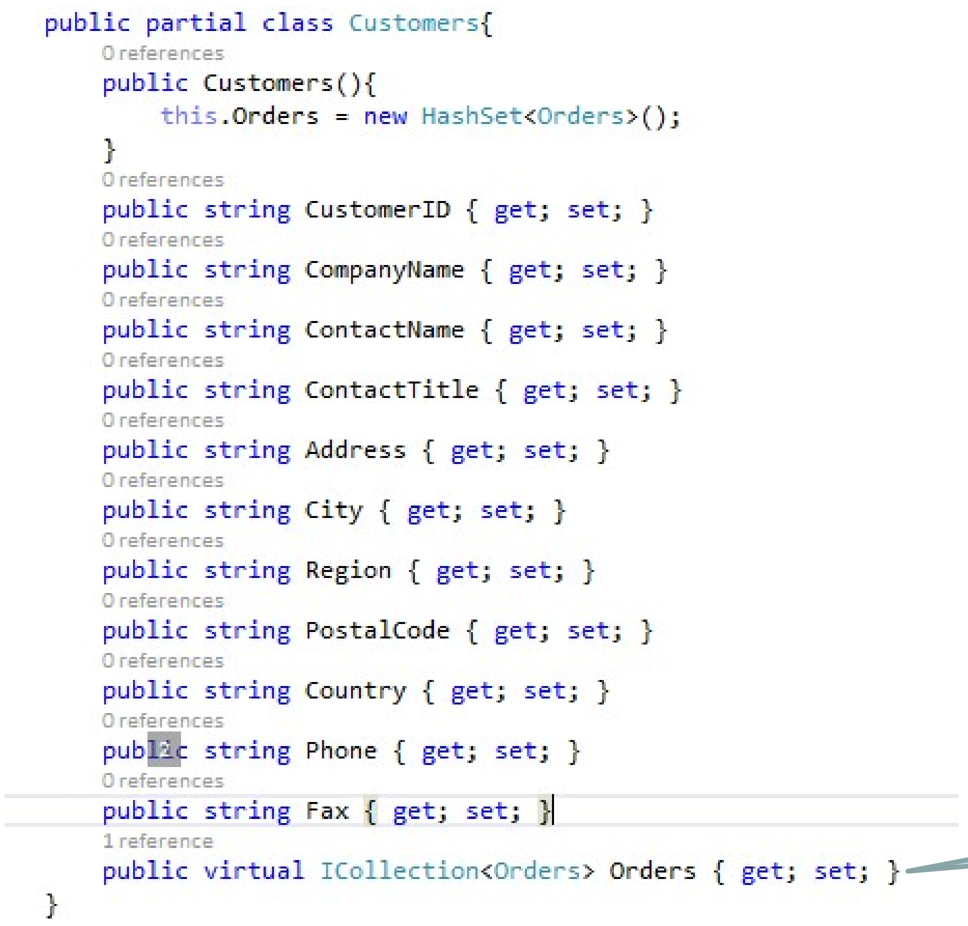
- **Entity Primary Key**
- **Entity Relationship**
- **Foreign key Convention**

Type Dicovery example



```
7 references
public partial class Orders{
    0 references
    public int OrderID { get; set; }
    0 references
    public string CustomerID { get; set; }
    0 references
    public Nullable<int> EmployeeID { get; set; }
    0 references
    public Nullable<System.DateTime> OrderDate { get; set; }
    0 references
    public Nullable<System.DateTime> RequiredDate { get; set; }
    0 references
    public Nullable<System.DateTime> ShippedDate { get; set; }
    0 references
    public Nullable<int> ShipVia { get; set; }
    0 references
    public Nullable<decimal> Freight { get; set; }
    0 references
    public string ShipName { get; set; }
    0 references
    public string ShipAddress { get; set; }
    0 references
    public string ShipCity { get; set; }
    0 references
    public string ShipRegion { get; set; }
    0 references
    public string ShipPostalCode { get; set; }
    0 references
    public string ShipCountry { get; set; }
    0 references
    public virtual Customers Customers { get; set; }
    0 references
    public virtual Employees Employees { get; set; }
    0 references
    public virtual Shippers Shippers { get; set; }
}
}
```

Type Discovery example



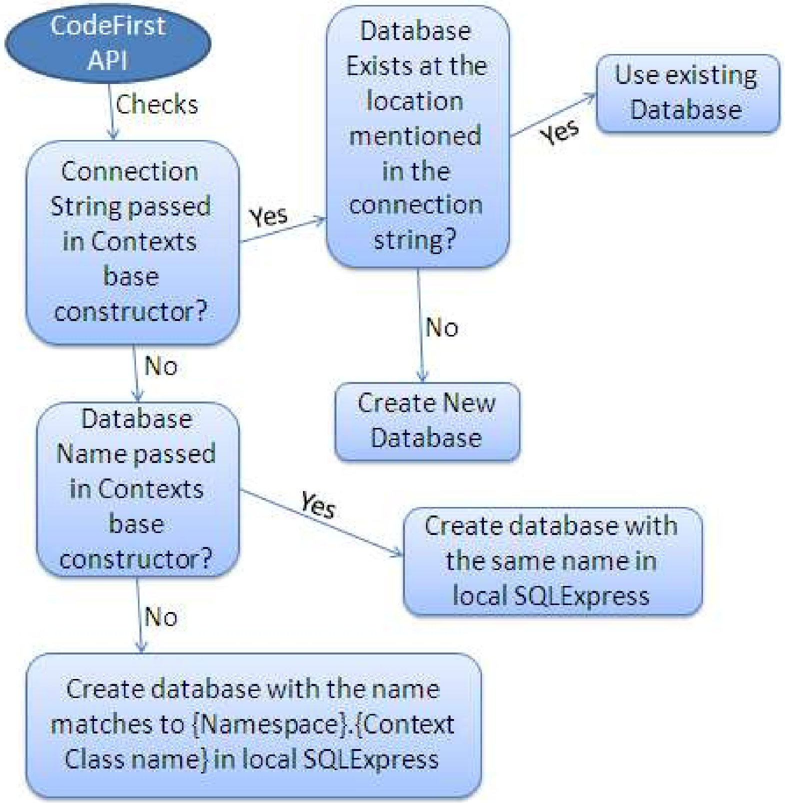
```
public partial class Customers{
    0 references
    public Customers(){
        this.Orders = new HashSet<Orders>();
    }
    0 references
    public string CustomerID { get; set; }
    0 references
    public string CompanyName { get; set; }
    0 references
    public string ContactName { get; set; }
    0 references
    public string ContactTitle { get; set; }
    0 references
    public string Address { get; set; }
    0 references
    public string City { get; set; }
    0 references
    public string Region { get; set; }
    0 references
    public string PostalCode { get; set; }
    0 references
    public string Country { get; set; }
    0 references
    public string Phone { get; set; }
    0 references
    public string Fax { get; set; }
    1 reference
    public virtual ICollection<Orders> Orders { get; set; }
}
```

Navigation Property

Conventions

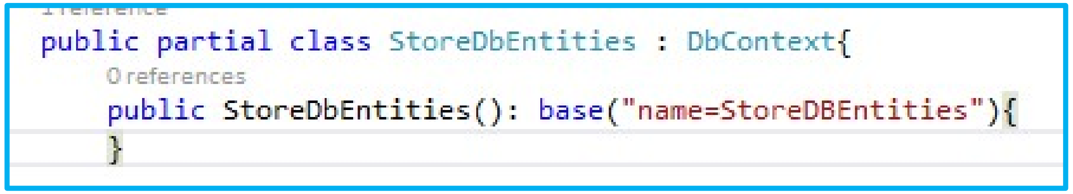
Default Convention For	Description
Table Name	<Entity Class Name> + 's' EF will create DB table with entity class name suffixed by 's'
Primary key Name	1) Id 2) <Entity Class Name> + "Id" (case insensitive) EF will create primary key column for the property named Id or <Entity Class Name> + "Id" (case insensitive)
Foreign key property Name	By default EF will look for foreign key property with the same name as principal entity primary key name. If foreign key property does not exists then EF will create FK column in Db table with <Dependent Navigation Property Name> + "_" + <Principal Entity Primary Key Property Name> e.g. EF will create Orders _OrderId foreign key column into Customer table if Customer entity does not contain foreignkey property for Orders where Orders contains OrderId
Null column	EF creates null column for all reference type properties and nullable primitive properties.
Not Null Column	EF creates NotNull columns for PrimaryKey properties and non-nullable value type properties.
DB Columns order	EF will create DB columns same as order of properties in an entity class. However, primary key columns would be moved first.
Properties mapping to DB	By default all properties will map to database. Use [NotMapped] attribute to exclude property or class from DB mapping.
Cascade delete	Enabled By default for all types of relationships.

Database Initialization


CodeFirst API
Checks
Connection String passed in Contexts base constructor?
Yes
Database Exists at the location mentioned in the connection string?
Yes
Use existing Database
No
Create New Database
No
Database Name passed in Contexts base constructor?
Yes
Create database with the same name in local SQLExpress
No
Create database with the name matches to {Namespace}.{Context Class name} in local SQLExpress

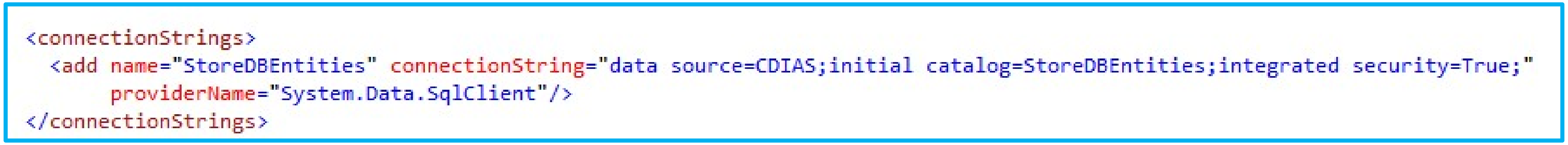
Database Initialization

- DbContext constructor:
  - No Parameter:
    - No parameter in the base constructor of the context class then it creates a database in the server with a name that matches your {Namespace}.{Context class name}.
  - Database Name:



```
public partial class StoreDbEntities : DbContext{
    0 references
    public StoreDbEntities(): base("name=StoreDBEntities"){
    }
```

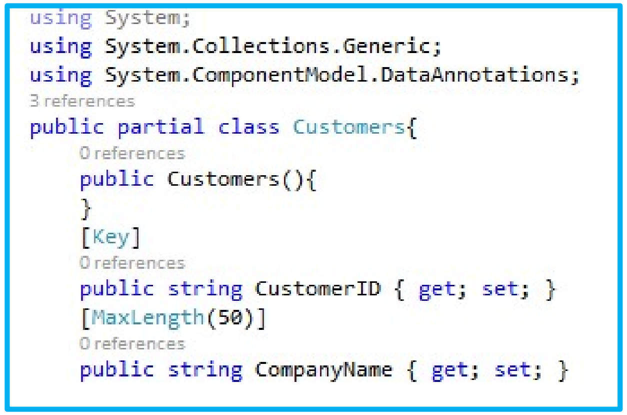
  - Connection String Name



```
<connectionStrings>
  <add name="StoreDBEntities" connectionString="data source=CDIAS;initial catalog=StoreDBEntities;integrated security=True;"
       providerName="System.Data.SqlClient"/>
</connectionStrings>
```

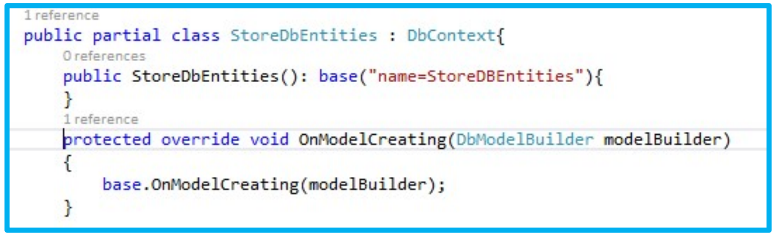
Configure entities/domain classes

- **DataAnnotation:**
  - DataAnnotation is a simple attribute based configuration, which you can apply to your domain classes and its properties.



```
using System;
using System.Collections.Generic;
using System.ComponentModel.DataAnnotations;
3 references
public partial class Customers{
    0 references
    public Customers(){
    }
    [Key]
    0 references
    public string CustomerID { get; set; }
    [MaxLength(50)]
    0 references
    public string CompanyName { get; set; }
```

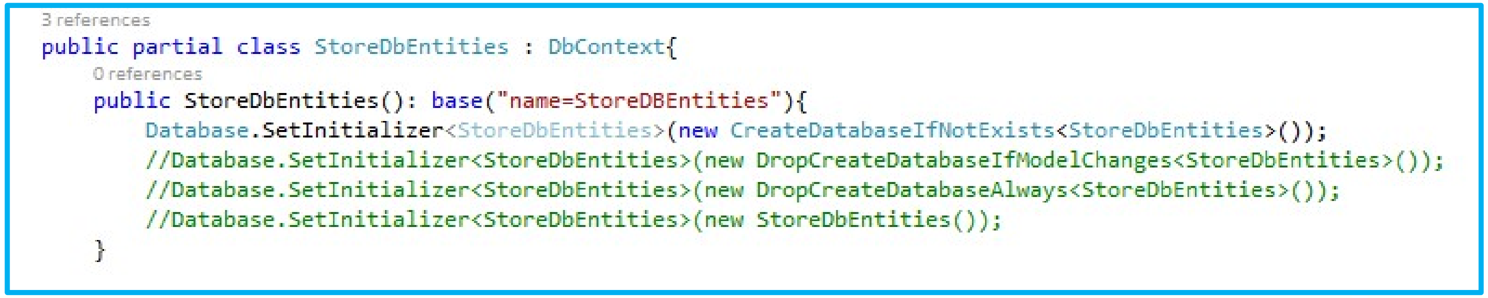
- **Fluent API:**
  - Entity Mappings
  - Property Mapping
  - EntityTypeConfiguration



```
1 reference
public partial class StoreDbEntities : DbContext{
    0 references
    public StoreDbEntities(): base("name=StoreDBEntities"){
    }
    1 reference
    protected override void OnModelCreating(DbModelBuilder modelBuilder)
    {
        base.OnModelCreating(modelBuilder);
    }
```

Database Initialization

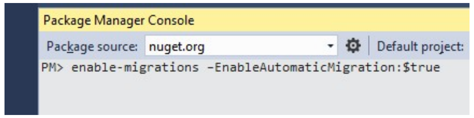
- **Database Initialization Strategies in Code-First:**
  - **CreateDatabaseIfNotExists**:
    - This is default initializer. As the name suggests, it will create the database if none exists as per the configuration. However, if the model class is changed and then run the application with this initializer, then it will throw an exception.
  - **DropCreateDatabaseIfModelChanges**:
    - This initializer drops an existing database and creates a new database, if the model classes (entity classes) have been changed.
  - **DropCreateDatabaseAlways**:
    - This initializer drops an existing database every time the application is executed, irrespective of whether the model classes have changed or not. This will be useful, when a fresh database is needed, every time the application run.
  - **Custom DB Initializer:**
    - It is possible to create a custom initializer, if any of the above does not fulfill the requirements or there is the need of other process that initializes the database using the above initializer.



```
3 references
public partial class StoreDbEntities : DbContext{
    0 references
    public StoreDbEntities(): base("name=StoreDBEntities"){
        Database.SetInitializer<StoreDbEntities>(new CreateDatabaseIfNotExists<StoreDbEntities>());
        //Database.SetInitializer<StoreDbEntities>(new DropCreateDatabaseIfModelChanges<StoreDbEntities>());
        //Database.SetInitializer<StoreDbEntities>(new DropCreateDatabaseAlways<StoreDbEntities>());
        //Database.SetInitializer<StoreDbEntities>(new StoreDbEntities());
    }
```

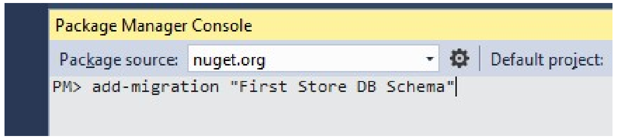
Migration

- There some problems with Database Initialization strategies, for example:
  - If the database already has data (other than seed data) or existing Stored Procedures, triggers etc, these strategies used to drop the entire database and recreate it, so all data and other DB objects will be lost.
  - Entity framework has introduced a migration tool that automatically updates the database schema, when the model changes without losing any existing data or other database objects. It uses a new database initializer called **MigrateDatabaseToLatestVersion**.
- Automatic Migration:

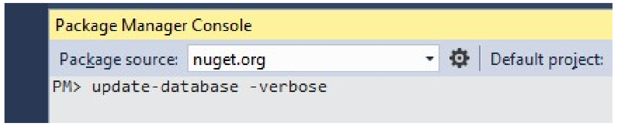


```
PM> enable-migrations -EnableAutomaticMigration:$true
```

- Code based Migration:



```
PM> add-migration "First Store DB Schema"
```

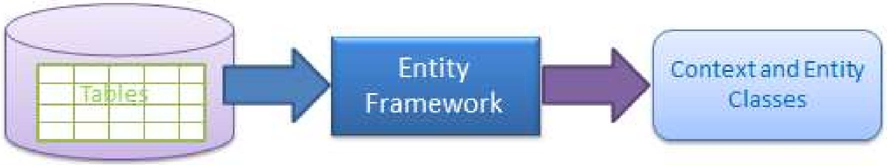


```
PM> update-database -verbose
```

DATABASE FIRST

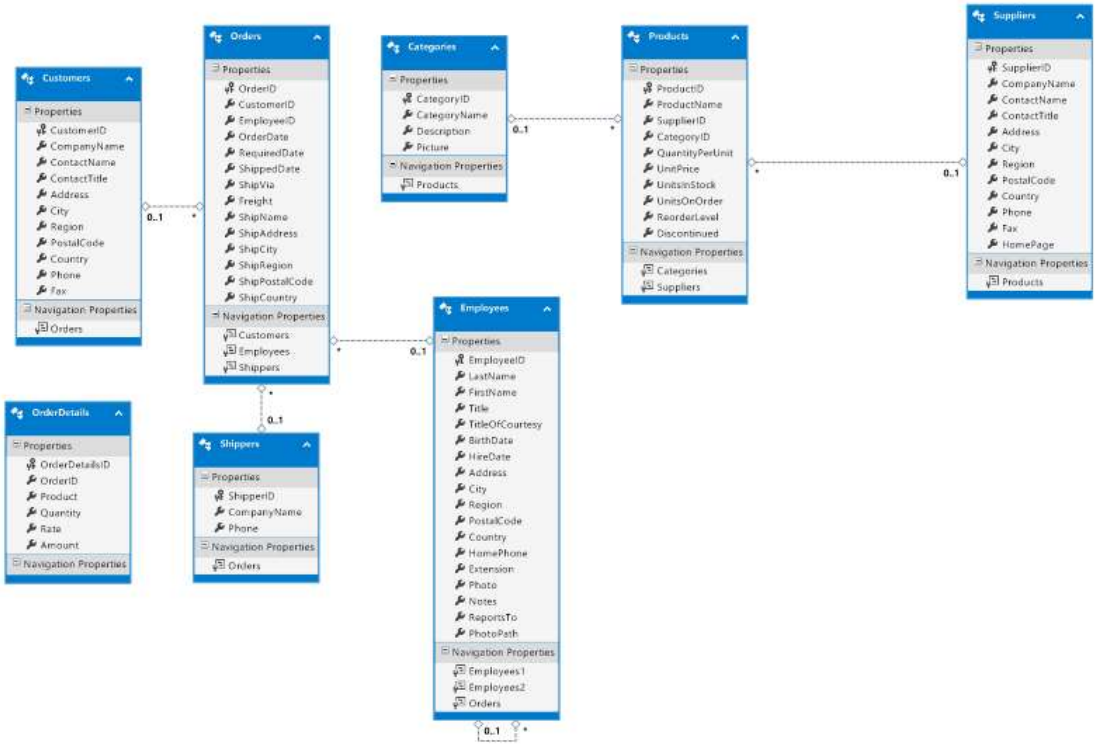
Database First Development

- In this approach, context and entity classes are created from an existing database.
- It will generate EDMX from an existing database



- Entity Data Model can be updated whenever database schema changes. Also, database-first approach supports stored procedure, view, etc.

Entity Data Model (EDM)



Customers
Properties
CustomerID
CompanyName
ContactName
ContactTitle
Address
City
Region
PostalCode
Country
Phone
Fax
Navigation Properties
Orders

Orders
Properties
OrderID
CustomerID
EmployeeID
OrderDate
RequiredDate
ShippedDate
ShipVia
Freight
ShipName
ShipAddress
ShipCity
ShipRegion
ShipPostalCode
ShipCountry
Navigation Properties
Customers
Employees
Shippers

Categories
Properties
CategoryID
CategoryName
Description
Picture
Navigation Properties
Products

Products
Properties
ProductID
ProductName
SupplierID
CategoryID
QuantityPerUnit
UnitPrice
UnitsInStock
UnitsOnOrder
ReorderLevel
Discontinued
Navigation Properties
Categories
Suppliers

Suppliers
Properties
SupplierID
CompanyName
ContactName
ContactTitle
Address
City
Region
PostalCode
Country
Phone
Fax
HomePage
Navigation Properties
Products

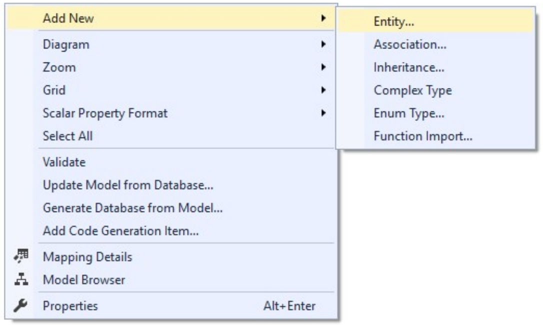
OrderDetails
Properties
OrderDetailsID
OrderID
Product
Quantity
Rate
Amount
Navigation Properties

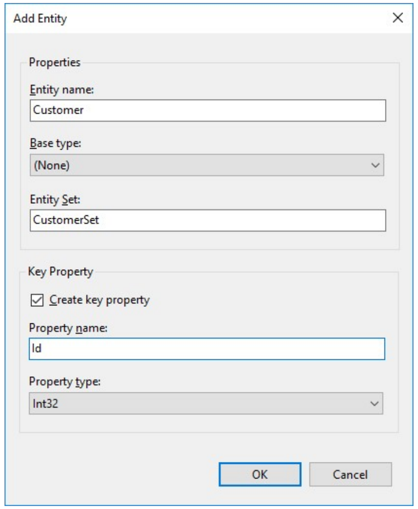
Shippers
Properties
ShipperID
CompanyName
Phone
Navigation Properties
Orders

Employees
Properties
EmployeeID
LastName
FirstName
Title
TitleOfCourtesy
BirthDate
HireDate
Address
City
Region
PostalCode
Country
HomePhone
Extension
Photo
Notes
ReportsTo
PhotoPath
Navigation Properties
Employees1
Employees2
Orders

MODEL FIRST

- The Model First approach, the Entities, relationships, and inheritance hierarchies are created directly on the design surface of EDMX and then generate database from your model.
- In the Model First approach, add new ADO.NET Entity Data Model and select **Empty EF Designer model** in Entity Data Model Wizard.




Add Entity

Properties

Entity name: Customer

Base type: (None)

Entity Set: CustomerSet

Key Property

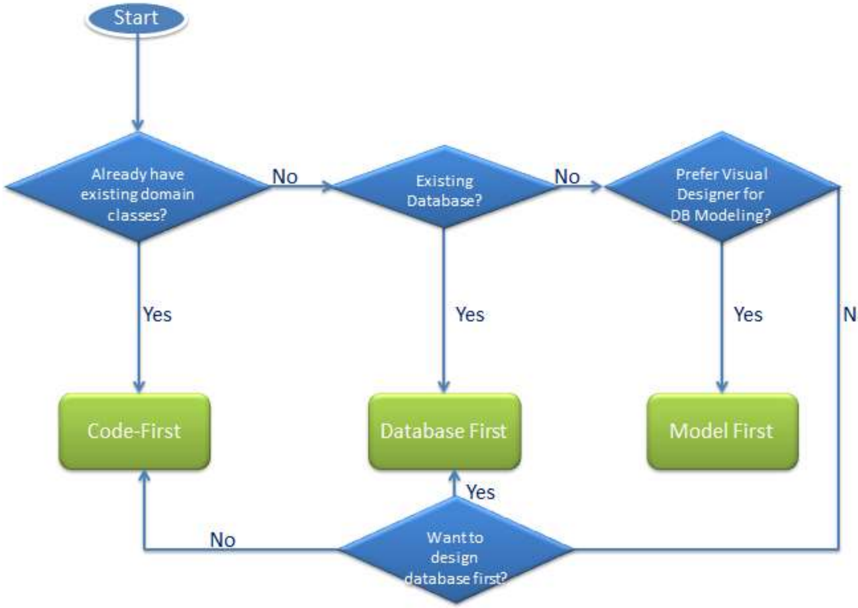
☑ Create key property

Property name: Id

Property type: Int32

Which development approach?

- Already have an existing application with domain classes?
  - Use the code-first approach;
- Have an existing database?
  - Then can create an EDM from an existing database in the database-first approach.
- If there is no database or domain classes, then it is possible to with Model-first approach.

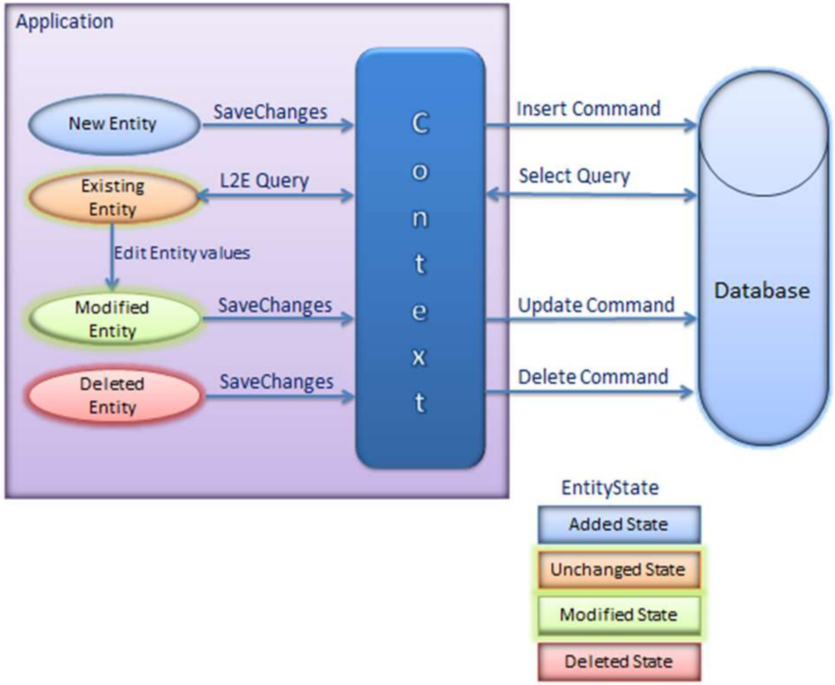

Start
Already have existing domain classes?
No
Existing Database?
No
Prefer Visual Designer for DB Modeling?
Yes
Yes
Yes
No
Code-First
Database First
Model First
Yes
No
Want to design database first?

Relations between Entities

- **One-to-one**
- **One-to-many**
- **Many-to-many**

Entity Life-cycle

- During an entity's lifetime, each entity has an entity state based on the operation performed on it via the context (DbContext).
- The entity state is an enum of type *System.Data.Entity.EntityState* that includes the following values:
- Added
- Deleted
- Modified
- Unchanged
- Detached

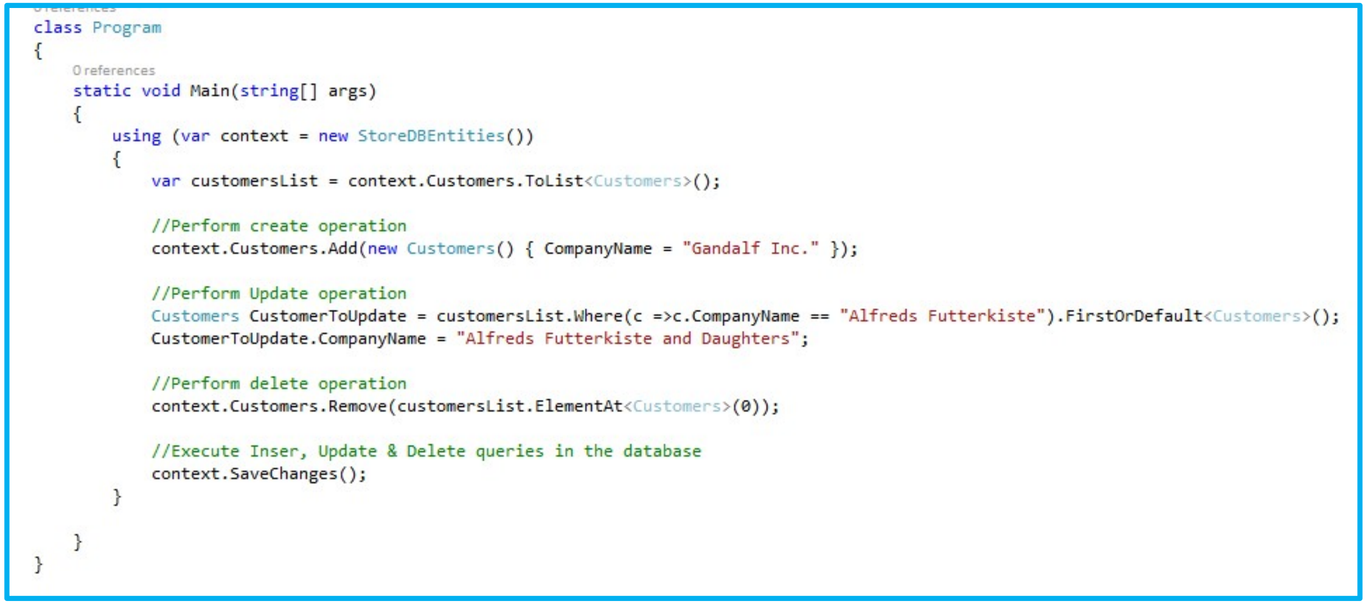


CRUD Operations

- Create
- Read
- Update
- Delete

CRUD Connected

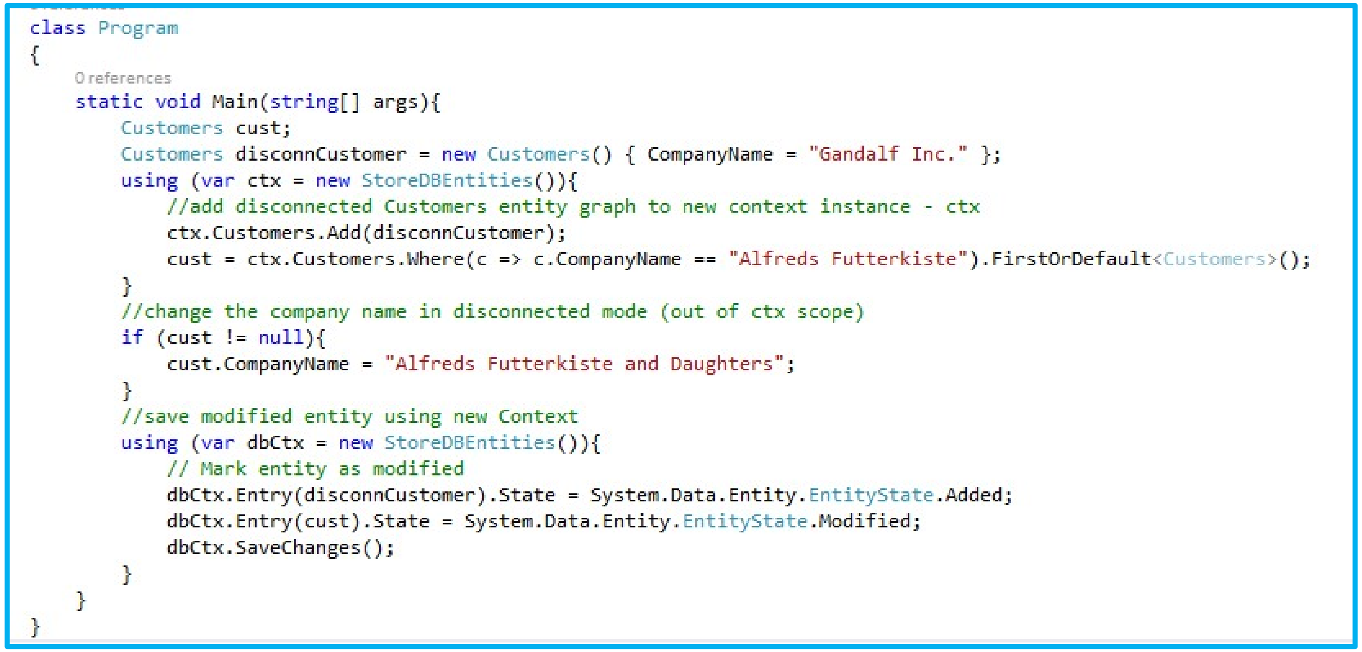
- CRUD operation in connected scenario is a fairly easy task because the context automatically tracks the changes that happened in the entity during its lifetime, provided AutoDetectChangesEnabled is true, by default.



```
class Program
{
    0 references
    static void Main(string[] args)
    {
        using (var context = new StoreDBEntities())
        {
            var customersList = context.Customers.ToList<Customers>();

            //Perform create operation
            context.Customers.Add(new Customers() { CompanyName = "Gandalf Inc." });

            //Perform Update operation
            Customers CustomerToUpdate = customersList.Where(c =>c.CompanyName == "Alfreds Futterkiste").FirstOrDefault<Customers>();
            CustomerToUpdate.CompanyName = "Alfreds Futterkiste and Daughters";

            //Perform delete operation
            context.Customers.Remove(customersList.ElementAt<Customers>(0));

            //Execute Inser, Update & Delete queries in the database
            context.SaveChanges();
        }

    }
}
```

CRUD Disconnected

- Entity Framework API provides some important methods that attaches disconnected entities to the new context and also set EntityStates to all the entities of an entity graph.



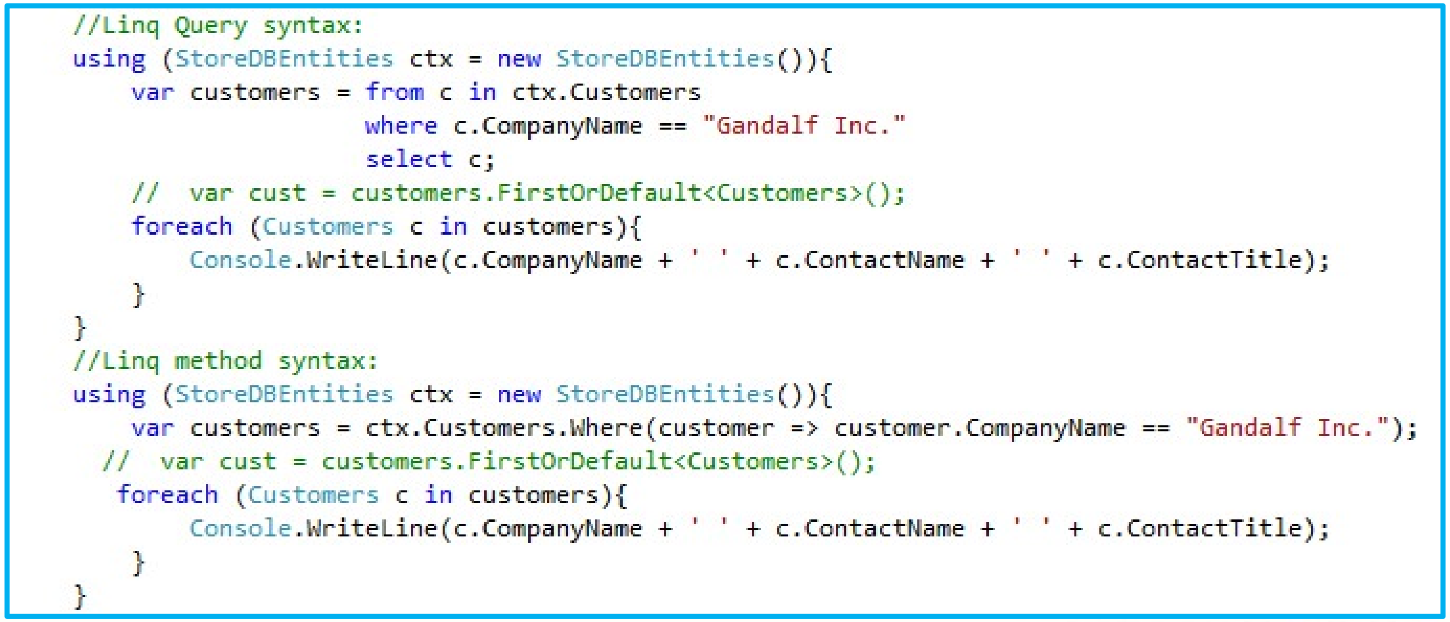
```
class Program
{
    0 references
    static void Main(string[] args){
        Customers cust;
        Customers disconnCustomer = new Customers() { CompanyName = "Gandalf Inc." };
        using (var ctx = new StoreDBEntities()){
            //add disconnected Customers entity graph to new context instance - ctx
            ctx.Customers.Add(disconnCustomer);
            cust = ctx.Customers.Where(c => c.CompanyName == "Alfreds Futterkiste").FirstOrDefault<Customers>();
        }
        //change the company name in disconnected mode (out of ctx scope)
        if (cust != null){
            cust.CompanyName = "Alfreds Futterkiste and Daughters";
        }
        //save modified entity using new Context
        using (var dbCtx = new StoreDBEntities()){
            // Mark entity as modified
            dbCtx.Entry(disconnCustomer).State = System.Data.Entity.EntityState.Added;
            dbCtx.Entry(cust).State = System.Data.Entity.EntityState.Modified;
            dbCtx.SaveChanges();
        }
    }
}
```

Linq to Entities

- **LINQ** stands for **L**anguage-**IN**tegrated **Q**uery. It was introduced along with Visual Studio 2008.
- It can be used and extended to support any Kind of Data Stores. LINQ to Entities is a subset of LINQ which allows us to write queries against the Entity Framework conceptual models.
- The query syntax
- The method syntax

Query vs method Syntax

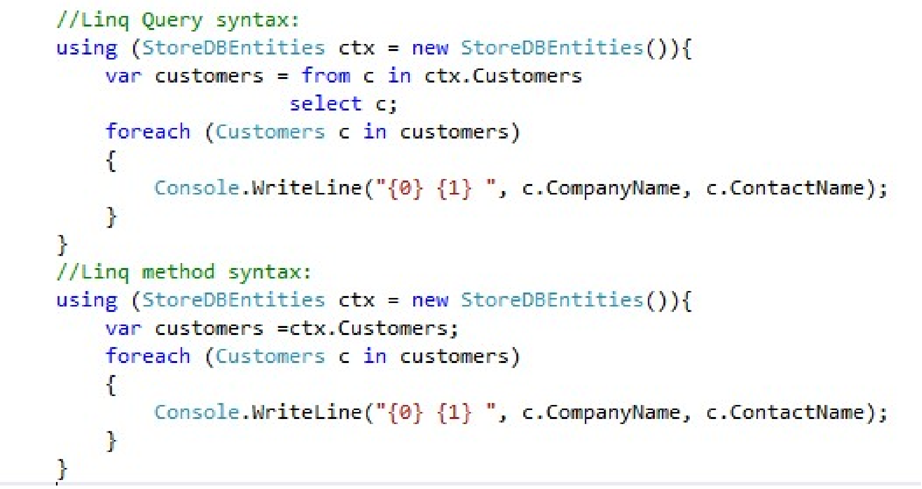
- Query expression syntax consists of a set of clauses written in a declarative syntax, is very similar to Transact-SQL.
  - The .NET Framework CLR does not understand these queries. Hence, they are translated to method syntax at the compile time.



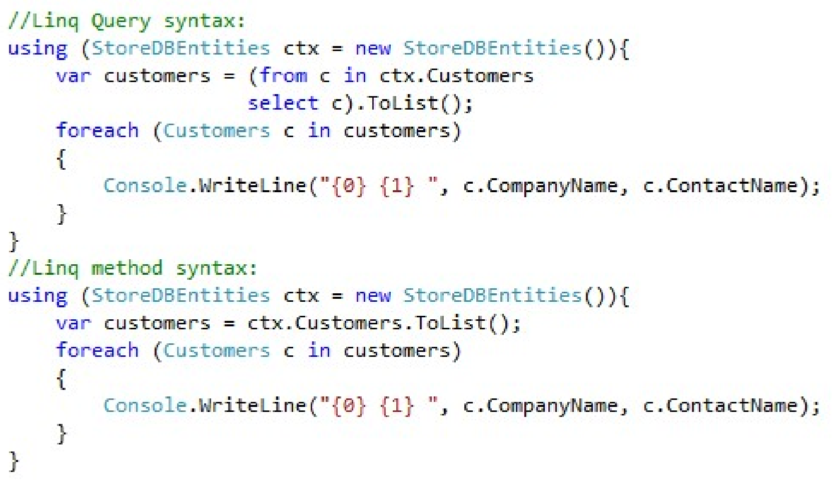
```
//Linq Query syntax:
using (StoreDBEntities ctx = new StoreDBEntities()){
    var customers = from c in ctx.Customers
                    where c.CompanyName == "Gandalf Inc."
                    select c;
    //  var cust = customers.FirstOrDefault<Customers>();
    foreach (Customers c in customers){
        Console.WriteLine(c.CompanyName + ' ' + c.ContactName + ' ' + c.ContactTitle);
    }
}
//Linq method syntax:
using (StoreDBEntities ctx = new StoreDBEntities()){
    var customers = ctx.Customers.Where(customer => customer.CompanyName == "Gandalf Inc.");
  //  var cust = customers.FirstOrDefault<Customers>();
   foreach (Customers c in customers){
        Console.WriteLine(c.CompanyName + ' ' + c.ContactName + ' ' + c.ContactTitle);
    }
}
```

Query all entities

- **Query all entities**
- **Query all entities ToList**



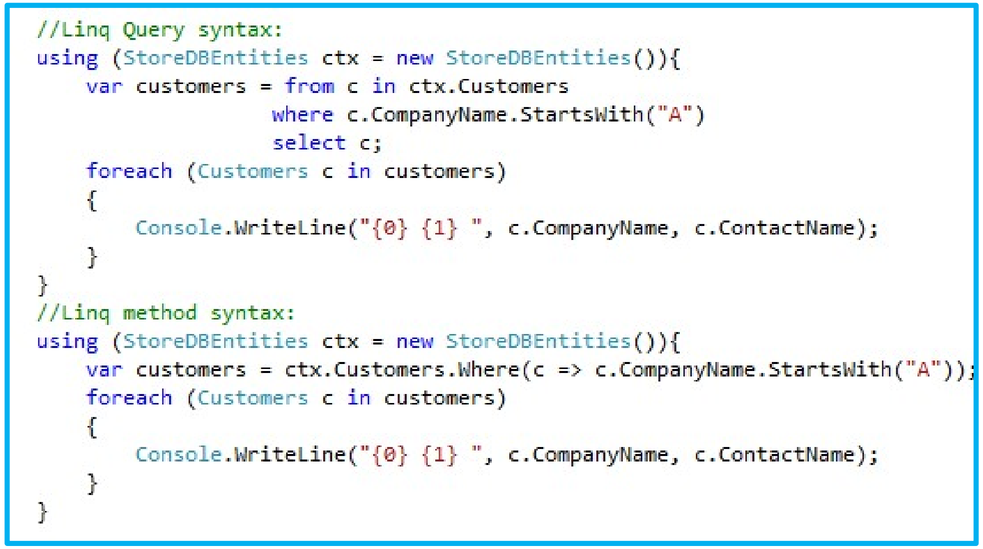
```
//Linq Query syntax:
using (StoreDBEntities ctx = new StoreDBEntities()){
    var customers = from c in ctx.Customers
                   select c;
    foreach (Customers c in customers)
    {
        Console.WriteLine("{0} {1} ", c.CompanyName, c.ContactName);
    }
}
//Linq method syntax:
using (StoreDBEntities ctx = new StoreDBEntities()){
    var customers =ctx.Customers;
    foreach (Customers c in customers)
    {
        Console.WriteLine("{0} {1} ", c.CompanyName, c.ContactName);
    }
}
```



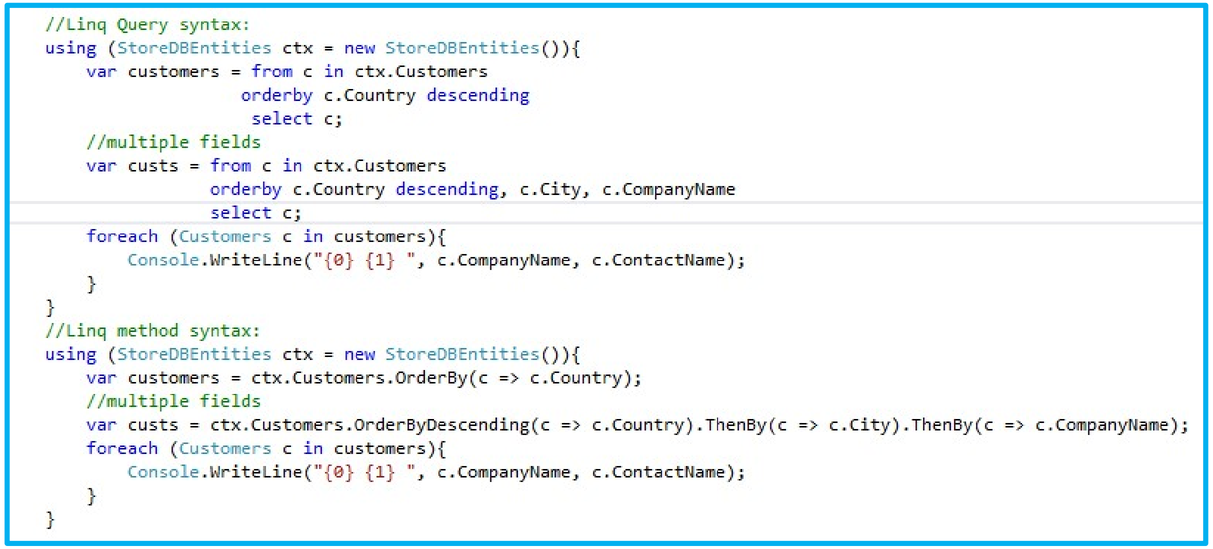
```
//Linq Query syntax:
using (StoreDBEntities ctx = new StoreDBEntities()){
    var customers = (from c in ctx.Customers
                    select c).ToList();
    foreach (Customers c in customers)
    {
        Console.WriteLine("{0} {1} ", c.CompanyName, c.ContactName);
    }
}
//Linq method syntax:
using (StoreDBEntities ctx = new StoreDBEntities()){
    var customers = ctx.Customers.ToList();
    foreach (Customers c in customers)
    {
        Console.WriteLine("{0} {1} ", c.CompanyName, c.ContactName);
    }
}
```

Filtering and Ordering the Query results

- **Filter results**
- **Order results**



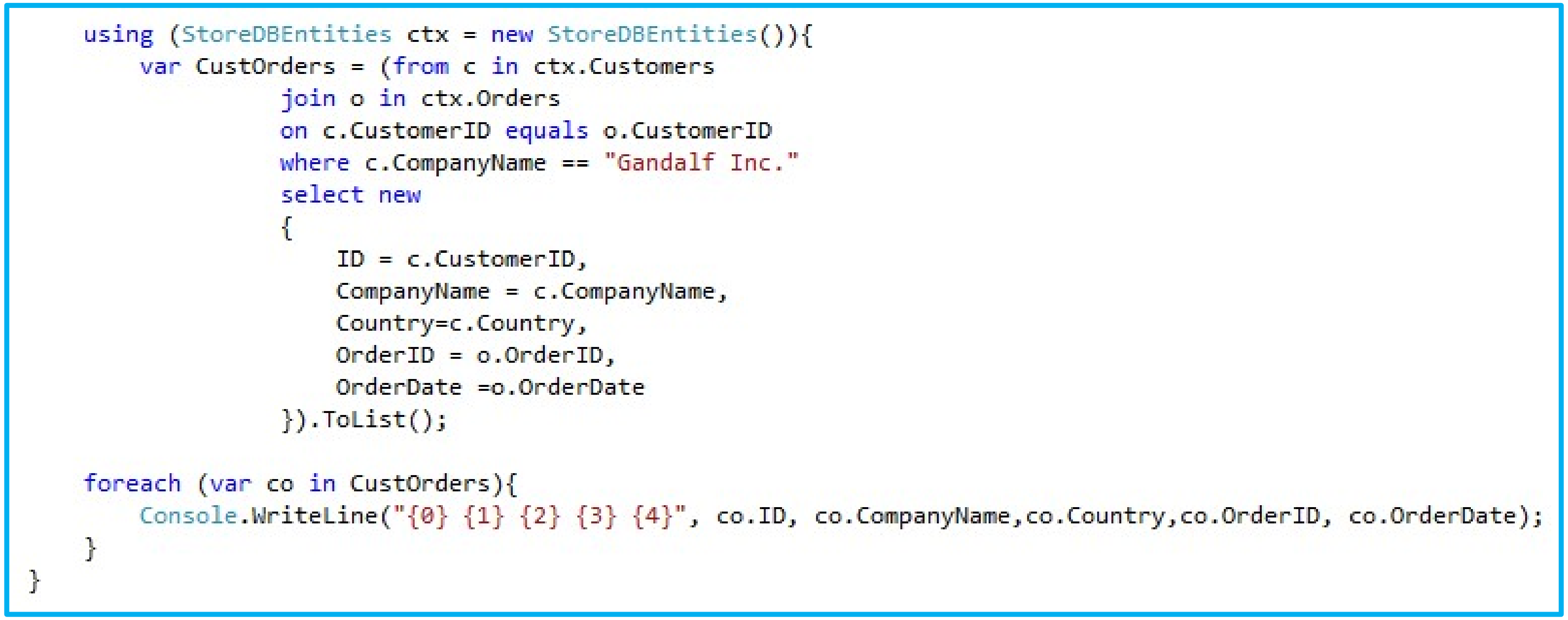
```
//Linq Query syntax:
using (StoreDBEntities ctx = new StoreDBEntities()){
    var customers = from c in ctx.Customers
                    where c.CompanyName.StartsWith("A")
                    select c;
    foreach (Customers c in customers)
    {
        Console.WriteLine("{0} {1} ", c.CompanyName, c.ContactName);
    }
}
//Linq method syntax:
using (StoreDBEntities ctx = new StoreDBEntities()){
    var customers = ctx.Customers.Where(c => c.CompanyName.StartsWith("A"));
    foreach (Customers c in customers)
    {
        Console.WriteLine("{0} {1} ", c.CompanyName, c.ContactName);
    }
}
```



```
//Linq Query syntax:
using (StoreDBEntities ctx = new StoreDBEntities()){
    var customers = from c in ctx.Customers
                    orderby c.Country descending
                     select c;
    //multiple fields
    var custs = from c in ctx.Customers
                orderby c.Country descending, c.City, c.CompanyName
                select c;
    foreach (Customers c in customers){
        Console.WriteLine("{0} {1} ", c.CompanyName, c.ContactName);
    }
}
//Linq method syntax:
using (StoreDBEntities ctx = new StoreDBEntities()){
    var customers = ctx.Customers.OrderBy(c => c.Country);
    //multiple fields
    var custs = ctx.Customers.OrderByDescending(c => c.Country).ThenBy(c => c.City).ThenBy(c => c.CompanyName);
    foreach (Customers c in customers){
        Console.WriteLine("{0} {1} ", c.CompanyName, c.ContactName);
    }
}
```

Join Queries

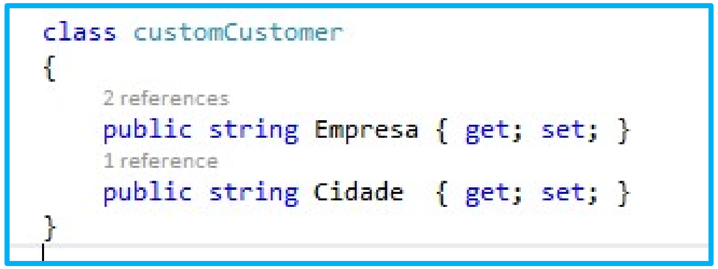
- The Join operators should only be used only if the tables do not have any navigational properties defined on them or want to fine tune the generated queries for performance benefits.



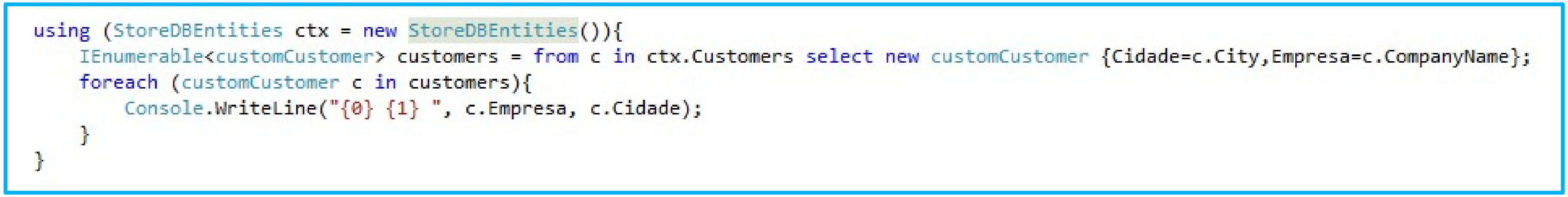
```
    using (StoreDBEntities ctx = new StoreDBEntities()){
        var CustOrders = (from c in ctx.Customers
                  join o in ctx.Orders
                  on c.CustomerID equals o.CustomerID
                  where c.CompanyName == "Gandalf Inc."
                  select new
                  {
                      ID = c.CustomerID,
                      CompanyName = c.CompanyName,
                      Country=c.Country,
                      OrderID = o.OrderID,
                      OrderDate =o.OrderDate
                  }).ToList();

    foreach (var co in CustOrders){
        Console.WriteLine("{0} {1} {2} {3} {4}", co.ID, co.CompanyName,co.Country,co.OrderID, co.OrderDate);
    }
}
```

Projection Query

- To return only some of the fields in the form of a concrete type:

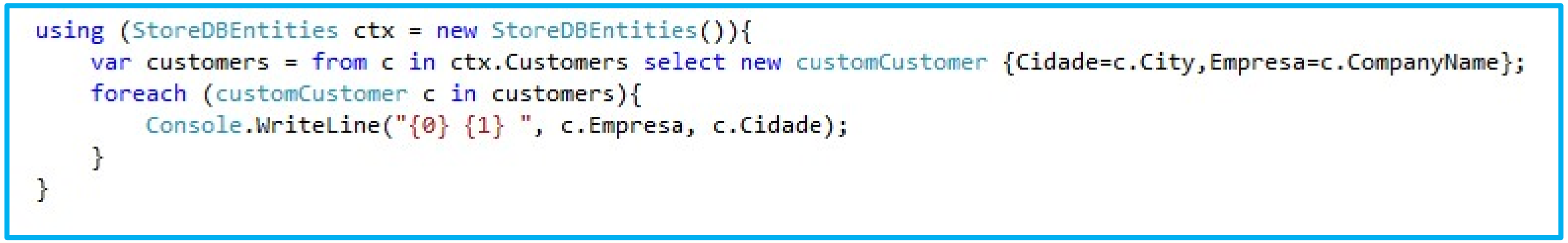


```
class customCustomer
{
    2 references
    public string Empresa { get; set; }
    1 reference
    public string Cidade  { get; set; }
}
```



```
using (StoreDBEntities ctx = new StoreDBEntities()){
    IEnumerable<customCustomer> customers = from c in ctx.Customers select new customCustomer {Cidade=c.City,Empresa=c.CompanyName};
    foreach (customCustomer c in customers){
        Console.WriteLine("{0} {1} ", c.Empresa, c.Cidade);
    }
}
```

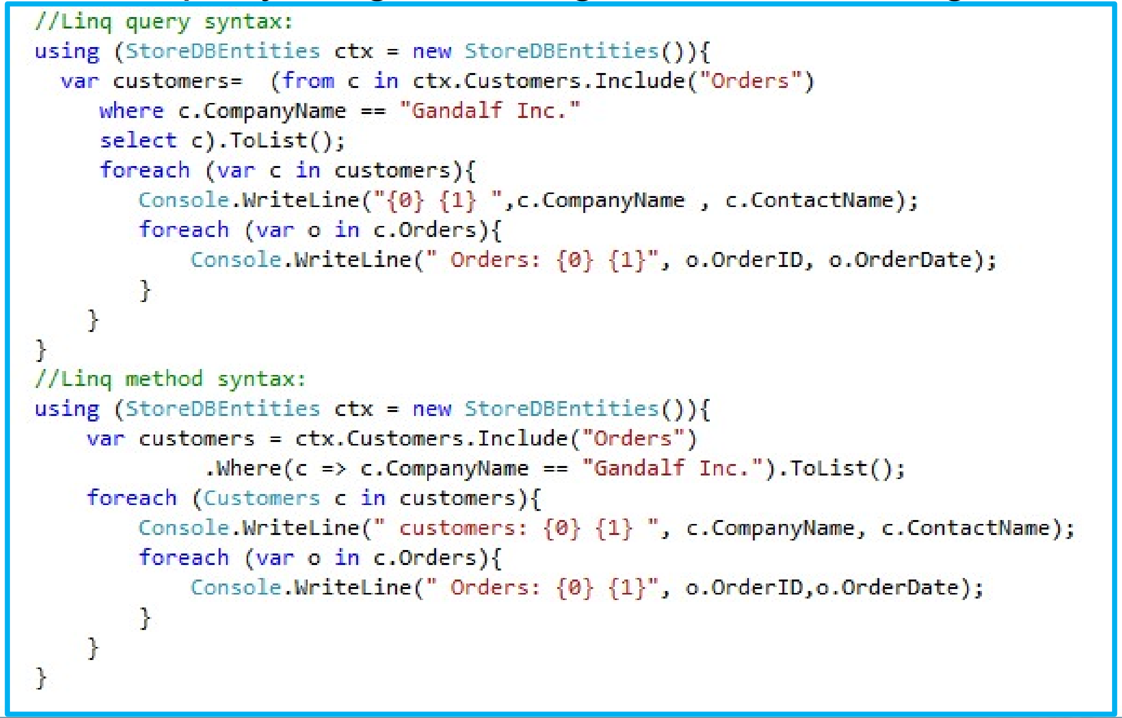
- To retrieve only some of the fields into an anonymous type:



```
using (StoreDBEntities ctx = new StoreDBEntities()){
    var customers = from c in ctx.Customers select new customCustomer {Cidade=c.City,Empresa=c.CompanyName};
    foreach (customCustomer c in customers){
        Console.WriteLine("{0} {1} ", c.Empresa, c.Cidade);
    }
}
```

Loading Related Data

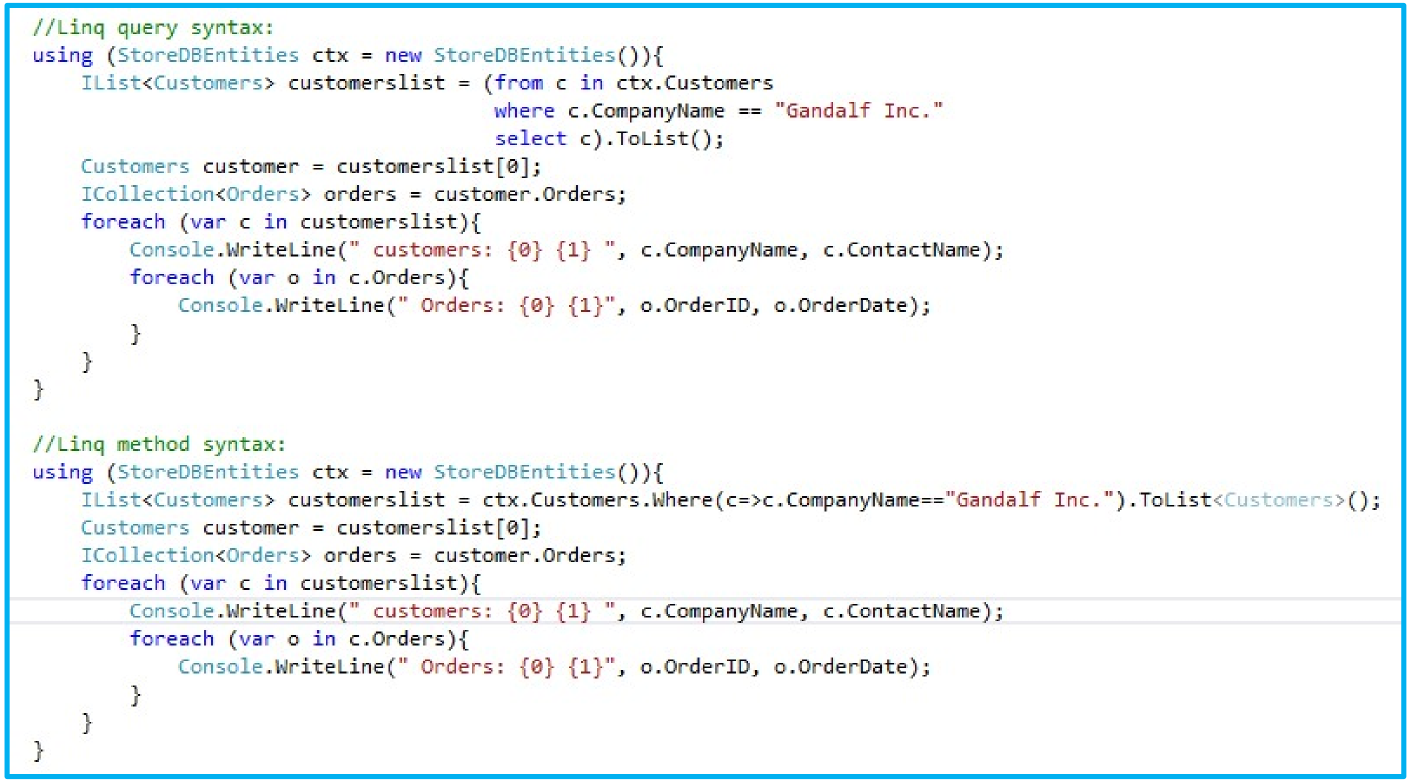
- **Entity Framework allows to retrieve Related data from multiple tables using navigational properties.**
- Eager loading is the process whereby a query for one type of entity also loads related entities as part of the query. Eager loading is achieved using the **Include()** method:



```
//Linq query syntax:
using (StoreDBEntities ctx = new StoreDBEntities()){
  var customers=  (from c in ctx.Customers.Include("Orders")
     where c.CompanyName == "Gandalf Inc."
     select c).ToList();
     foreach (var c in customers){
        Console.WriteLine("{0} {1} ",c.CompanyName , c.ContactName);
        foreach (var o in c.Orders){
            Console.WriteLine(" Orders: {0} {1}", o.OrderID, o.OrderDate);
        }
    }
}
//Linq method syntax:
using (StoreDBEntities ctx = new StoreDBEntities()){
    var customers = ctx.Customers.Include("Orders")
             .Where(c => c.CompanyName == "Gandalf Inc.").ToList();
    foreach (Customers c in customers){
        Console.WriteLine(" customers: {0} {1} ", c.CompanyName, c.ContactName);
        foreach (var o in c.Orders){
            Console.WriteLine(" Orders: {0} {1}", o.OrderID,o.OrderDate);
        }
    }
}
```

Loading Related Data

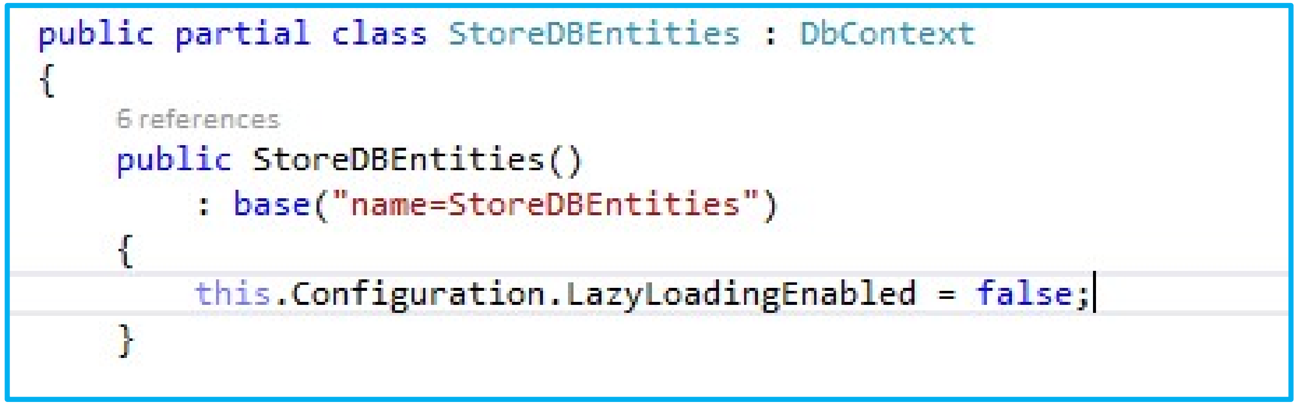
- Lazy loading means delaying the loading of related data, until you specifically request for it.



```
//Linq query syntax:
using (StoreDBEntities ctx = new StoreDBEntities()){
    IList<Customers> customerslist = (from c in ctx.Customers
                                      where c.CompanyName == "Gandalf Inc."
                                      select c).ToList();
    Customers customer = customerslist[0];
    ICollection<Orders> orders = customer.Orders;
    foreach (var c in customerslist){
        Console.WriteLine(" customers: {0} {1} ", c.CompanyName, c.ContactName);
        foreach (var o in c.Orders){
            Console.WriteLine(" Orders: {0} {1}", o.OrderID, o.OrderDate);
        }
    }
}

//Linq method syntax:
using (StoreDBEntities ctx = new StoreDBEntities()){
    IList<Customers> customerslist = ctx.Customers.Where(c=>c.CompanyName=="Gandalf Inc.").ToList<Customers>();
    Customers customer = customerslist[0];
    ICollection<Orders> orders = customer.Orders;
    foreach (var c in customerslist){
        Console.WriteLine(" customers: {0} {1} ", c.CompanyName, c.ContactName);
        foreach (var o in c.Orders){
            Console.WriteLine(" Orders: {0} {1}", o.OrderID, o.OrderDate);
        }
    }
}
```

Deactivate Lazy Loading

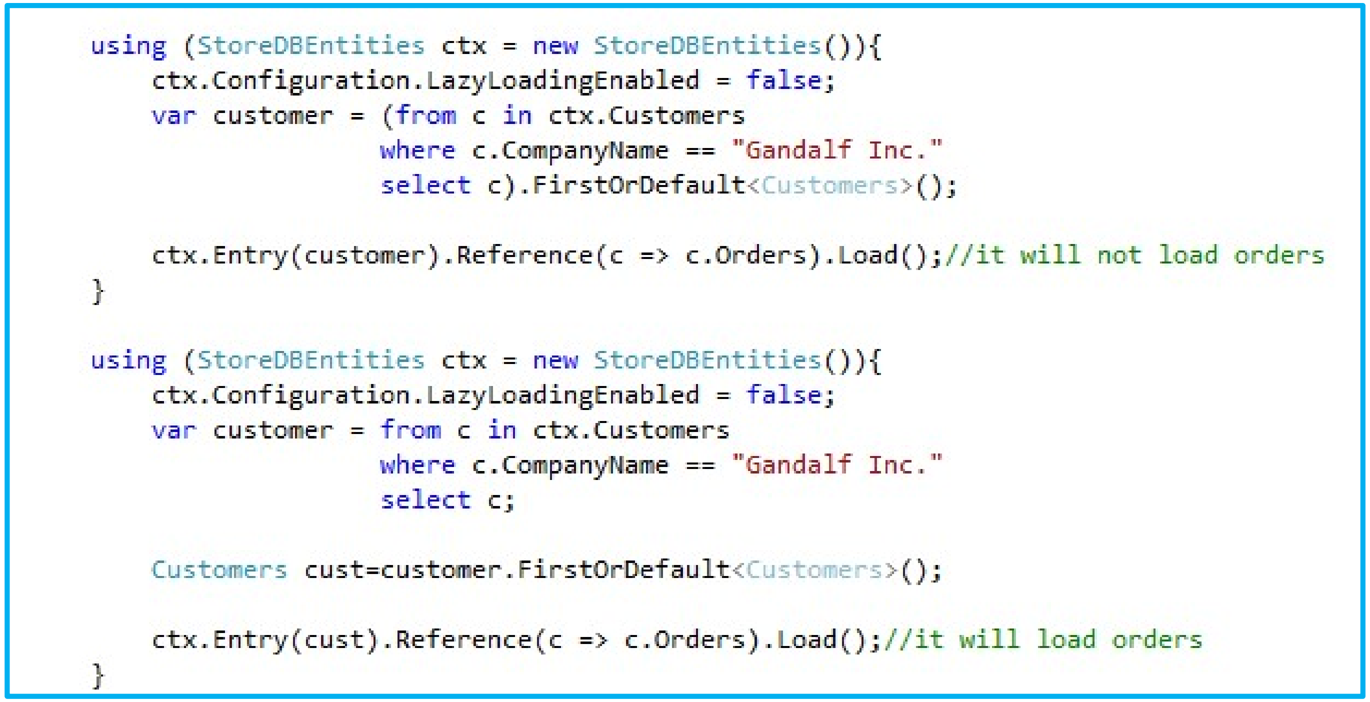
- To turn off lazy loading for a particular property, do not make it virtual.
- To turn off lazy loading for all entities in the context, set its configuration property to false:



```
public partial class StoreDBEntities : DbContext
{
    6 references
    public StoreDBEntities()
        : base("name=StoreDBEntities")
    {
        this.Configuration.LazyLoadingEnabled = false;
    }
```

Explicit Loading

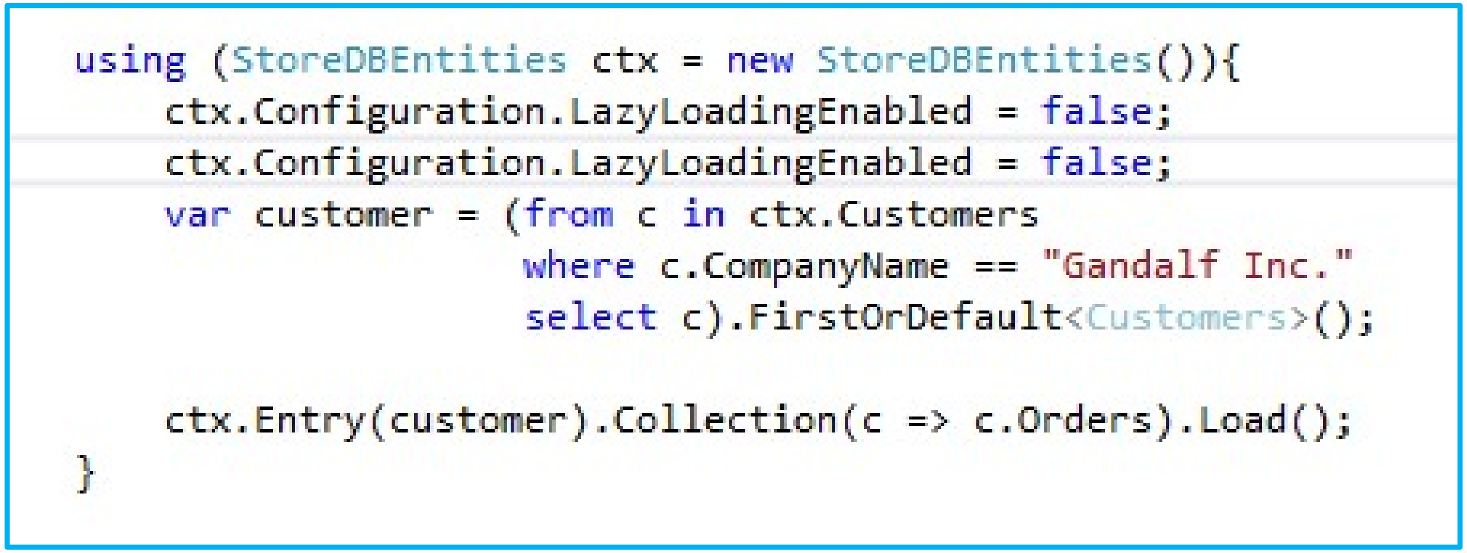
- With lazy loading disabled, it is still possible to lazily load related entities, but it must be done with an explicit call. Use the Load method to accomplish this:
  - Explicitly loads Orders of particular Customer using the Reference() method
  - It will execute two queries:



```
using (StoreDBEntities ctx = new StoreDBEntities()){
    ctx.Configuration.LazyLoadingEnabled = false;
    var customer = (from c in ctx.Customers
                   where c.CompanyName == "Gandalf Inc."
                   select c).FirstOrDefault<Customers>();

    ctx.Entry(customer).Reference(c => c.Orders).Load();//it will not load orders
}

using (StoreDBEntities ctx = new StoreDBEntities()){
    ctx.Configuration.LazyLoadingEnabled = false;
    var customer = from c in ctx.Customers
                   where c.CompanyName == "Gandalf Inc."
                   select c;

    Customers cust=customer.FirstOrDefault<Customers>();

    ctx.Entry(cust).Reference(c => c.Orders).Load();//it will load orders
}
```

Explicit Loading

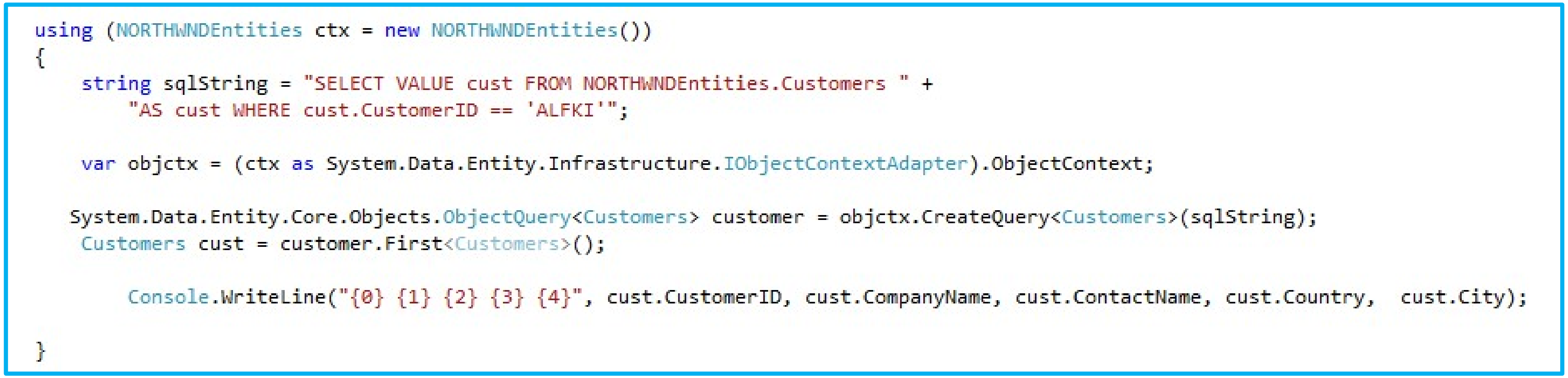
- With lazy loading disabled, it is still possible to lazily load related entities, but it must be done with an explicit call. Use the Load method to accomplish this:
  - Use the Collection() method instead of Reference() method to load collection navigation property:
  - It will execute one query:



```
using (StoreDBEntities ctx = new StoreDBEntities()){
    ctx.Configuration.LazyLoadingEnabled = false;
    ctx.Configuration.LazyLoadingEnabled = false;
    var customer = (from c in ctx.Customers
                    where c.CompanyName == "Gandalf Inc."
                    select c).FirstOrDefault<Customers>();

    ctx.Entry(customer).Collection(c => c.Orders).Load();
}
```

Linq to SQL

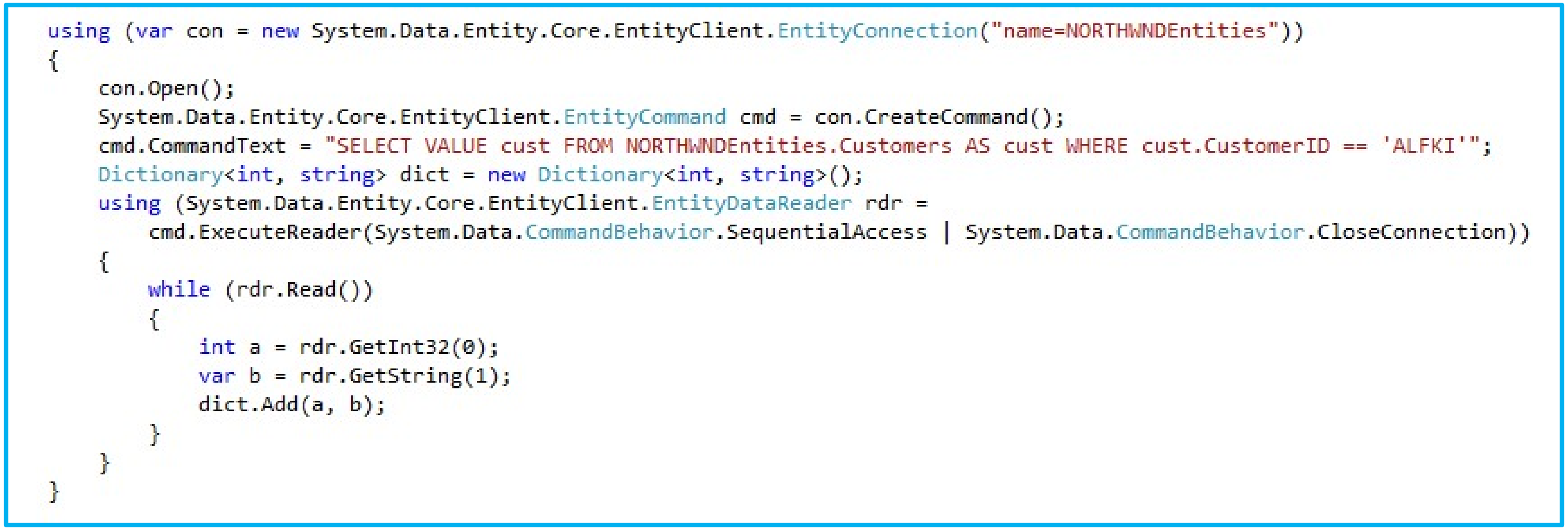
- Another way to create a query is by using Entity SQL.
- It is processed by the Entity Framework’s Object Services directly. It returns ObjectQuery instead of IQueryable.
- Use ObjectContext to create a query using Entity SQL.



```
using (NORTHWNDEntities ctx = new NORTHWNDEntities())
{
    string sqlString = "SELECT VALUE cust FROM NORTHWNDEntities.Customers " +
        "AS cust WHERE cust.CustomerID == 'ALFKI'";

    var objctx = (ctx as System.Data.Entity.Infrastructure.IObjectContextAdapter).ObjectContext;

   System.Data.Entity.Core.Objects.ObjectQuery<Customers> customer = objctx.CreateQuery<Customers>(sqlString);
    Customers cust = customer.First<Customers>();

        Console.WriteLine("{0} {1} {2} {3} {4}", cust.CustomerID, cust.CompanyName, cust.ContactName, cust.Country,  cust.City);

}
```

Linq to SQL

- Use EntityConnection and EntityCommand to execute Entity SQL
- EntityDataReader doesn't return ObjectQuery. Instead, it returns the data in rows & columns.



```
using (var con = new System.Data.Entity.Core.EntityClient.EntityConnection("name=NORTHWNDEntities"))
{
    con.Open();
    System.Data.Entity.Core.EntityClient.EntityCommand cmd = con.CreateCommand();
    cmd.CommandText = "SELECT VALUE cust FROM NORTHWNDEntities.Customers AS cust WHERE cust.CustomerID == 'ALFKI'";
    Dictionary<int, string> dict = new Dictionary<int, string>();
    using (System.Data.Entity.Core.EntityClient.EntityDataReader rdr =
        cmd.ExecuteReader(System.Data.CommandBehavior.SequentialAccess | System.Data.CommandBehavior.CloseConnection))
    {
        while (rdr.Read())
        {
            int a = rdr.GetInt32(0);
            var b = rdr.GetString(1);
            dict.Add(a, b);
        }
    }
}
```

Exercise

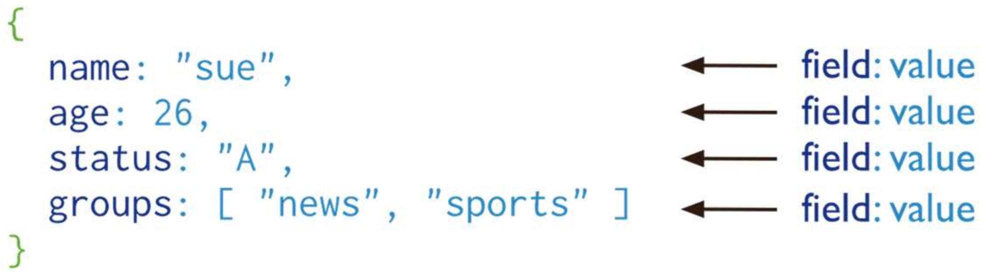
- **Exercise 1 – EntityFramwork Core**

NOSQL - MongoDB

- MongoDB is a document database designed for ease of development and scaling.
- Document Database
  - A record in MongoDB is a document, which is a data structure composed of field and value pairs. MongoDB documents are similar to JSON objects. The values of fields may include other documents, arrays, and arrays of documents.

NOSQL - MongoDB

- Documents (i.e. objects) correspond to native data types in many programming languages.
- Embedded documents and arrays reduce need for expensive joins.
- Dynamic schema supports fluent polymorphism.

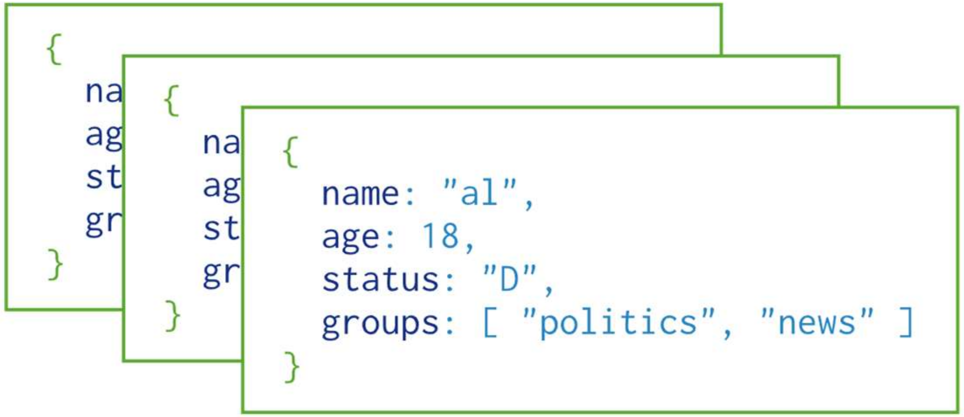


```
{
   name: "sue",
   age: 26,
   status: "A",
   groups: [ "news", "sports" ]
}
```

NOSQL - MongoDB

- Databases and Collections

- MongoDB provides the db.createCollection() method to explicitly create a collection with various options, such as setting the maximum size or the documentation validation rules. If you are not specifying these options, you do not need to explicitly create the collection since MongoDB creates new collections when you first store data for the collections.



```
{
   name: "al",
   age: 18,
   status: "D",
   groups: [ "politics", "news" ]
}
```

Collection

NOSQL - MongoDB

Installation

- NuGet is the simplest way to get the driver. Use MongoDB.Driver for all new projects.

Connect to MongoDB Atlas

To connect to a MongoDB Atlas cluster, use the Atlas connection string for your cluster:

```
using MongoDB.Bson;
using MongoDB.Driver;
// ...
var client = new MongoClient(
   "mongodb+srv://<username>:<password>@<cluster-address>/test?w=majority"
);
var database = client.GetDatabase("test");
```

MongoDB

- Exercicio 2