Module 7: Accessing a Database

Contents:

Module overview

Lesson 1: Creating and Using Entity Data Models

Lesson 2: Querying Data by Using LINQ

Lab: Retrieving and Modifying Grade Data

Module review and takeaways

Module overview

Many applications require access to data that is stored in a database. Microsoft® Visual Studio® 2017 and the Microsoft .NET Framework provide tools and functionality that you can use to easily access, query, and update data.

In this module, you will learn how to create and use entity data models (EDMs) and how to query many types of data by using Language-Integrated Query (LINQ).

Objectives

After completing this module, you will be able to:

- Create, use, and customize an EDM.
- Query data by using LINQ.

Lesson 1: Creating and Using Entity Data Models

Data access applications have traditionally been tedious to develop. They often contain queries that are written as text strings that cannot be type-checked or syntax-checked at compile time, and results are returned as untyped data records. The ADO.NET Entity Framework solves these problems and simplifies the process of developing data access applications by using EDMs.

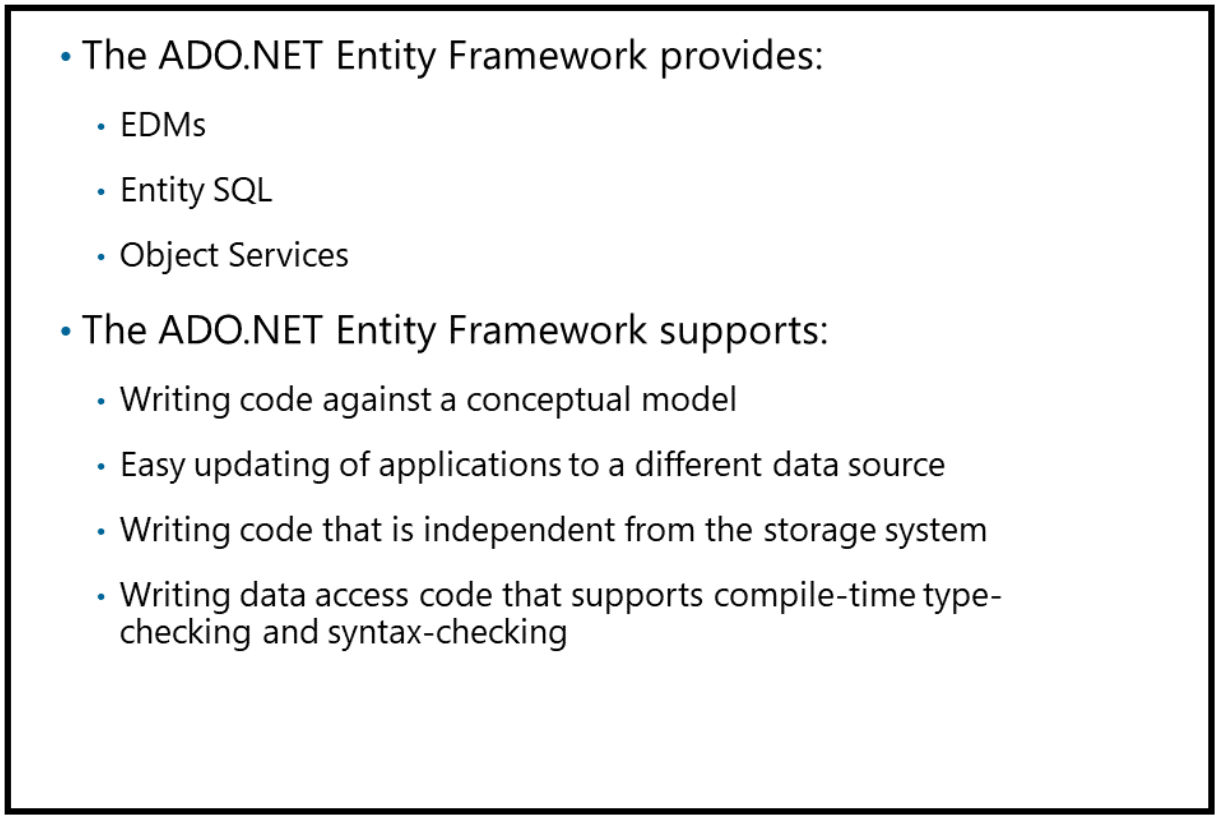
In this lesson, you will learn how to use the ADO.NET Entity Data Tools to create EDMs, how to customize the classes that the tools generate, and how to access the entities in the generated model.

Lesson objectives

After completing this lesson, you will be able to:

- Describe the ADO.NET Entity Framework.
- Use the ADO.NET Entity Data Model Tools.
- Customize generated classes.
- Read and modify data by using the Entity Framework.

Introduction to the ADO.NET Entity Framework



Historically when you write code to access data that is stored in a database, you have to understand the structure of the data in the database and how it all interrelates. Often it is stored in a normalized fashion, where tables do not logically map to the real-life objects that they represent. The ADO.NET Entity Framework enables you to develop applications that target a conceptual model instead of the normalized database structure in the storage layer.

The ADO.NET Entity Framework provides the following:

- *EDMs*. These are models that you can use to map database tables and queries to .NET Framework objects.
- *Entity Structured Query Language (SQL)*. This is a storage independent query language that enables you to query and manipulate EDM constructs.
- *Object Services*. These are services that enable you to work with the Common Language Runtime (CLR) objects in a conceptual model.

  These components enable you to:

- Write code against a conceptual model that includes types that support inheritance and relationships.

- Update applications to target a different storage model without rewriting or redistributing all of your data access code.
- Write standard code that is not dependent on the data storage system.
- Write data access code that supports compile-time type-checking and syntax-checking.

The conceptual model that you work with in the Entity Framework describes the semantics of the business view of the data. It defines entities and relationships in a business sense and is mapped to the logical model of the underlying data in the data source. For example, in a human resources application, entities may include employees, jobs, and branch locations. An entity is a description of the items and their properties, and they are linked by relationships, such as an employee being related to a particular branch location.

> **Additional Reading:** For more information about the ADO.NET Entity Framework, refer to the ADO.NET Entity Framework page at **http://go.microsoft.com/fwlink/?LinkID=267806**.

Using the ADO.NET Entity Data Model Tools

- Tools support:
  - Database-first design by using the Entity Data Model Wizard
  - Code-first design by using the Generate Database Wizard
- They also provide:
  - Designer pane for viewing, updating, and deleting entities and their relationships
  - Update Model Wizard for updating a model with changes that are made to the data source
  - Mapping Details pane for viewing, updating, and deleting mappings

Visual Studio 2017 provides the Entity Data Model Tools that you can use to create and update EDMs in your applications. It supports both database-first design and code-first design:

- *Database-first design*. In database-first design, you design and create your database before you generate your model. This is commonly used when you are developing applications against an existing data source; however, this can limit the flexibility of the application in the long term.
- *Code-first design*. In code-first design, you design the entities for your application and then create the database structure around these entities. Developers prefer this method because it enables you to design your application around the business functionality that you require. However, in reality, you often have to work with an existing data source.

Using the Entity Data Model Tools

Visual Studio 2017 provides the ADO.NET Entity Data Model Tools, which include the Entity Data Model Designer for graphically creating and relating entities in a model and three wizards for working with models and data sources. The following table describes the wizards.

Wizard	Description
Entity Data Model Wizard	Enables you to generate a new conceptual model from an existing data source by using the database-first design method.
Update Model Wizard	Enables you to update an existing conceptual model with changes that are made to the data source on which it is based.
Generate Database Wizard	Enables you to generate a database from a conceptual model that you have designed in the Entity Data Model Designer by using the code-first design method.

When you create a model by using the Entity Data Model Wizard, the model opens in the Designer pane, displaying the entities that the wizard has generated and the relationships between them. You can use this pane to add, modify, and delete entities and relationships.

By default, when you create a model from a database, the Entity Designer automatically generates the mappings from the data source to the conceptual model. You can view, modify, and delete these mappings in the Mapping Details pane.

> **Additional Reading:** For more information about the Entity Data Model Tools, refer to the ADO.NET Entity Data Model Tools page at **http://go.microsoft.com/fwlink/?LinkID=267807**.

Demonstration: Creating an Entity Data Model

In this demonstration, you will use the Entity Data Wizard to generate an EDM for an existing database.

Demonstration steps

You will find the steps in the “Demonstration: Creating an Entity Data Model” section on the following page: **https://github.com/MicrosoftLearning/20483-Programming-in-C-Sharp/blob/master/Instructions/20483C_MOD07_DEMO.md**.

Customizing Generated Classes

- Do not modify the automatically generated classes in a model
- Use partial classes and partial methods to add business functionality to the generated classes

```
public partial class Employee
{
    partial void OnDateOfBirthChanging(DateTime? value)
    {
        if (GetAge() < 16)
        {
            throw new Exception("Employees must be 16 or over");
        }
    }
}
```

When you use the Entity Data Model Wizard to create a model, it automatically generates classes that expose the entities in the model to your application code. These classes contain properties that provide access to the properties in the entities.

The following code example shows an **Employee** class that the Entity Data Model Wizard generated.

Wizard-Generated Classes

```
namespace FourthCoffee.Employees
{
    using System;
    using System.Collections.Generic;
    public partial class Employee
    {
        public int EmployeeID { get; set; }
        public string FirstName { get; set; }
        public string LastName { get; set; }
        public Nullable<System.DateTime> DateOfBirth { get; set; }
        public Nullable<int> Branch { get; set; }
        public Nullable<int> JobTitle { get; set; }
        public virtual Branch Branch1 {get; set; }
        public virtual JobTitle JobTitle1 {get; set; }
    }
}
```

You may find that you want to add custom business logic to the entity classes; however, if at any time in the future you run the Update Model Wizard, the classes will be regenerated, and your code will be overwritten. However, the generated classes are defined as partial classes.

The **partial** keyword allows certain construct's definitions to be split into multiple parts, across multiple source files. Classes, structs, and interfaces can all be partial. When using the **partial** modifier, all definitions of that type must contain the **partial** keyword, and all must exist in the same namespace within the same assembly.

Partial types cannot be split across multiple .dll and .exe files. When the compiler encounters a partial definition, it searches the assembly for others of the same type and combines all their content to a single type. A type may be split in to as many or as few partial definitions as you'd like. A single partial definition is completely valid.

Entity Data Model Wizard will always produce its classes as partial; therefore, you can extend them to add custom functionality to the classes.

For example, if you have a date of birth property in your model, you could write a **GetAge** method in a partial class to enable a run-time calculation of the employee's age.

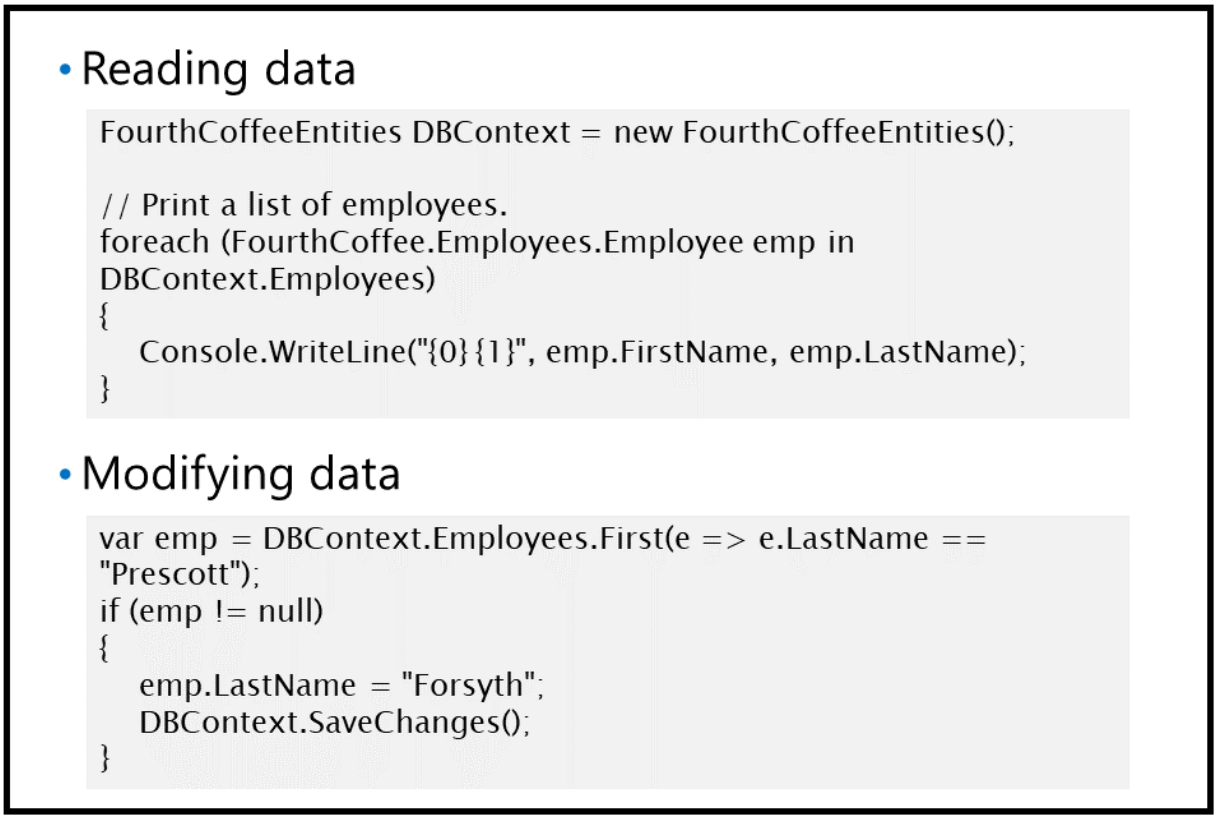
The following code example shows how you can add business logic to a generated class by using a partial class.

Adding Business Logic in a Partial Class

```
public partial class Employee
{
    public int GetAge()
    {
        DateTime DOB = (DateTime)_DateOfBirth;
        TimeSpan difference = DateTime.Now.Subtract(DOB);
        int ageInYears = (int)(difference.Days / 365.25);
        return ageInYears;
    }
}
```

Additional Reading: For more information about partial classes, refer Partial Classes and Methods (C# Programming Guide) at **http://go.microsoft.com/fwlink/?LinkID=267808**.

Reading and Modifying Data by Using the Entity Framework



The automatically generated code files for a model also contains a partial class that inherits from the **System.Data.Entity.DbContext** class. The **DbContext** class provides facilities for querying and working with entity data as objects. It contains a default constructor which initializes the class by using the connection string that the wizard generates in the application configuration file. This defines the data connection and model definition to use. The **DbContext** class also contains a **DbSet** property that exposes a **DbSet(TEntity)** class for each entity in your model. The **DbSet(TEntity)** class represents a typed entity set that you can use to read, create, update, and delete data.

The following code example shows the class for the FourthCoffeeEntities model.

FourthCoffeeEntities Class

```
public partial class FourthCoffeeEntities : DbContext
{
    public FourthCoffeeEntities() :
base("name=FourthCoffeeEntities")
    {
    }
```

```
    public DbSet<Branch> Branches { get; set; }
    public DbSet<Employee> Employees { get; set; }
    public DbSet<JobTitle> JobTitles { get; set; }
}
```

To use the typed entity set, you create an instance of the **DbContext** class and then access the properties by using the standard dot notation.

The following code example shows how to read and update data by using the **DbSet(TEntity)** class.

Reading Data

```
FourthCoffeeEntities DBContext = new FourthCoffeeEntities();
// Print a list of employees.
foreach (FourthCoffee.Employees.Employee emp in DBContext.Employees)
{
    Console.WriteLine("{0} {1}", emp.FirstName, emp.LastName);
}
```

The **DbSet(TEntity)** class implements the **IEnumerable** interface which provides a number of extension methods that enable you to easily locate specific data in the source. For example, the **First** extension method locates the first match for the specified condition, such as a last name of Prescott.

The following code example shows how to use the **First** extension method to locate an employee and then how to update the data by using standard dot notation.

Locating and Modifying Data

```
// Update the employee with a surname of "Prescott."
var emp = DBContext.Employees.First(e => e.LastName == "Prescott");
if (emp != null)
```

```
{
    emp.LastName = "Forsyth";
}
```

> **Additional Reading:** For more information about the **DbSet(Entity)** class, refer to the DbSet(Entity) Class page at **https://aka.ms/moc-20483c-m7-pg1**.
> For more information about the **Enumerable** methods, refer to the Enumerable Methods page at **https://aka.ms/moc-20483c-m7-pg2**.

After you change the data in the model, you must explicitly apply those changes to the data in the data source. You can do this by calling the **SaveChanges** method of the **ObjectContext** object.

The following code example shows how to use the **SaveChanges** method.

Persisting Changes To The Database

```
DBContext.SaveChanges();
```

Demonstration: Reading and Modifying Data in an EDM

In this demonstration, you will use the **ObjectSet(TEntity)** class to read and modify data in an EDM.

Demonstration steps

You will find the steps in the **Demonstration: Reading and Modifying Data in an EDM** section on the following page: **https://github.com/MicrosoftLearning/20483-Programming-in-C-Sharp/blob/master/Instructions/20483C_MOD07_DEMO.md**.

Lesson 2: Querying Data by Using LINQ

As an alternative to using the Entity Framework for querying data, you can use LINQ. This also supports compile-time syntax-checking and type-checking and also uses Microsoft IntelliSense® in Visual Studio. LINQ defines a range of standard query operators that enable you to retrieve exactly the data that you require in a declarative way.

In this lesson, you will learn how to query data and use anonymous methods and how to force query execution to override the default deferred query execution behavior.

Lesson objectives

After completing this lesson, you will be able to:

- Query data.
- Query data by using anonymous types.
- Force query execution.

Querying Data

- Use LINQ to query a range of data sources, including:
  - .NET Framework collections
  - SQL Server databases
  - ADO.NET data sets
  - XML documents
- Use LINQ to:
  - Select data
  - Filter data by row
  - Filter data by column

You can use LINQ to query data from a wide range of data sources, including .NET Framework collections, Microsoft SQL Server® databases, ADO.NET data sets, and XML documents. In fact, you can use it to query any data source that implements the **IEnumerable** interface.

The syntax of all LINQ queries has the same basis, as follows:

from <*variable names*> **in** <*data source*>

select <*variable names*>

However, you can customize this syntax in many ways to retrieve exactly the data that you require in the format that you want. The following code examples all use LINQ to Entities to query data in an EDM; however, the syntax of the query itself does not change if you use a different type of data source.

Selecting Data

The following code example shows how to use a simple **select** clause to return all of the data in a single entity.

Using a select Clause

```
IQueryable<Employee> emps = from e in DBContext.Employees
                                                    select e;
```

The return data type from the query is an **IQueryable<Employee>**, enabling you to iterate through the data that is returned.

Filtering Data by Row

The following code example shows how to use the **where** keyword to filter the returned data by row to contain only employees with a last name of Prescott.

Using a where Clause

```
string _LastName = "Prescott";
IQueryable<Employee> emps = from e in DBContext.Employees
                                                    where
e.LastName == _LastName
                                                    select e;
```


Filtering Data by Column

The following code example shows how to declare a new type in which to store a subset of columns that the query returns; in this case, just the **FirstName** and **LastName** properties of the **Employee** entity.

Using a New Class to Return a Subset of Columns

```
private class FullName
{
    public string Forename { get; set; }
    public string Surname { get; set; }
}
private void FilteringDataByColumn()
{
    IQueryable<FullName> names = from e in DBContext.Employees
                                                        select new
FullName { Forename = e.FirstName, Surname = e.LastName };
}
```


Working with the Results

To then work with the data that is returned from any of these queries, you use dot notation to access the properties of the members of the **IQueryable<>** type, as the following code example shows.

Accessing the Returned Data

```
foreach (var emp in emps)
{
    Console.WriteLine("{0} {1}", emp.FirstName, emp.LastName);
}
```

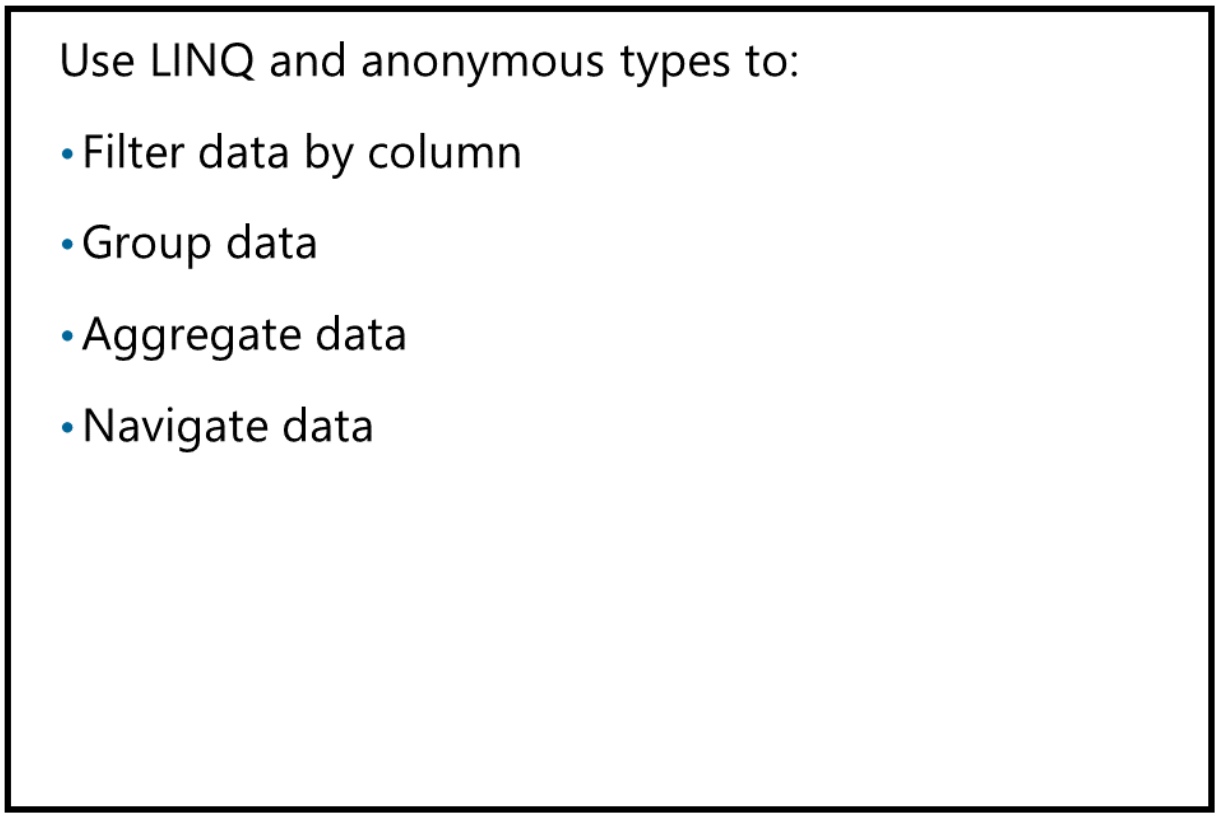

Demonstration: Querying Data

In this demonstration, you will use LINQ to Entities to query data.

Demonstration steps

You will find the steps in the **Demonstration: Querying Data** section on the following page: **https://github.com/MicrosoftLearning/20483-Programming-in-C-Sharp/blob/master/Instructions/20483C_MOD07_DEMO.md**.

Querying Data by Using Anonymous Types



In the examples in the previous topic and demonstration, the return data was always stored in a strongly typed **IQueryable<Type>** variable; however, in the filtering by column scenario, it is necessary to define the type containing a subset of columns

before defining the query. Although this is a perfectly valid way of working, it can become tedious to explicitly define multiple classes.

You can use anonymous types to store the returned data by declaring the return type as an implicitly typed local variable, a **var**, and by using the **new** keyword in the **select** clause to create the instance of the type.

Filtering Data by Column

The following code example shows how to use the **var** data type and the **new** keyword in the **select** clause to filter the returned data by column.

Using an Anonymous Type to Return a Subset of Columns

```
var names = from e in DBContext.Employees
                                    select new { e.FirstName,
e.LastName };
```

Anonymous types enable you to perform more complex queries in LINQ.

Grouping Data

The following code example shows how to use a **group** clause to group the returned employees by their job title ID.

Using a group Clause

```
var emps = from e in DBContext.Employees
                   group e by e.JobTitle into eGroup
                   select new { Job = eGroup.Key, Names = eGroup };
```

Aggregating Data

The following code example shows how to use a **group** clause with an aggregate function to count the number of employees with each job title.

Using a group Clause with an Aggregate Function

```
var emps = from e in DBContext.Employees
                   group e by e.JobTitle into eGroup
                   select new { Job = eGroup.Key, CountOfEmployees =
eGroup.Count() };
```


Navigating Data

The following code example shows how to use navigation properties to retrieve data from the **Employees** entity and the related **JobTitles** entity.

Using Dot Notation to Navigate Related Entities

```
var emps = from e in DBContext.Employees
                   select new
                   {
                       FirstName = e.FirstName, LastName =
e.LastName, Job = e.JobTitle1.Job
                   };
```


Demonstration: Querying Data by Using Anonymous Types

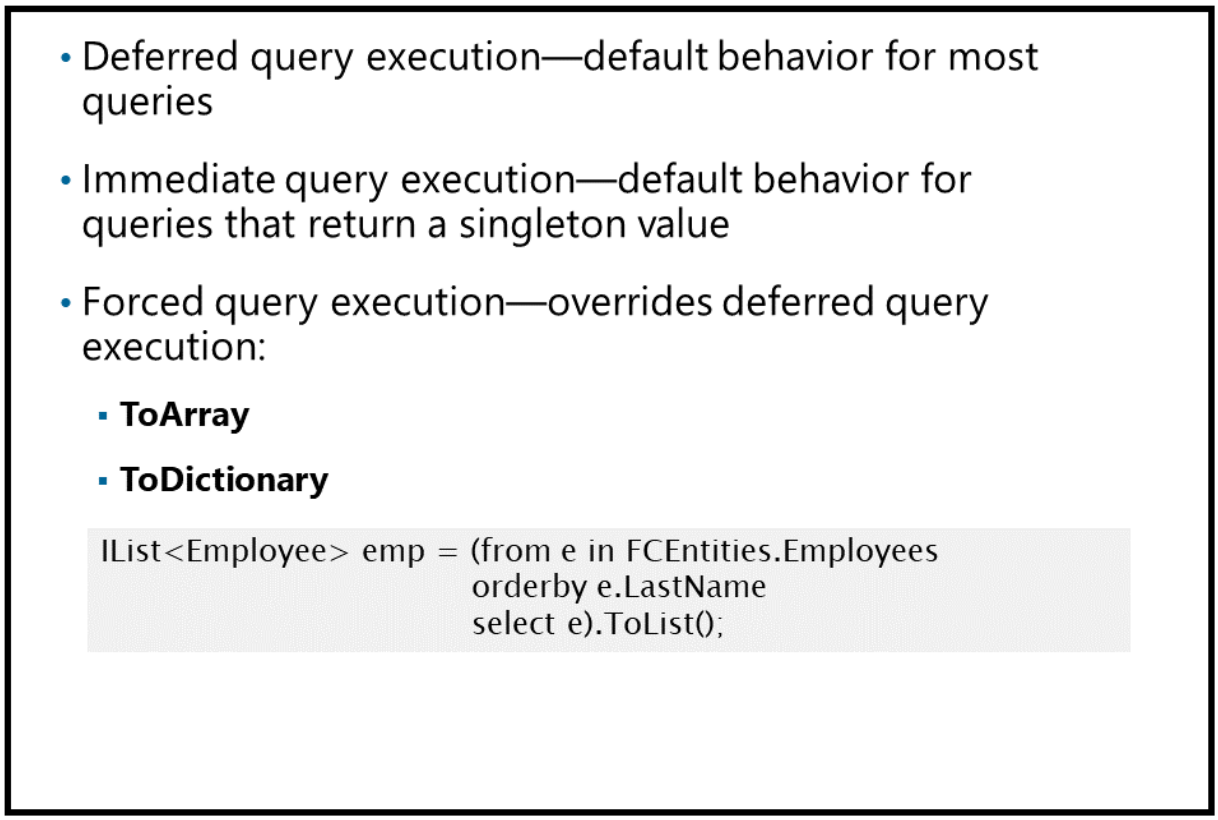
In this demonstration, you will use LINQ to Entities to query data by using anonymous types.

Demonstration steps

You will find the steps in the **Demonstration: Querying Data by Using Anonymous Types** section on the following page: **https://github.com/MicrosoftLearning/20483-**

Programming-in-C-Sharp/blob/master/Instructions/20483C_MOD07_DEMO.md.

Forcing Query Execution



By default, when you define a LINQ query that returns a sequence of values, it is not run until you actually try to use some of the returned data. This feature is known as *deferred query execution* and ensures that you can create a query to retrieve data in a multiple-user scenario and know that whenever it is executed you will receive the latest information.

In the following code example, the query is not actually executed until the start of the **foreach** block.

Deferred Query Execution

```
IQueryable<Employee> emps = from e in DBContext.Employees
                                                    select e;
foreach (var emp in emps)
{
    Console.WriteLine("{0} {1}", emp.FirstName, emp.LastName);
}
```

Conversely, when you define a LINQ query that returns a singleton value, for example, an **Average**, **Count**, or **Max** function, the query is run immediately. This is known as *immediate query execution* and is necessary in the singleton result scenario because the query must produce a sequence to calculate the singleton result.

You can override the default deferred query execution behavior for queries that do not produce a singleton result by calling one of the following methods on the query:

- **ToArray**
- **ToDictionary**
- **ToList**

In the following code example, the query is executed immediately after it is defined.

Forcing Query Execution

```
IList<Employee> emps = (from e in DBContext.Employees
                                                    select
e;).ToList()
foreach (var emp in emps)
{
    Console.WriteLine("{0} {1}", emp.FirstName, emp.LastName);
}
```

Demonstration: Retrieving and Modifying Grade Data Lab

In this demonstration, you will learn about the tasks that you will perform in the lab for this module.

Demonstration steps

You will find the steps in the **Demonstration: Retrieving and Modifying Grade Data Lab** section on the following page: **https://github.com/MicrosoftLearning/20483-Programming-in-C-Sharp/blob/master/Instructions/20483C_MOD07_DEMO.md**.

Lab: Retrieving and Modifying Grade Data

Scenario

You have been asked to upgrade the prototype application to use an existing SQL Server database. You begin by working with a database that is stored on your local machine and decide to use the Entity Data Model Wizard to generate an EDM to access the data. You will need to update the data access code for the Grades section of the application, to display grades that are assigned to a student and to enable users to assign new grades. You also decide to incorporate validation logic into the EDM to ensure that students cannot be assigned to a full class and that the data that users enter when they assign new grades conforms to the required values.

Objectives

After completing this lab, you will be able to:

- Create an EDM from an existing database.
- Update data by using the .NET Entity Framework.
- Extend an EDM to validate data.

Lab setup

Estimated Time: 75 minutes

You will find the high-level steps on the following page: **https://github.com/MicrosoftLearning/20483-Programming-in-C-Sharp/blob/master/Instructions/20483C_MOD07_LAB_MANUAL.md**.

You will find the detailed steps on the following page: **https://github.com/MicrosoftLearning/20483-Programming-in-C-**

Sharp/blob/master/Instructions/20483C_MOD07_LAK.md.

Exercise 1: Creating an Entity Data Model from The School of Fine Arts Database

Scenario

In this exercise, you will use the Entity Data Model Wizard to generate an EDM from the **SchoolGradesDB** SQL Server database and then review the model and the code that the wizard generates.

> **Result**: After completing this exercise, the prototype application should include an EDM that you can use to access The School of Fine Arts database.

Exercise 2: Updating Student and Grade Data by Using the Entity Framework

Scenario

In this exercise, you will add functionality to the prototype application to display the grades for a user. The grade information in the database stores the subject ID for a grade, so you will add code to the application to convert this to the subject name for display purposes. You will also add code to display the **Add Grade** view to the user and then use the information that the user enters to add a grade for the current student. Finally, you will run the application and verify that the grade display and grade-adding functionality works as expected.

> **Result**: After completing this exercise, users will be see the grades for the current student and add new grades.

Exercise 3: Extending the Entity Data Model to Validate Data

Scenario

In this exercise, you will update the application to validate data that the user enters.

First, you will add code to check whether a class is full before enrolling a student and throw an exception if it is. Then you will add validation code to check that a user enters a valid date and assessment grade when adding a grade to a student. Finally, you will run the application and verify that the data validation works as expected.

Result: After completing this exercise, the application will raise and handle exceptions when invalid data is entered.

Module review and takeaways

In this module, you learned how to create and use EDMs and how to query many types of data by using LINQ.

Review Question(s)

Check Your Knowledge

Discovery

What advantages does LINQ provide over traditional ways of querying data?

Show solution Reset

Check Your Knowledge

Select the best answer

Fourth Coffee wants you to add custom functionality to an existing EDM in its Coffee Sales application. You need to write a method for adding a new product to the application. In which of the following locations should you write your code?

In the relevant generated class in the EDM project.

In a partial class in the EDM project.

Check answer Show solution Reset