ENTITTY FRAMEWORK

You can query EDM mainly by three ways, 1) LINQ to Entities 2) Entity SQL 3) Native SQL.

1) LINQ to Entities: L2E query syntax is easier to learn than Entity SQL. You can use your LINQ skills for querying with EDM. Following code snippet shows how you can query with EDM created in previous step.

```
   //Querying with LINQ to Entities
    using (var objCtx = new SchoolDBEntities())
    {
        var schoolCourse = from cs in objCtx.Courses
                           where cs.CourseName == "Course1"
                           select cs;
        Course mathCourse = schoolCourse.FirstOrDefault<Course>();
        IList<Course> courseList = schoolCourse.ToList<Course>();

        string courseName = mathCourse.CourseName;
    }
```

First, you have to create object of context class which is SchoolDBEntities. You should initialize it in "using()" so that once it goes out of scope then it will automatically call Dispose() method of context class. Now, you can use LINQ with context object. LINQ query will return IQueryable<> object but underlying type of var will be ObjectQuery. You can then get single object using FirstOrDefault<>() or list of objects by using ToList<>().

2) Entity SQL: Another way to create a query, instead of LINQ to Entities, is by using the Entity Framework's Object Services directly. You can create an ObjectQuery directly combined with the Entity Framework's T-SQL-like query language, called Entity SQL, to build the query expression.

Following code snippet shows same query result as L2E query above.

```
    //Querying with Object Services and Entity SQL
    using (var objCtx = new SchoolDBEntities())
    {
        string sqlString = "SELECT VALUE cs FROM
SchoolDBEntities.Courses
                            AS cs WHERE cs.CourseName == 'Maths'";
        ObjectQuery<Course> course =
objCtx.CreateQuery<Course>(sqlString);
        Course coursename1 = course.FirstOrDefault<Course>();
    }
```

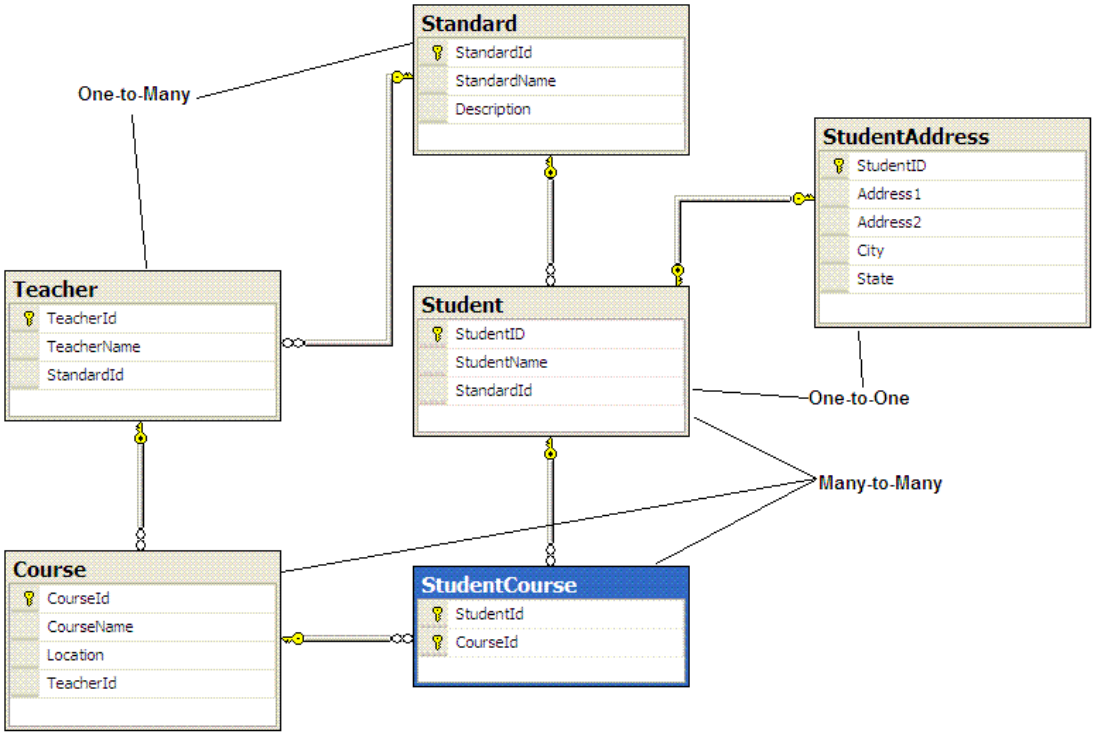
Here also, you have to create object of context class. Now you have to write SQL query as per Entity to SQL syntax and pass it in CreateQuery<>() method of context object. It will return ObjectQuery<> result. You can then single object using FirstOrDefault<>() or list of object by using ToList<>().

3) Native SQL In the Entity Framework v4 new methods ExecuteFunction(), ExecuteStoreQuery() and ExecuteStoreCommand() were added to the class ObjectContext. So you can use these methods to execute Native SQL to the database as following:

```
    //Querying with native sql
    using (var objCtx = new SchoolDBEntities())
        {
            //Inserting Student using ExecuteStoreCommand
            int InsertedRows = objCtx.ExecuteStoreCommand("Insert into
Student(StudentName,StandardId) values('StudentName1',1)");

            //Fetching student using ExecuteStoreQuery
            var student = objCtx.ExecuteStoreQuery<Student>("Select *
from Student where StudentName = 'StudentName1'", null).ToList();

        }
```

Entity Relationships:

You can have three types of relations in EDM as in database. 1) One to One 2) One to Many 3) Many to Many. Let's examine database table design before going into relationships in EDM. Following figure is a database diagram of SchoolDB used in this tutorial.

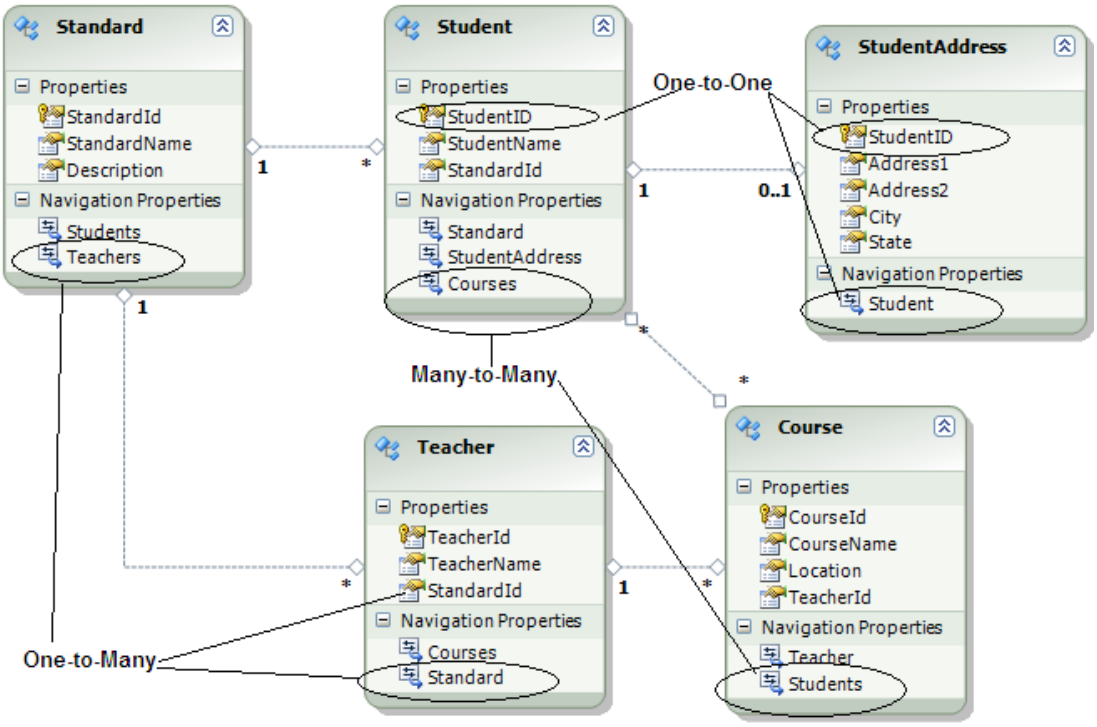


As you can see in the figure, student and StudentAddress have One-to-One relationshionship where each student has zero or one address.

Standard and Teacher has One-to-Many relationship where one standard can have many teachers but one teacher can't go to many standard (Standard is a classroom).

Student and Course has Many-to-Many relationships by using StudentCourse table. StudentCourse consists primary key of both the tables and thus it makes relation Many-to-Many.

When you create ADO.NET Entity Data Model from this database using 'Generate from existing database' option, it will create following entities and relationships into EDM:



As you can see in the above figure, Student and StudentAddress has One to One relationship (zero or one). StudentAddress entity has StudentId property as PK which makes it One-to-One relationship. Standard and teach has One-to-Many relation marked by multiplicity where 1 is for One and * is for Many.

Standard entity has navigation property "Teachers" which indicates that one Standard can have list of teachers and Teacher entity has "Standard" navigation property which indicates that Teacher is associated with one Standard. This makes it One-to-Many relationship.

Student and Course have Many-to-Many relationships marked by * multiplicity, but it doesn't display entityset for middle table "StudentCourse" where primary key of both tables will be stored. This is because The EDM represents many-to-many relationships by not having entityset for the joining table in CSDL, instead it manages through mapping. It can do this only when the join table has just the relevant keys and no additional fields. If the join tables had additional properties, such as DateCreated, the EDM would have created entities for them and you have to manage Many-to-Many relationship entities manually.

So now let's see how Many-to-Many relationship is being managed in EDM.

Open EDM in XML view. You can see that SSDL has StudentCourse entityset but CSDL doesn't have StudentCourse entityset instead it's being mapped in navigation property of Student and Course entity. In MSL (C-S Mapping), it has a mapping between Student and Course into StudentCourse table in <AssociationSetMapping/>

```
<AssociationSetMapping Name="StudentCourse" TypeName="SchoolDBModel.StudentCourse" StoreEntitySet="StudentCourse">
  <EndProperty Name="Course">
    <ScalarProperty Name="CourseId" ColumnName="CourseId" />
  </EndProperty>
  <EndProperty Name="Student">
    <ScalarProperty Name="StudentID" ColumnName="StudentId" />
  </EndProperty>
</AssociationSetMapping>
```

Thus Many-to-Many relationship is being managed in C-S mapping in EDM. So when you add student in course or Course in Student entity and when you save it then it will insert PK of added student and course in StudentCourse table. So this mapping not only enables a convenient association directly between the two entities, but also manages querying, inserts, and updates across this join.

But remember EDM does this only when joining table has PK columns for both tables. If you have some other columns in joining table then EDM will treat as normal entity and you have to use 'Join' in your query to fetch the data.

Entity Graph:

When an entity has relation with other entities then it called entity graph because more entities are involved, for example Student entity graph includes many other entities like Standard, StudentAddress & Course.

Querying Entity Graph:

Projection:

Projection is a process of selecting data in different shape rather than specific entity being queried. There are many ways of projection. Let's see some projection style:

If you want to get the single student object when there are many students whose name is "Student1" in the database then use FirstOrDefault<>

```
var student = (from s in ctx.Students
               where s.StudentName == "Student1"
               select s).FirstOrDefault<Student>();
```

If you want to list of all students whose name is "Student1" (provided there are many students has same name) then use ToList<>:

```
var studentList = (from s in ctx.Students
                   where s.StudentName == "Student1"
                   select s).ToList<Student>();
```

If you want to group students by standardId then use group:

```
var students = from s in ctx.Students
                   groupby s.StandardId into studentsByStandard
                   select studentsByStandard;
```

If you want to get the list of students sorted by StudentName then use OrderBy:

```
    var student1 = from s in ctx.Students
                   orderby s.StudentName ascending
                   select s;
```

If you want to get only StudentName, StandardName and list of Courses for that student in single object then write following projection:

```
    var projectionResult = from s in ctx.Students
                       where s.StudentName == "Student1"
                       select new {
                       s.StudentName, s.Standard.StandardName,
s.Courses
                       };
```

Type of projectionResult in above query will be anonymous type because there is no class/entity which has these properties. So compiler will mark it as anonymous.

So this way you can do projection of result the way you want data. There are different other ways of projection but all projection styles requires knowledge of LINQ.

Significance of SaveChanges:

SaveChanges method of ObjectContext is a gateway to persist all changes made to entities to the database. When you call ObjectContext.SaveChanges(), it performs insert, update or delete operation on the database based on EntityState of the entities.

Following code shows how you can persist modification made to the Student entities of SchoolDB EDM created either with EntityObject entities or POCO Proxy entities.

```
      //Update entity using SaveChanges method
      using (SchoolEntities ctx = new SchoolDBEntities())
            {
                var stud = (from s in ctx.Students
                            where s.StudentName == "Student1"
                            select s).FirstOrDefault();

                stud.StudentName = "Student2";

                int num = ctx.SaveChanges();
            }
```

As you can see in above code, we fetch the single Student entity whose name is “Student1” and then we change the StudentName property to “Student2”. It saves this modification to the database when we do ctx.SaveChanges(). This method also returns the number of rows updated in the database.

SaveChanges also accepts SaveOptions parameter. SaveOption is an Enum which has three values:

1. AcceptAllChangesAfterSave: After saving entities values to the database, context change entity states. Added and Modified entities become Unchanged and deleted entities are removed from the context.
2. DetectChangesBeforeSave: It tells context to detect changes before saving.
3. None: Neither AcceptAllChangesAfterSave or DetectChangesBeforeSave occurs

So this way SaveChanges method is the most important method in the EntityFramework.

In this chapter we will learn how to add and save single entity using DBContext which in-tern insert single row in database table.

We will see how to add single 'Standard' entity:

```
    // create new Standard entity object
    var newStandard = new Standard();

    // Assign standard name
    newStandard.StandardName = "Standard 1";

    //create DBContext object
    using (var dbCtx = new SchoolDBEntities())
    {
        //Add standard object into Standard DBset
        dbCtx.Standards.Add(newStandard);
        // call SaveChanges method to save standard into database
        dbCtx.SaveChanges();
    }
```

As we have learned about DBContext.Entry method in one of the previous chapters. Entry method is useful to get any DBEntityEntry for given Entity. DBEntityEntry provides access to information and control of entities that are being tracked by the DbContext.

As a general rule, we can add any existing Entity and mark it as deleted as following:

```
    using (var dbCtx = new SchoolDBEntities())
        {

            //if already loaded in existing DBContext then use
Set().Remove(entity) to delete it.
            var newtchr = dbCtx.Teachers.Where(t => t.TeacherName ==
"New teacher4")
```

```
                                    .FirstOrDefault<Teacher>();
            dbCtx.Set(Teacher).Remove(newtchr);

            //Also, you can mark an entity as deleted
            //dbCtx.Entry(tchr).State =
System.Data.EntityState.Deleted;

            //if not loaded in existing DBContext then use following.
            //dbCtx.Teachers.Remove(newtchr);

            dbCtx.SaveChanges();
        }
```

You can delete an entity by two way:

1. If DBContext already loaded entity which you are going to delete then you can delete it using Set method:

   dbCtx.Set(Teacher).Remove(newtchr);

   or

   dbCtx.Teachers.Remove(newtchr);

2. If entity is not loaded in the DBContext then you can use Entry method to mark entity as deleted:

   dbCtx.Entry(tchr).State = System.Data.EntityState.Deleted;

So this way you can easily delete an entity.

Delete Entity using DBContext

As we have learned about DBContext.Entry method in one of the previous chapters. Entry method is useful to get any DBEntityEntry for given Entity. DBEntityEntry provides access to information and control of entities that are being tracked by the DbContext.

As a general rule, we can add any existing Entity and mark it as deleted as following:

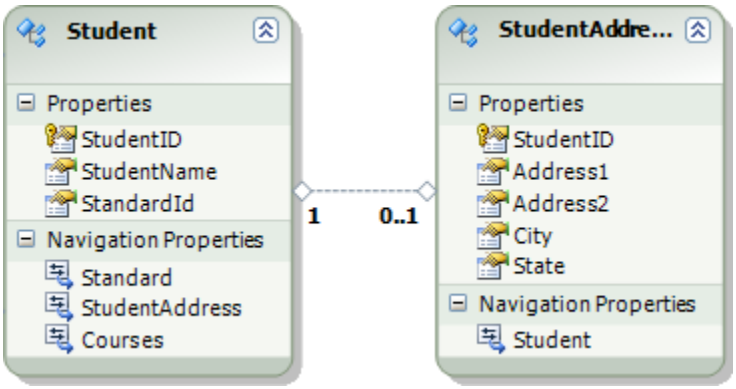
```
    using (var dbCtx = new SchoolDBEntities())
        {

            //if already loaded in existing DBContext then use
Set().Remove(entity) to delete it.
            var newtchr = dbCtx.Teachers.Where(t => t.TeacherName ==
"New teacher4")
                                    .FirstOrDefault<Teacher>();
            dbCtx.Set(Teacher).Remove(newtchr);

            //Also, you can mark an entity as deleted
```

```
            //dbCtx.Entry(tchr).State =
System.Data.EntityState.Deleted;

            //if not loaded in existing DBContext then use following.
            //dbCtx.Teachers.Remove(newtchr);

            dbCtx.SaveChanges();
        }
```

Above code results in following delete query which deletes the row from Teacher table.

Add One-to-One Relationship Entity Graph using DBContext

We will see how to add new Student and StudentAddress entities which has One-to-One relationship that results in new rows in Student and StudentAddress table.



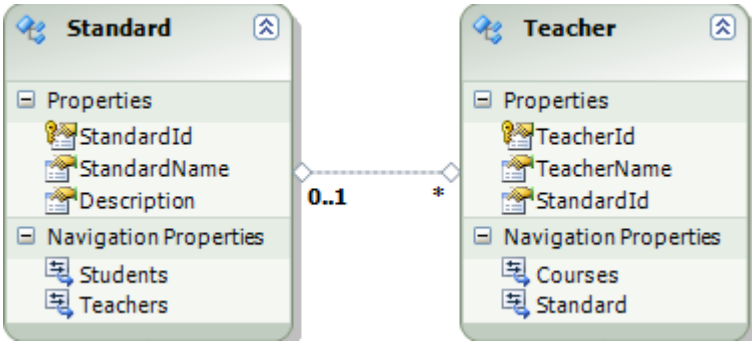
[Student and StudentAddress has One-to-One relationship]

```
            // create new student entity object
            var student = new Student();

            // Assign student name
            student.StudentName = "New Student1";

            // Create new StudentAddress entity and assign it to
student entity
            student.StudentAddress = new StudentAddress() { Address1 =
"Student1's Address1",
                    Address2 = "Student1's Address2", City =
"Student1's City",
                    State = "Student1's State" };
            //create DBContext object
            using (var dbCtx = new SchoolDBEntities())
            {
                //Add student object into Student's EntitySet
                dbCtx.Students.Add(student);
```

```
                // call SaveChanges method to save student &
StudentAddress into database
                dbCtx.SaveChanges();
```

Add One-to-Many Relationship Entity Graph using DBContext

We will see how to add new Standard and Teacher entities which has One-to-Many relationship which results in single entry in 'Standard' database table and multiple entry in 'Teacher' table.

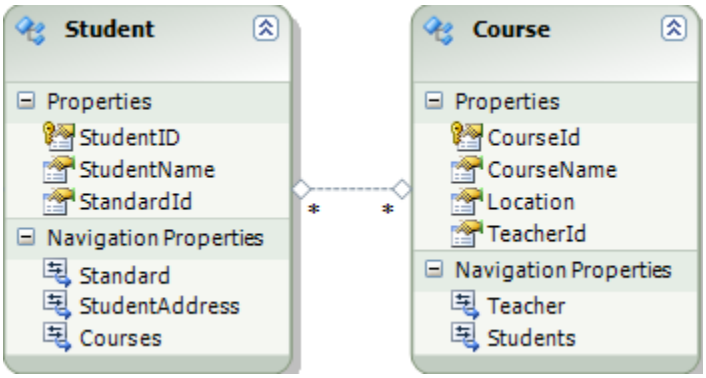


[Standard and Teacher has One-to-Many relationship]

```
    //Create new standard
    var standard = new Standard();
    standard.StandardName = "Standard1";

    //create three new teachers
    var teacher1 = new Teacher();
    teacher1.TeacherName = "New Teacher1";

    var teacher2 = new Teacher();
    teacher2.TeacherName = "New Teacher2";

    var teacher3 = new Teacher();
    teacher3.TeacherName = "New Teacher3";

    //add teachers for new standard
    standard.Teachers.Add(teacher1);
    standard.Teachers.Add(teacher2);
    standard.Teachers.Add(teacher3);

    using (var dbCtx = new SchoolDBEntities())
    {
        //add standard entity into standards entitySet
        dbCtx.Standards.Add(standard);
        //Save whole entity graph to the database
        dbCtx.SaveChanges();
    }
```

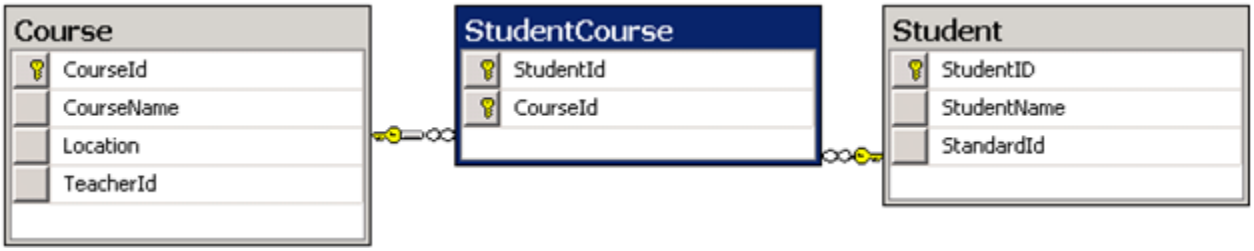
Add Many-to-Many Relationship Entity Graph using DBContext

We will see how to add new courses in student's course collection. Student and Course has Many-to-Many relationship which results in insert new rows in Student and StudentCourse tables.



[Student and Course has Many-to-Many relationship]

If you see database design, actually there are three tables participates in Many-to-Many relationship between Student and Course, Student, Course and StudentCourse tables. StudentCourse table consist StudentID and CourseId where both StudentId and CourseId is composite key (combined primary key).



Now let's see code to add these entities into DBContext:

```
//Create student entity
var student1 = new Student();
student1.StudentName = "New Student2";

//Create course entities
var course1 = new Course();
course1.CourseName = "New Course1";
course1.Location = "City1";

var course2 = new Course();
course2.CourseName = "New Course2";
course2.Location = "City2";

var course3 = new Course();
course3.CourseName = "New Course3";
course3.Location = "City1";

// add multiple courses for student entity
student1.Courses.Add(course1);
```

```
    student1.Courses.Add(course2);
    student1.Courses.Add(course3);

    using (var dbCtx = new SchoolDBEntities())
    {
        //add student into DBContext
        dbCtx.Students.Add(student1);
        //call SaveChanges
        dbCtx.SaveChanges();
    }
```

SaveChanges results in seven inserts query, 1 for student, 3 for Course and 3 for StudentCourse table.

Update Entity using DBContext

As we have learned about DBContext.Entry method in previous chapter, Entry method is useful to get any DBEntityEntry for given Entity. DBEntityEntry provides access to information and control of entities that are being tracked by the DbContext.

As a general rule, we can add any existing modified Entity and mark it as modified as following:

dbCtx.Entry(Entity).State = System.Data.EntityState.Modified;

Let's see how to update an existing single 'Standard' entity:

```
    Student stud ;
    // Get student from DB
    using (var ctx = new SchoolDBEntities())
    {
        stud = ctx.Students.Where(s => s.StudentName == "New
Student1").FirstOrDefault<Student>();
    }

    // change student name in disconnected mode (out of DBContext
scope)
    if (stud != null)
    {
        stud.StudentName = "Updated Student1";
    }

    //save modified entity using new DBContext
    using (var dbCtx = new SchoolDBEntities())
    {
        //Mark entity as modified
        dbCtx.Entry(stud).State = System.Data.EntityState.Modified;
        dbCtx.SaveChanges();
    }
```

As you see in the above code snippet, we are doing following steps:

1. Get the existing student
2. Change student name out of DBContext scope (disconnected mode)
3. We pass modified entity into Entry method to get its DBEntityEntry object and then marking its state as Modified
4. Calling SaveChanges to update student information into the database.

Update One-to-Many Entities

Connected Scenario:

Following code shows how we can save modified Standard and Teachers entity graph which has One-to-Many relationship to the database in connected scenario:

```
using (var ctx = new SchoolDBEntities())
           {
            //fetching existing standard from the db
            Standard std = (from s in ctx.Standards
                            where s.StandardName == "standard3"
                            select s).FirstOrDefault<Standard>();
            std.StandardName = "Updated standard3";
            std.Description = "Updated standard";
            //getting first teacher to be removed
            Teacher tchr = std.Teachers.FirstOrDefault<Teacher>();
         //removing teachers (enable cascading delete for the teacher)
            if (tchr != null)
                ctx.Teachers.DeleteObject(tchr);
          Teacher stdTeacher = std.Teachers.FirstOrDefault<Teacher>();
            if (stdTeacher != null)
                stdTeacher.TeacherName = "Updated Teacher";
             Teacher newTeacher = new Teacher();
             newTeacher.TeacherName = "New Teacher";

            std.Teachers.Add(newTeacher);
```

```
            Teacher existingTeacher = (from t in ctx.Teachers
      e t.StandardId != std.StandardId
              select t).FirstOrDefault<Teacher>();
            if (existingTeacher != null)
                std.Teachers.Add(existingTeacher);

            ctx.SaveChanges();
        }
```

Update Many-to-Many Entities

Connected Scenario:

Following code saves modified Student and Courses (for that student) to the database:

```
using (var ctx = new SchoolDBEntities())
```

```
            {
                Student student = (from s in ctx.Students
                                   where s.StudentName == "Student3"s
                                  select s).FirstOrDefault<Student>();

                student.StudentName = "Updated Student3";

              Course cours = student.Courses.FirstOrDefault<Course>();
                //removing course from student
                student.Courses.Remove(cours);
```

```
                ctx.SaveChanges();
            }
```