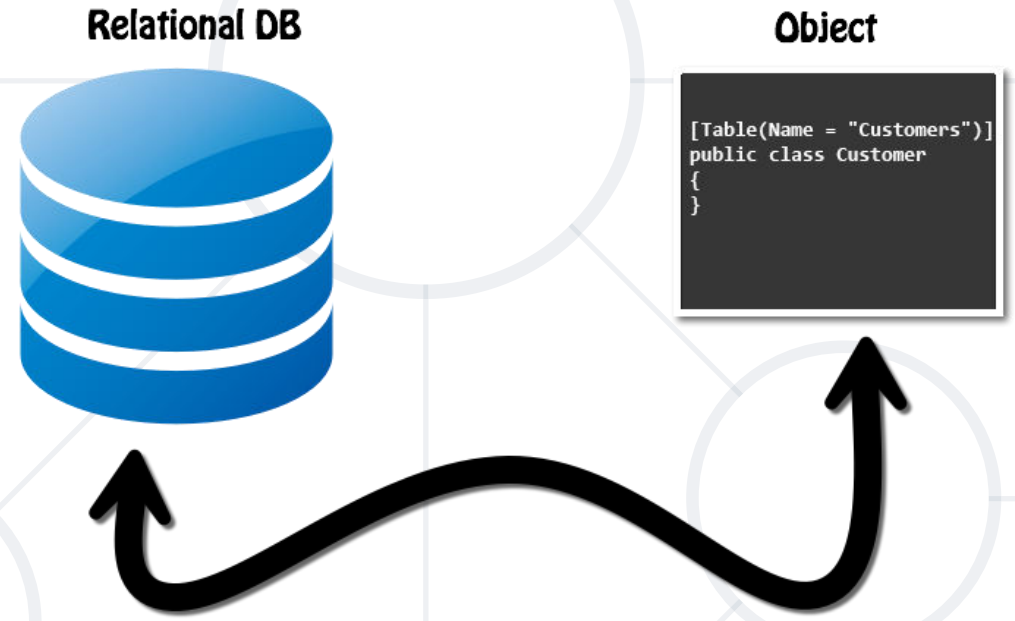
ORM Fundamentals

The ORM Concept, Config, CRUD Operations



SoftUni Team

Technical Trainers

Software University

SoftUni Foundation

Software University

http://softuni.bg

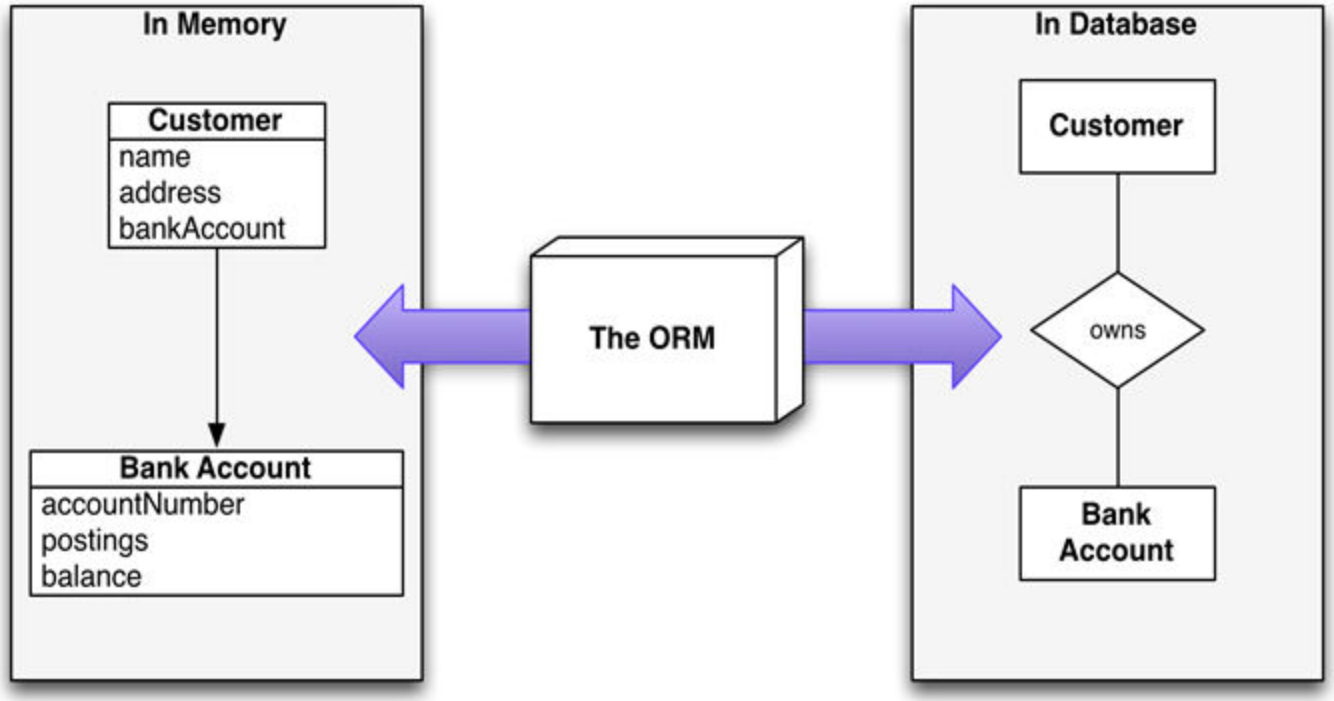
Table of Contents

Questions

SoftUni Foundation

sli.do

#csharp-db

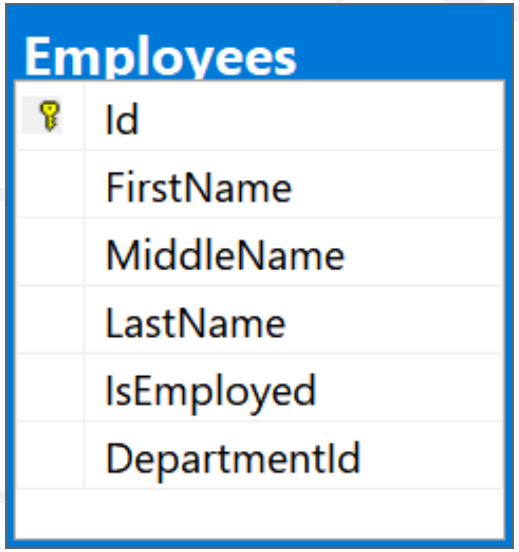


Introduction to ORM

Object-Relational Mapping

What is ORM?

- **Object-Relational Mapping** (ORM) allows manipulating databases **using common classes and objects**
  - **Database Tables → C#/Java/etc. classes**



```
public class Employee
{
    public int Id { get; set; }
    public string FirstName { get; set; }
    public string MiddleName { get; set; }
    public string LastName { get; set; }
    public bool IsEmployed { get; set; }
    public Department Department { get; set; }
}
```

ORM Frameworks: Features

- **ORM frameworks** typically **provide** the following functionality:
  - **Automatically generate SQL** to perform data operations

```
database.Employees.Add(new Employee
{
   FirstName = "Gosho",
   LastName = "Ivanov",
   IsEmployed = true    });
```

```
INSERT INTO Employees
(FirstName, LastName, IsEmployed)
VALUES
('Gosho', 'Ivanov', 1)
```

  - **Create object model from database schema** (DB First model)
  - **Create database schema from object model** (Code First model)
  - **Query data by object-oriented API** (e.g. LINQ queries)

ORM Advantages and Disadvantages

SoftUni Foundation

- Object-relational mapping (ORM) **advantages**:
  - Developer productivity: **writing less code**
  - Abstract from differences between object and relational world
  - **Manageability of the CRUD operations** for complex relationships
  - **Easier maintainability**
- **Disadvantages**:
  - **Reduced performance** (due to overhead or autogenerated SQL)
  - **Reduces flexibility** (some operations are hard to implement)

Custom ORM Framework

Overview and Features

MiniORM Core: Overview

- Designed after **Entity Framework Core**
- Provides **LINQ-based data queries** and **CRUD operations**
- **Change tracking** of in-memory objects
- Maps navigation properties
- Maps collections
  - **One-to-many**, **Many-to-many**, etc.

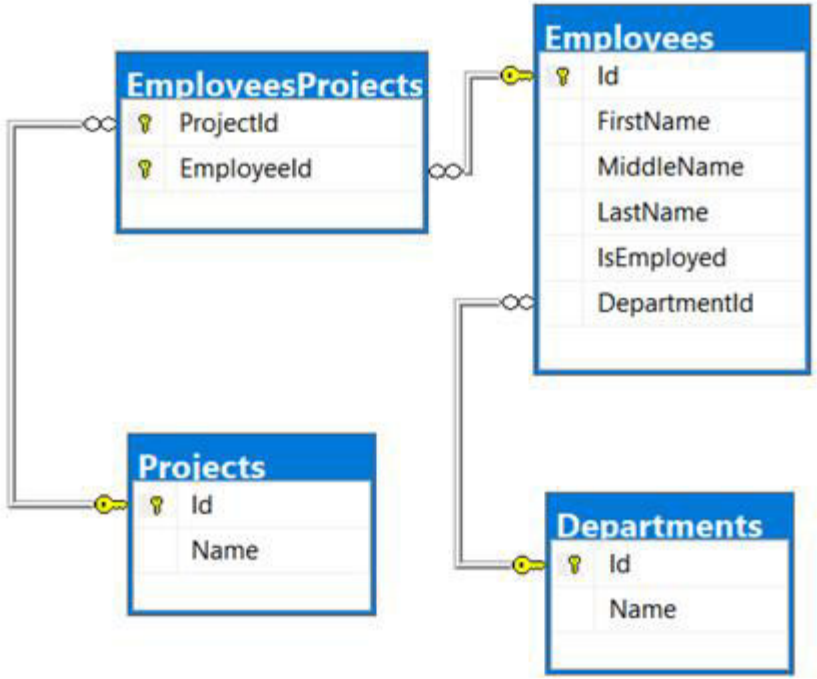
MiniORM Core Workflow: Overview

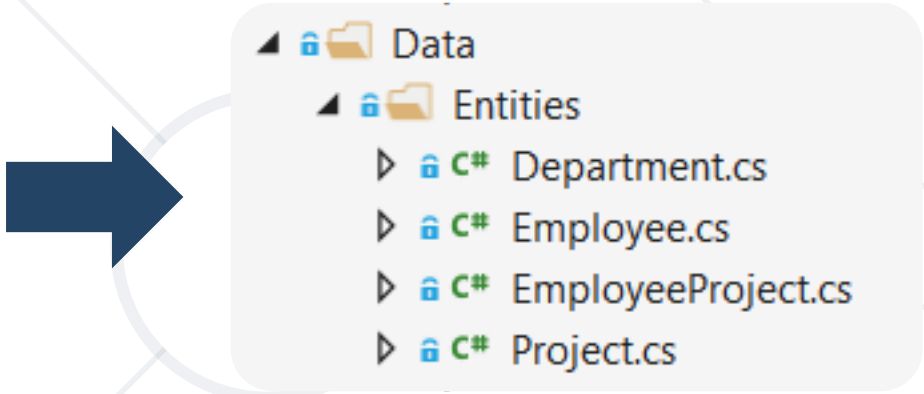
SoftUni Foundation

- **Define data model** (database-first)
  - Entity Classes
  - **DbContext** (with **DbSets**)
- Initialize **DbContext**
  - Using connection string
- Query data using context
- Manipulate data (add/remove/update entities)
- Context gets persisted into database

Database First Model

- **Database First model** - models the entity classes after the database





MiniORM Components

- The **DbContext** class
  - Holds the **database connection** and the **DB Sets**
  - Provides **LINQ-based** data access
  - Provides **change tracking**, and an API for **CRUD** operations
- **DB Sets**
  - Hold **entities** (objects with their attributes and relations)
  - Each database **table** is typically mapped to a single **C# class**

MiniORM Components (2)

- **Associations** (relationship mappings)
  - An association is **a primary key** / **foreign key**-based **relationship** between two entity classes
  - Allows **navigation** from one entity to another

```
var courses = student.Courses.Where(…);
```

  - MiniORM **supports one-to-one**, **one-to-many** and **many-to-many** relationships

```
public class Employee
{
    public int Id { get; set; }
    public string FirstName { get; set; }
    public string MiddleName { get; set; }
    public string LastName { get; set; }
    public bool IsEmployed { get; set; }
    public Department Department { get; set; }
}
```

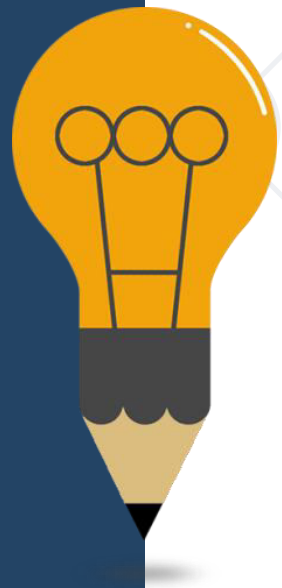
Entity Classes

Data Holders

Entity Classes

SoftUni Foundation

- **Entity classes** are regular **C# classes**
- Used for **storing** the **data** from the DB **in-memory**



Employees
Id
FirstName
MiddleName
LastName
IsEmployed
DepartmentId

```
public class Employee
{
    public int Id { get; set; }
    public string FirstName { get; set; }
    public string MiddleName { get; set; }
    public string LastName { get; set; }
    public bool IsEmployed { get; set; }
    public Department Department { get; set; }
}
```

Entity Classes: Navigation Properties

- Reference type properties
- Point to relevant object, connected by foreign key
- Set by the framework
- Example: Employee's Department:

```
public class Employee {
    public int Id { get; set; }
    ...
    [ForeignKey(nameof(Department))]
    public int DepartmentId { get; set; }
    public Department Department { get; set; }
}
```

Entity Classes: Navigation Properties (2)

- Navigation Properties can also be collections
- Usually of type **ICollection<T>**
- Holds all of the objects whose **foreign keys** are the same as the entity's **primary key**
- Set by the ORM framework

```
public class Department
{
  public int Id { get; set; }
  ...
  public ICollection<Employee> Employees { get; set; }
}
```

```
public class DbSet<TEntity> : ICollection<TEntity>
    where TEntity : class, new()
{
    internal DbSet([NotNull] IEnumerable<TEntity> entities)...

    internal ChangeTracker<TEntity> ChangeTracker { get; set; }

    internal IList<TEntity> Entities { get; set; }

    public void Add(TEntity item)...

    public void Clear()...

    public bool Contains(TEntity item) => this.Entities.Contains(item);
```

DbSet<T>

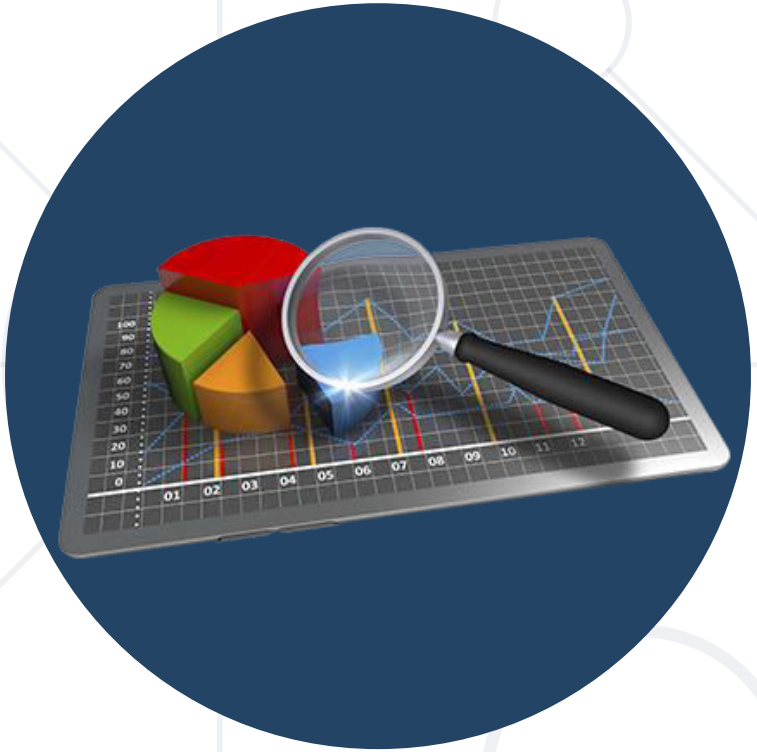
Specialized Collections

DbSet<T> Class

- Generic collection with additional features
- Each **DbSet<T>** corresponds to a single database table
- Inherits from **ICollection<T>**
  - **foreach**-able
  - Supports **LINQ** operations
- Usually several **DbSets** are part of a **DbContext**

DbSet<T> Features

- Each DbSet tracks its own entities through a change tracker
- Has every other feature of an ICollection<T>
  - **Adding**/**Updating** elements
  - **Removing** an entity/a range of entities
  - **Checking** for element **existence**
  - Accessing the **count** of elements



DbContext

DbContext Class

SoftUni Foundation

- Holds several **DbSet<T>**
- Responsible for **populating** the **DbSets**
- Users create a **DbContext**, which **inherits** from **DbContext**
  - Using one DbSet per database table

```
public class SoftUniDbContext : DbContext
{
   public DbSet<Employee> Employees { get; }
   public DbSet<Department> Departments { get; }
   public DbSet<Project> Projects { get; }
   public DbSet<EmployeeProject> EmployeesProjects { get; }
}
```

```
internal class ChangeTracker<T>
    where T: class, new()
{
    private readonly List<T> allEntities;

    private readonly List<T> added;

    private readonly List<T> removed;

    public ChangeTracker(IEnumerable<T> entities)...

    private static List<T> CloneEntities(IEnumerable<T> entities)...

    public IReadOnlyCollection<T> AllEntities => this.allEntities.AsReadOnly();

    public IReadOnlyCollection<T> Added => this.added.AsReadOnly();

    public IReadOnlyCollection<T> Removed => this.removed.AsReadOnly();
```

ChangeTracker<T>

Change Tracking Class

ChangeTracker<T>

- Container for tracking changes
- Holds 3 collections:
  - **All** entities
  - **Added** entities
  - **Removed** entities
- Also can track **modified entities**
  - Through **cloning entities** at initialization

ChangeTracker<T>: Cloning Entities

- In order to check for entity modification, the change tracker **clones** all entities on initialization
- Cloning process:
  - Create **new** blank **instance** of entity
  - Find all **properties**, which are valid SQL types
  - Set blank instance's property **values** to existing entity values

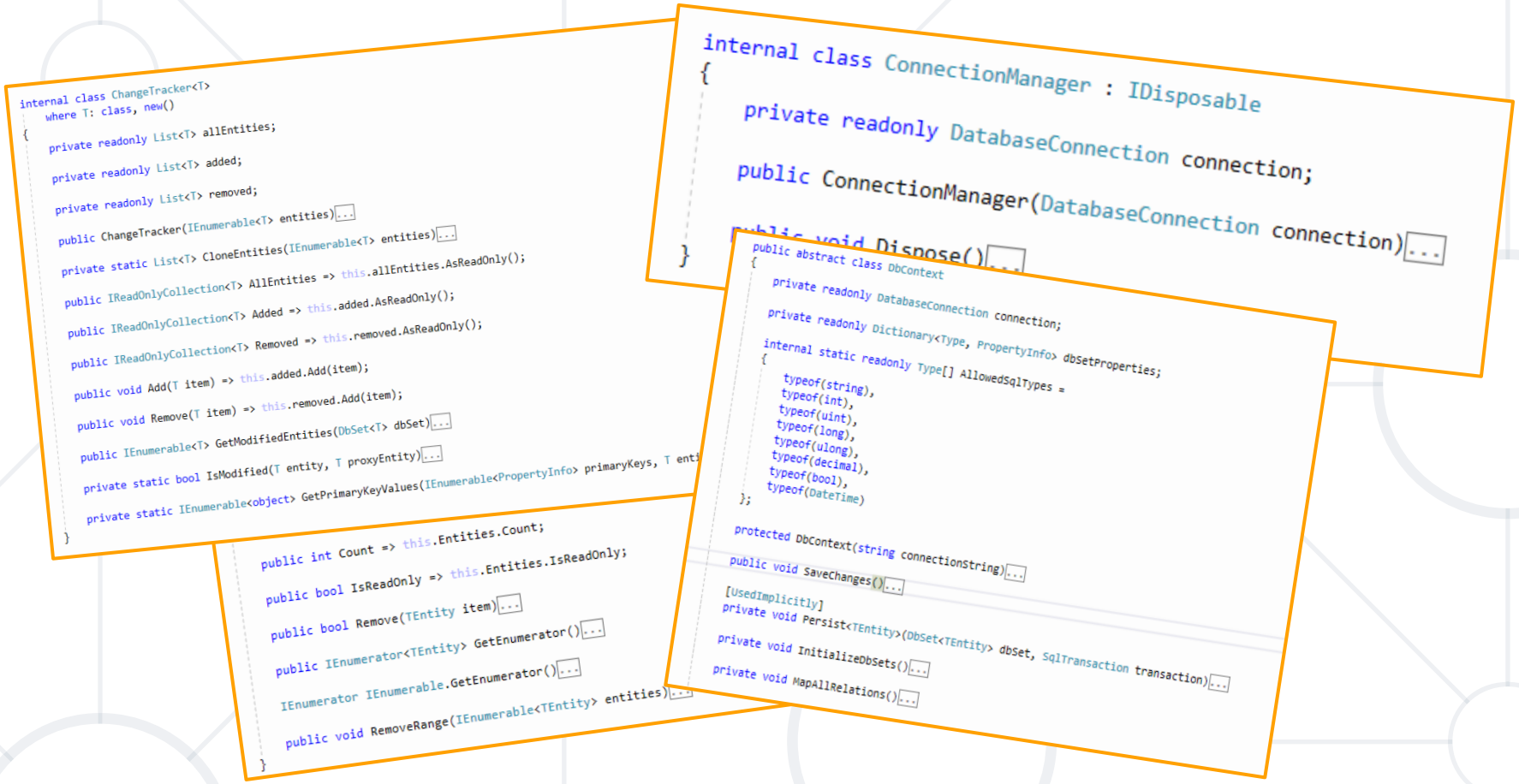
ChangeTracker<T>: Cloning Entities (2)

SoftUni Foundation

- Cloning Process:

```
private static List<T> CloneEntities(IEnumerable<T> entities)
{
   var clonedEntities = new List<T>();
   var propertiesToClone =
// TODO: get properties with SQL types

foreach (var entity in entities) {
     var clonedEntity = Activator.CreateInstance<T>();
     foreach (var property in propertiesToClone) {
       var value = property.GetValue(entity);
       property.SetValue(clonedEntity, value);
     }
     clonedEntities.Add(clonedEntity);
   }
   return clonedEntities;
}
```



Writing an ORM Framework

Live Demo

Reading Data

Querying the DB using MiniORM

Using DbContext Class

SoftUni Foundation

- First create instance of the **DbContext**:

```
var context = new SoftUniDbContext(connectionString);
```

- In the constructor you can pass a database connection string
- DbContext **properties**:
  - All **entity classes** (tables) are listed as **properties**
    - e.g. **DbSet<Employee> Employees { get;}**

Reading Data with LINQ Query

SoftUni Foundation

- Executing **LINQ-to-Entities** query over entity:

```
var context = new
SoftUniDbContext(connectionString)

var employees = context.Employees
   .Where(e => e.JobTitle == "Design Engineer")
   .ToArray();
```

- **Employees** property in the **DbContext**:

```
public class SoftUniDbContext : DbContext
{
   public DbSet<Employee> Employees { get; }
   public DbSet<Project> Projects { get; }
   public DbSet<Department> Departments { get; }
}
```

Reading Data with LINQ Query

SoftUni Foundation

- We can also use **extension methods** for constructing the query

```
var context = new
SoftUniDbContext(connectionString)
var employees = context.Employees
    .Where(c => c.JobTitle == "Design Engineering")
    .Select(c => c.FirstName)
    .ToList();
```

- Find element by **ID**

```
var context = new SoftUniEntities()
var project = context.Projects
      .Single(e => e.Id == 2);
Console.WriteLine(project.Name);
```

CRUD Operations

With MiniORM

Creating New Entities

- To create a new database row use the method **Add(…)** of the corresponding DbSet:

```
var project = new Project()
{
  Name = "Judge System"
};

context.Projects.Add(project);
context.SaveChanges();
```

Create a new Project object

Add the object to the DbSet

Execute SQL statements

Updating Existing Entities

- **DbContext** allows modifying entity properties and persisting them in the database
  - Just load an entity, modify it and call **SaveChanges()**
- The **DbContext** automatically tracks all **changes** made on its entity **objects**

```
Employees employee =
  context.Employees.First();
employees.FirstName = "Alex";
context.SaveChanges();
```

SELECT the first order

Execute an SQL UPDATE

Deleting Existing Data

- Delete is done by **Remove()** on the specified entity collection
- **SaveChanges()** method performs the delete action in the database

```
Employees employee =
  softUniEntities.Employees.First();
softUniEntities.Employees.Remove(employee);
softUniEntities.SaveChanges();
```

Mark the entity for deleting at the next save

Execute the SQL DELETE command

Summary

- **ORM frameworks** map database schema to objects in a programming language
- **LINQ** can be used to query the DB through the **DB context**

```
var employees = context.Employees
   .Where(c => c.JobTitle == "Design
Engineering")
   .Select(c => c.FirstName)
   .ToList();
```

Questions?

SoftUni

Software University

SoftUni Svetlina

SoftUni Creative

SoftUni Digital

SoftUni Foundation

SoftUni Kids

SoftUni Diamond Partners

SoftUni Foundation

XSsoftware

telenor

SoftwareGroup
doing it right

NETPEAK

SmartIT

INDEAVR
Serving the high achievers

INFRAGISTICS®

SUPERHOSTING.BG

SoftUni Organizational Partners

SoftUni Foundation

License

- This course (slides, examples, demos, videos, homework, etc.) is licensed under the "Creative Commons Attribution-NonCommercial-ShareAlike 4.0 International" license

Trainings @ Software University (SoftUni)

SoftUni Foundation

- Software University – High-Quality Education and Employment Opportunities
  - softuni.bg
- Software University Foundation
  - http://softuni.foundation/
- Software University @ Facebook
  - facebook.com/SoftwareUniversity
- Software University Forums
  - forum.softuni.bg