# Exercises: MiniORM

This document defines the **exercise assignments** for the ["Databases Advanced – Entity Framework" course @ Software University.](https://softuni.bg/trainings/1442/databases-advanced-entity-framework-october-2016) Please submit your solutions in [Judge.](https://judge.softuni.bg/)

Following guides of this document you will be able to create your custom ORM with basic functionality (insert, update and retrieve single object or set of objects). Our ORM will have options to work with already created tables in a database or create new tables if such are not present yet.

## Create Attributes

Start with creating new Project called MiniORM. Add folder Attributes to keep our attributes that would be required for our ORM. Create 3 attributes:

* **Id** – has no properties, it will be applied on **fields only**
* **Column** – has property **Name**, it will be applied on **fields only**
* **Entity** – has property **TableName**, it will be applied on **classes only**

Follow the good practices in naming custom attributes.

## Create Entities

In our project create folder called Entities where we could keep every one of our entities. Now let’s **create class User** with fields and properties (**id, username, password, age, registrationDate**). Add appropriate **attributes** to our newly created class and its fields. Create **constructor** that **sets all properties except Id**. Order of the parameters in the constructor must be **the same as the sequence of columns in the table in the database**

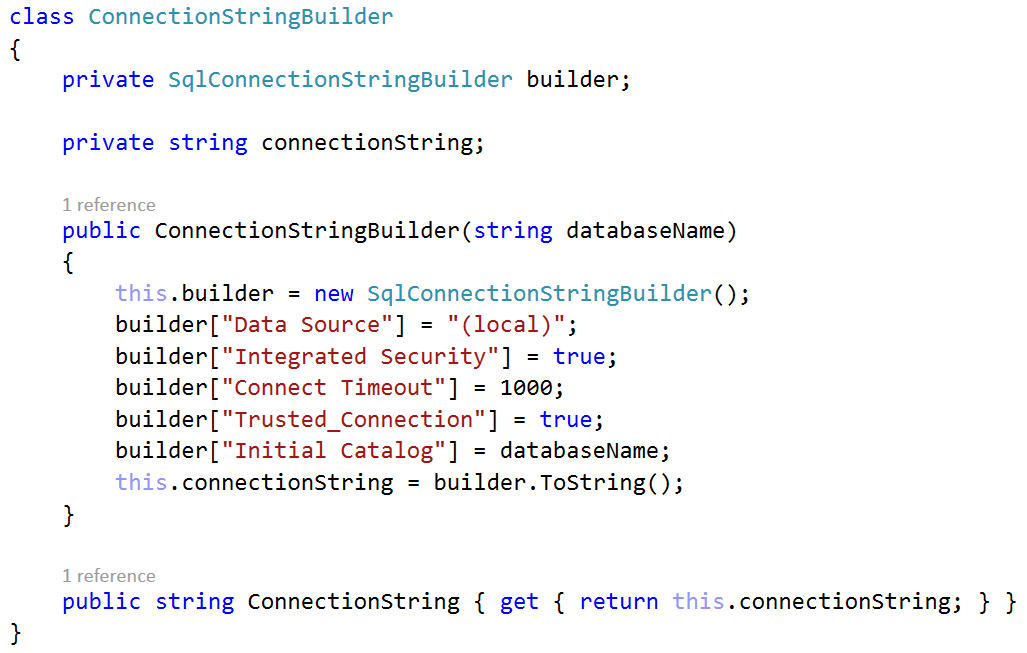


## Create Database Connection String Builder

Now we can **create class that** **generate connection string by given name of a database**. For the purpose of that exercise we assume the database is on our localhost and we would use our windows credentials to log in, but that class can be extended to create connection strings for remote databases with username and password required to log in. We can use the SqlStringConnectionBuilderclass to help us with the creation of the string. For the connection we could set default values of:

* **Data source** – (local)
* **Integrated Security** – true
* **Connect Timeout** – 1000
* **Trusted Connection** – true

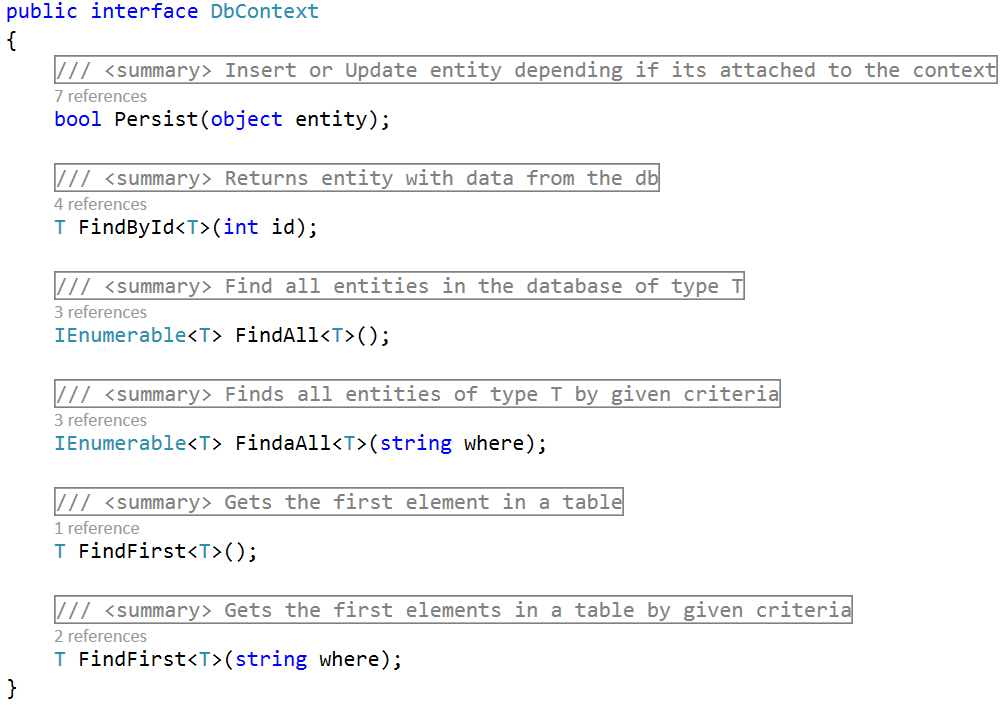
The class should generate connection string that we would use to connect to the database.



## Create Database Context

It’s time to create interface that would define the operations we can perform with the database. Name your interface **DbContext** and defined the following methods in it.

* **bool Persist(object entity)** – it will insert or update entity depending if it is attached to the context
* **T FindById<T> (int id)** – return entity object of type T by given id
* **IEnumerable<T> FindAll<T> ()** – returns collection of all entity objects of type T
* **IEnumerable<T> FindaAll<T> (string where)** – returns collection of all entity objects of type T matching the criteria given in “where”
* **T FindFirst<T> ()** – returns the first entity object of type T
* **T FindFirst<T> (string where)** – returns the first entity object of type T matching the criteria given in “where”



## Create Entity Manager

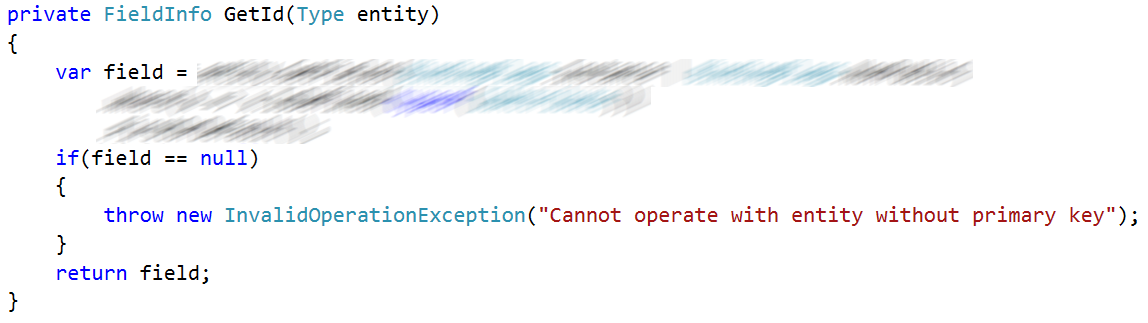
Enough with the preparation it’s time to write the core of our Mini ORM. That would be **EntityManager** class that would be responsible for inserting, updating and retrieving entity objects. That class **would implement methods of the DbContext interface**. That class would require a **SqlConnection** object that would be initialized with a given connection string. Also for initial set up the Entity Manager should know what to do in case of table of the database is missing (should create a new table or throw exception).

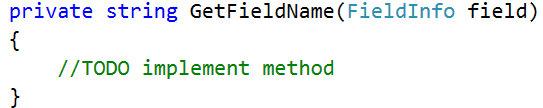
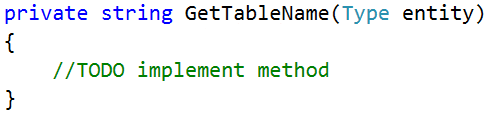


## Helper Methods

It’s time for some reflection. We would create 3 methods that would help us

* **FieldInfo GetId (Type entity)** – Get the field with Id attribute of the given entity. If there is no field with Id attribute throw exception.
* **string GetTableName (Type entity)** – Returns the value of the **TableName** property of Entity attribute or if it’s not set returns the name of the entity.
* **string GetFieldName (FieldInfo field)** – Returns the value of the Name property of Column attribute or if it’s not set returns the name of the field.

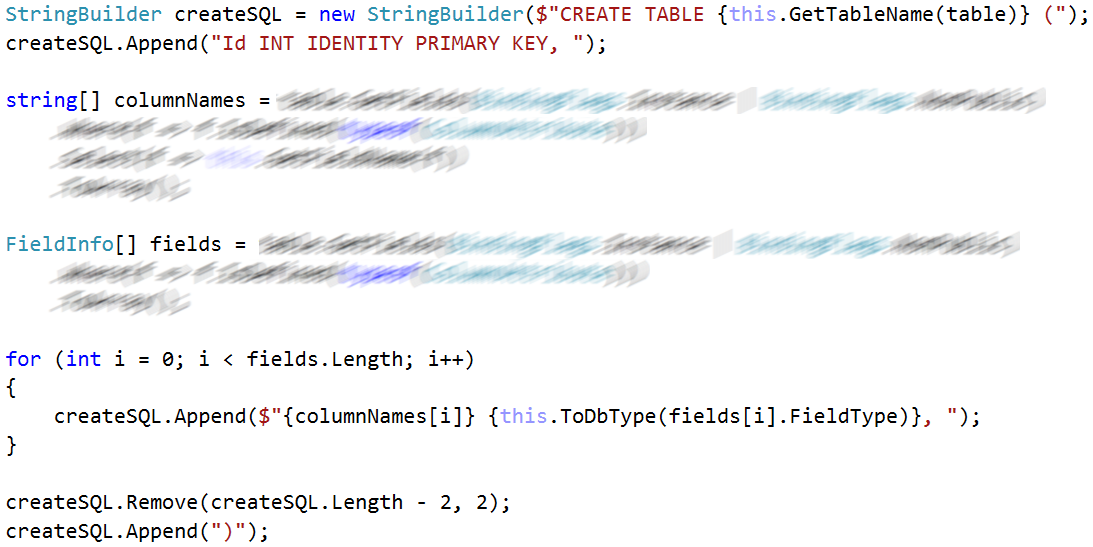




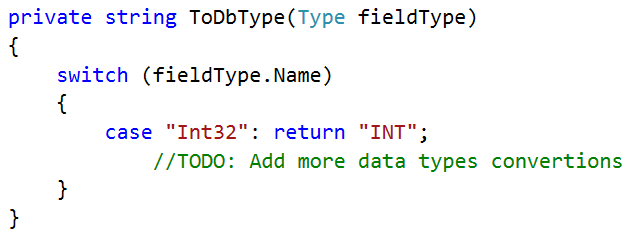
## Create Table

In case we set our **Entity Manager to be code first** (create tables from classes) we should have method that create that table for us. We could just create a method that we can easily execute if that option is set to true. The method should just **compose CREATE TABLE statement and execute it**.

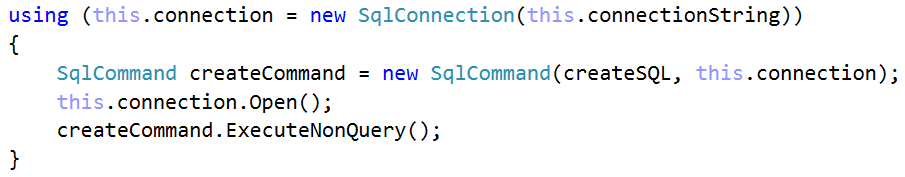
To prepare CREATE TABLE statement we need to know the **name of the table** and its **column names and their types**. (psst using reflection and some of the helper methods in the previous task will be handy). The **Id** **column** should be **identity** and **primary key**. All other columns should have the appropriate database type depending on the field type (for example. String -> VARCHAR (100), Int32 -> INT…)



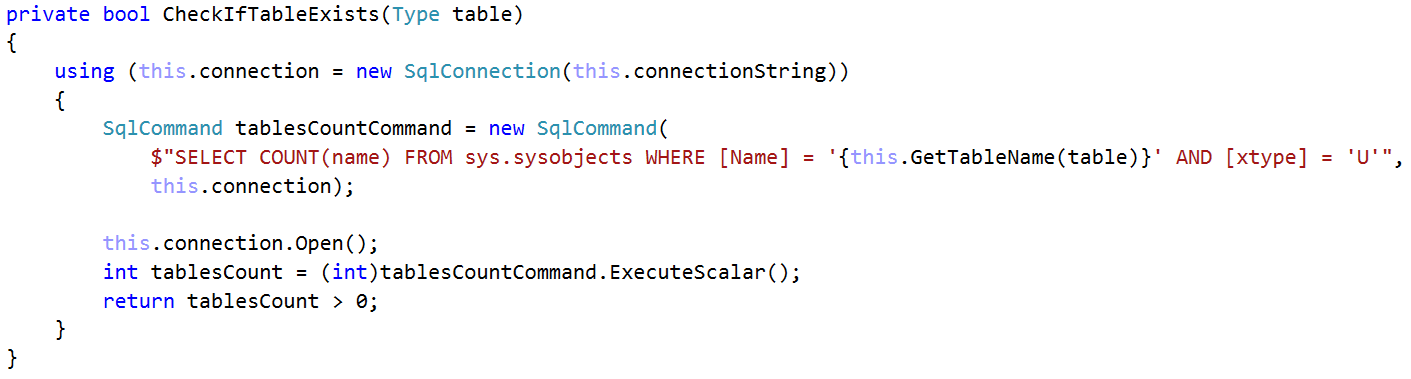
**ToDbType** method returns the most appropriate database type depending on the type of the field.



Now we just can execute the command and our table will be created.

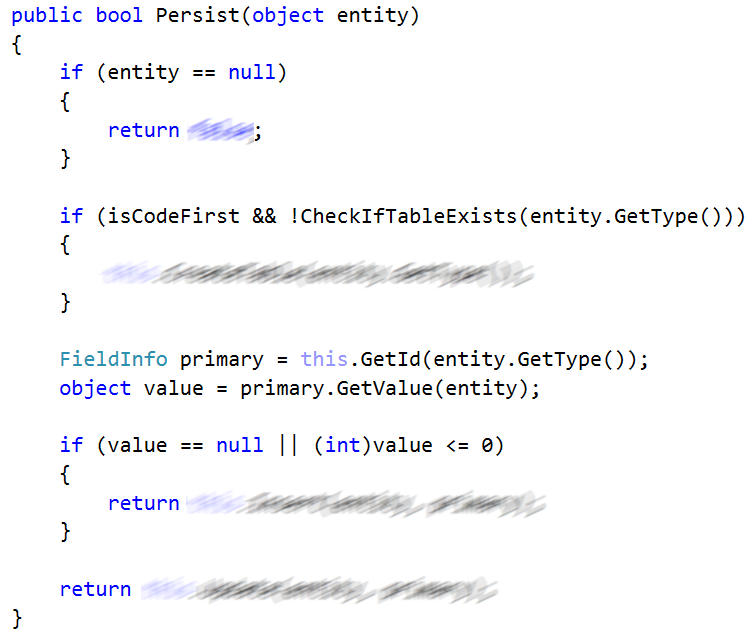


But before creating a table we should **check if the table with the same name already exists** in the database. If such table exists we should use it instead of creating new table. So a method that do that for us would be helpful.



## Persist Object in the Database

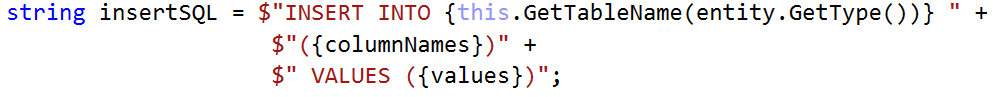
The logic behind the persist method is pretty simple. If the **isCodeFirst** option is set to **true** and the **table is not present** in the database -> **create the table**. Then if the given **object** to be persisted **is not contained** in the database -> **add it**, otherwise **update its properties with the new values**. The method returns whether the object was **successfully persisted** in the database or not.

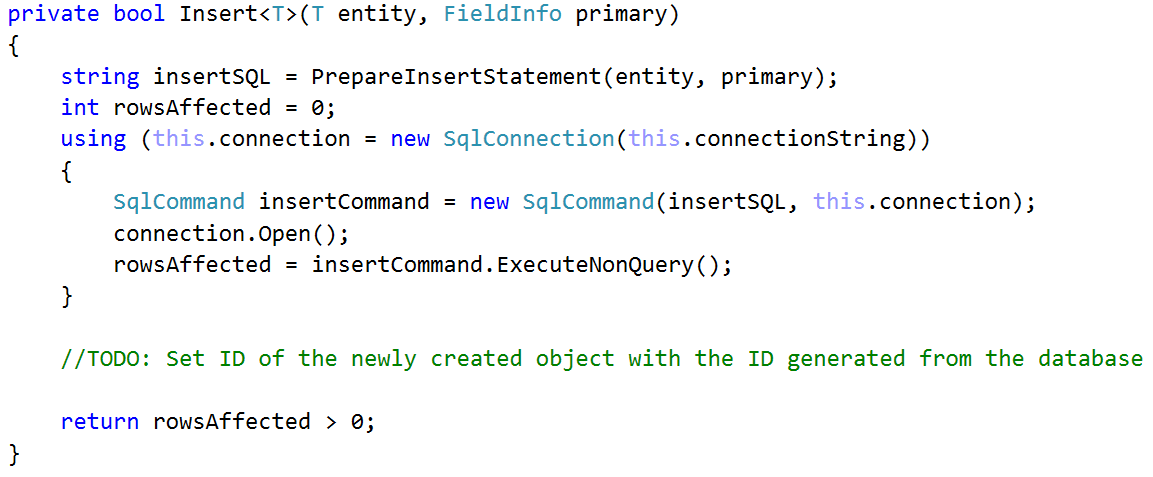


So far we need to implement 2 more methods **Insert<T>(T entity, FieldInfo primary)** and **Update<T>(T entity, primary)**. Both methods would prepare query statements and execute them.

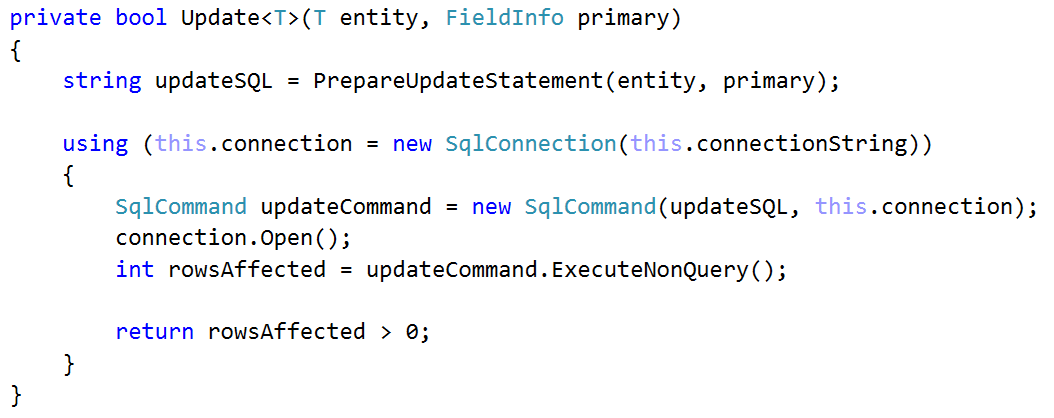
The difference between them is when you insert new entity you should **set its Id**. The Id is generated from the table in the database. Both methods return whether the entity was successfully persisted.

Here are some tips for the Insert method:





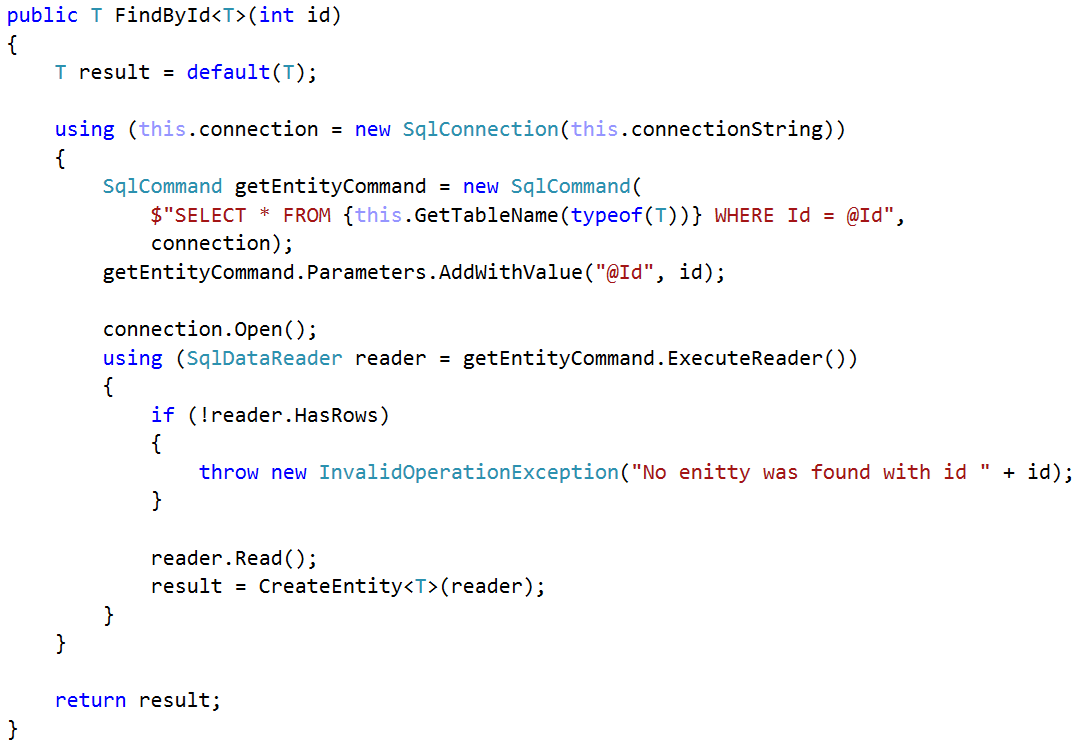
And some tips for the update method



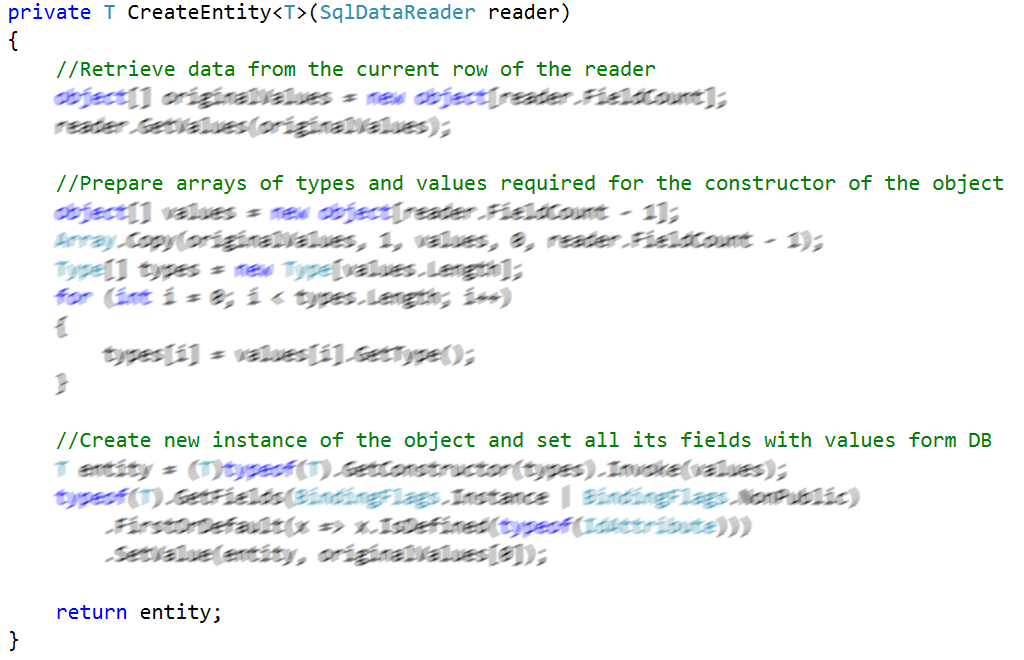
## Fetching Results

Finally, when we persisted our entities (objects) in the database let ‘s implement functionality to **get them out of the database and persist them in the operating memory**. We would implement just several methods to get objects from the database. That would be **all Find methods from the DbContext** (check Problem 4. for more information about each one of them).

Here is tip of how to implement **FindById<T>(int id)** the other ones are similar and they would be on you ☺.

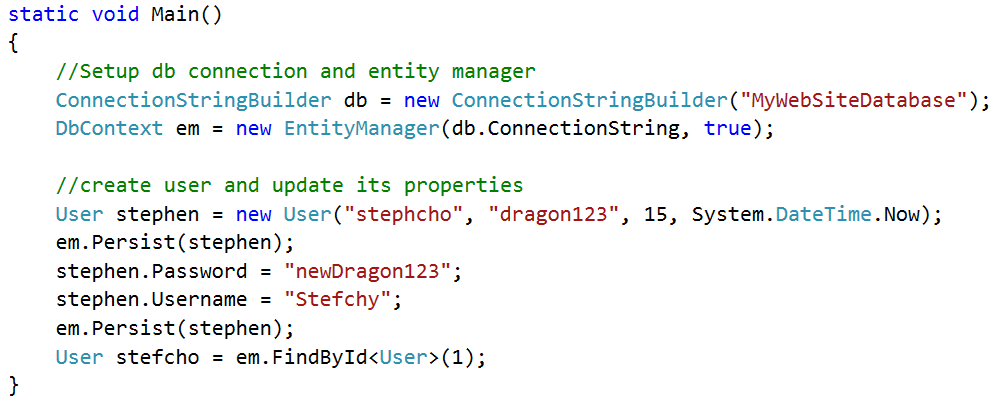


Here you can see that we used some new method **CreateEntity**. That method receives **SqlReader** object, **retrieve information from the current row** of the reader and **create new instance of the object** of type T.



## Test Framework

If you came to this point you are done with building our MiniORM. Now let’s test it to make sure it works as expected. Create several users and persist them in the database. Then update some of the properties of the users (e.g., change password or increase age or some other change). Remember you need to use the persist method to commit changes of the object to the database. Make sure the data is always updated in the database. Here is some example of usage:



## Fetch Users

Insert several users in the database and **print the usernames and passwords** of those who are **registered after 2010** year and are **at least 18 years old**.

## Add New Entity

Add new class **Book (id, title, author, publishedOn, language, isHardCovered)**. Add several books to the database. Write a program that check all the books whether their **title is over N symbols long** (where N is a number given as an input) and **have hard cover**. If that is true, then **trim the exceeding length** and make it exactly N symbols long. Make sure they are **properly persisted in the database** after those changes. Finally **print** on the console the **number of books with title exactly N symbols long**.

### Example

|  |  |
| --- | --- |
| **Input** | **Output** |
| 30 | 9 books now have title with length of 30 |
| 100 | 0 books now have title with length of 100 |

## Update Entity

Add new column **Rating** (possible values in range **from 0 to 10**) to the book entity. Find **latest 3 books with highest rating** and **print their titles, authors and rating**. Sort them **by rating descending** then **by title ascending**. Use the following format to print the output:

{Book title} ({Author}) – {rating}/10

### Example

|  |
| --- |
| **Output** |
| Around the World in Eighty Days (Jules Verne) – 9.7/10  The Foundation (Isaac Asimov) – 9.7/10  Harry Potter and the Philosopher's Stone (JK Rowling) – 9.6/10 |

## Update Records

Write a program that receives as an **input year** and then **changes all book titles to be uppercase if they are released after that given year** (exclusive) and has hard cover. **Print their count** on the console followed by **their titles ordered lexicographically**.

### Example

|  |  |
| --- | --- |
| **Input** | **Output** |
| 2000 | Books released after 2000 year: 4  HARRY POTTER 4  HARRY POTTER 5  HARRY POTTER 7  HARRY POTTER AND HALF-BLOOD PRINCE |

## Delete Records

**Extend your ORM** to be possible to **delete objects**. Then find **all books that has rating below 2** and **delete them** from the database. Print on the console the **number of books that has been deleted**.

### Example

|  |
| --- |
| **Output** |
| 3 books has been deleted from the database |

## Delete Inactive Users

Add two columns to the **User** entity:

* **LastLoginTime** – that will keep the last time our user has been logged in the system
* **IsActive** – that will keep whether the last login time has been for more than a year.

Populate the database with sample data of users.

Write a program that **get as an input username** then check in the database whether the user with **that username is active or not**. If it is active **print his last login time relatively** to the moment of executing the program (check table below for examples). If it is not active **prompt the user to confirm if that user should be deleted**. If the user confirms then **delete that user from the database** otherwise **do nothing**.

|  |  |
| --- | --- |
| **Last time user was logged in** | **Relative time** |
| < 1 second | less than a second |
| < 1 minute | less than a minute |
| < 1 hour (x minutes) | {x} minutes ago |
| < 1 day (x hours) | {x} hours ago |
| < 1 month (x days) | {x} days ago |
| < 1 year (x months) | {x} months ago |
| > 1 year | more than a year |

### Example

|  |  |
| --- | --- |
| **Input** | **Output** |
| dragonZ999 | User dragonZ999 was last online 3 hours ago |
| bobara2016 | User bobara2016 was last online less than a minute |
| pesh123  yes | User pesh123 was last online more than a year.  Would you like to delete that user? (yes/no)  User pesh123 was successfully deleted from the database |
| bigCatTom  no | User bigCatTom was last online more than a year  Would you like to delete that user? (yes/no)  User bigCatTom was not deleted from the database |