# Exercises: Entity Framework Code First + OOP Intro

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)

# OOP Intro

## Define a class Person

Define a class **Person** with **public** properties for **name** and **age**.

### Bonus

Try to create a few objects of type Person:

|  |  |
| --- | --- |
| **Name** | **Age** |
| Pesho | 20 |
| Gosho | 18 |
| Stamat | 43 |

Use both the inline initialization and the default constructor. Your fields should be **public**, otherwise you won’t see them.

## Create Person Constructors

Add 3 constructors to the **Person** class from the last task, use constructor chaining to reuse code:

1. The first should take no arguments and produce a person with name “**No name**” and age = **1**.
2. The second should accept only an integer number for the age and produce a person with name “**No name**” and age equal to the passed parameter.
3. The third one should accept a string for the name and an integer for the age and should produce a person with the given name and age.
4. The fourth one should take both parameters.

Your task is to create a command which parses the input parameters (if any), split by “,” and based on them to instantiate new object of type Person.

### Examples

|  |  |
| --- | --- |
| **Input** | **Output** |
| Pesho,20 | Pesho 20 |
| Gosho | Gosho 1 |
| 43 | No name 43 |
|  | No name 1 |

## Oldest Family Member

Use the same class **Person** with fields **name** and **age**. Create a class **Family**. The class should have **list of people**, method for adding members (void AddMember(Person member)) and a method returning the oldest family member (Person GetOldestMember()). Write a program that reads name and age for **N** people and **add them to the family**. Then **print** the **name** and **age** of the oldest member.

### Examples

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Input** | **Output** |  | **Input** | **Output** |
| 3  Pesho 3  Gosho 4  Annie 5 | Annie 5 |  | 5  Steve 10  Christopher 15  Annie 4  Ivan 35  Maria 34 | Ivan 35 |

## Students

Define class **Student**. Add **string field** for a student’s **name** that you are going to receive as a console input. Then add a **static Integer field** to **keep track of how many students’ instances are created**. Initialize the static field with **0 (zero)** and **increment in the constructor**. When you receive command **“End”** stop reading more students names and print their total count on the console.

### Examples

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Input** | **Output** |  | **Input** | **Output** |
| Atanas  Martin  End | 2 |  | Minka  End | 1 |

## Planck Constant

Create class **Calculation**. Define static constant with value **6.62606896e-34** (Planck constant) and **3.14159** (Pi). Add **static method** that returns reduced Planck constant by the formula:

{Planck constant} / (2 \* {Pi constant})

Print the result of the method on a **single line on the console**. **Do not format** in any way the **result.**

## Math Utilities

Define **MathUtil** class that supports **basic** mathematical operations:

* **Sum <first number> <second number>**
* **Subtract <first number> <second number>**
* **Multiply <first number> <second number>**
* **Divide <dividend> <divisor>**
* **Percentage <total number> <percent of that number>**

Use **static methods** and make sure that the application will work with **floating point numbers**.

Read from the console until you receive command **“End”.** Results must be formatted with **2 digits after the floating point.**

### Examples

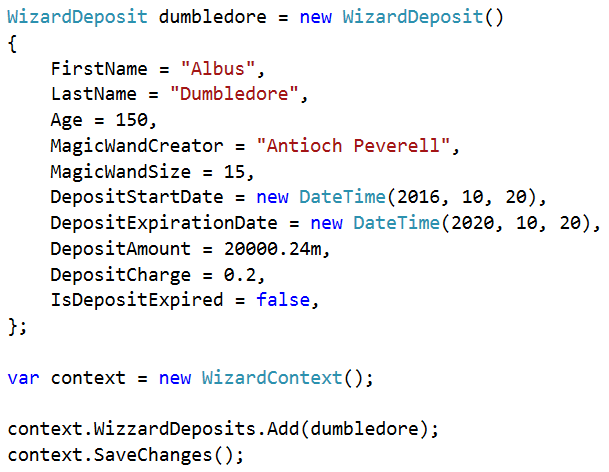
|  |  |
| --- | --- |
| **Input** | **Output** |
| Sum 5 5  Multiply 5.5 11  Percentage 1500 99  Divide 12.24 2  Subtract 10.6 0.6  End | 10.00  60.50  1485.00  6.12  10.00 |

# Code First

## Gringotts Database

Your task is to create table **WizardDeposits** using the Code First approach. The table should contain the following fields:

* **Id** – Primary Key (number in range [1, 231-1].
* **FirstName** – Text field with max length of 50 symbols.
* **LastName** - Text field with max length of 60 symbols. Required
* **Notes** – Text field with max length of 1000 symbols
* **Age** – Non-negative number. Required
* **MagicWandCreator** – Text field with max length of 100 symbols
* **MagicWandSize** – Number in range [1, 215-1]
* **DepositGroup** - Text field with max length of 20 symbols
* **DepositStartDate** – Date and time field
* **DepositAmount** – Floating point number field
* **DepositInterest** - Floating point number field
* **DepositCharge** - Floating point number field
* **DepositExpirationDate** – Date and time field
* **IsDepositExpired** – Boolean field

Add several records to the database using Entity Framework. Use the following example of wizard deposit creation and addition to the database.

## Create User

Your task is to create table **Users** using the Code First approach. The table should contain the following fields:

* **Id** – Primary Key (number in range [1, 231-1])
* **Username** – Text with length between 4 and 30 symbols. Required.
* **Password** – Required field. Text with length between 6 and 50 symbols. Should contain at least:
  + 1 lowercase letter
  + 1 uppercase letter
  + 1 digit
  + 1 special symbol (!, @, #, $, %, ^, &, \*, (, ), \_, +, <, >, ?)
* **E-mail** – Required field. Text that is considered to be in format **<user>@<host>** where:
  + **<user>** is a sequence of letters and digits, where '**.**', '**-**' and '**\_**' can appear between them (they cannot appear at the beginning or at the end of the sequence).
  + **<host>** is a sequence of at least two words, separated by dots '**.**' (dots cannot appear at the beginning or at the end of the sequence)
* **ProfilePicture** – Image file with size maximum of 1MB
* **RegisteredOn** – Date and time of user registration
* **LastTimeLoggedIn** – Date and time of the last time the user logged in
* **Age** – number in range [1, 120]
* **IsDeleted** – Shows whether the user is deleted or not

Seed some users in the database. Test if validation of the fields works as expected.

## Hospital Database

Congrats! You were hired as a Junior Database App Developer. But before starting to work you were required to provide some documents such as fit note from your GP. So, you go to him to get it. When you tell him, what do you need and what kind of job you are about to start. He told you that he was just looking for someone to make a software to help him managing and keeping data about his patients. He offered you to give you the fit note for free if you help him. You decided that’s a great opportunity to save 20 leva and go out tonight with friends and also you would expand your portfolio with 1 project.

Your task is to design a database using the Code First approach. The GP needs to keep information about his patients. Each patient has first name, last name, address, email, date of birth, picture, information whether he has medical insurance or not and should keep history about all his visitations, diagnoses and prescribed medicaments. Each visitation has date and comments. Each diagnose has name and comments for it. Each medicament has **name**. Make sure all data is validated before inserting in the database.

### Bonus Task

Make console based user interface so the doctor can use easily the database.

### \*\*Bonus Task

The console based user interface is a good start point but almost nobody use them nowadays (except BDZ Passenger Services). So, make a nice good looking graphical user interface for the program.

## Hospital Database Modification

Your GP bragged around in the hospital about the cool software you made for him. And now the hospital administration wants to modify your program so they can use it too. They want to store information about the **doctors** (**name** and **specialty**). Each doctor can perform **many** **visitations** and in each **visitation**, can be **performed** **by** only **one** **doctor**. Make the necessary changes in the database to satisfy the new needs of the hospital administration. When migrating to the new database schema make sure no data is lost. If you made some user interface (graphical or not), make changes in it be more adequate for the changes.

### \*\*Bonus Task

Make authentication system for doctors. Each doctor should be able to log in with his email and password. When the doctor is logged in he can see only information related about himself (his visitations, the patients he examined, etc.).

## Get Users by Email Provider

Insert some sample [data](https://softuni.bg/downloads/svn/DB-Fundamentals/DB-Advanced-EntityFramework/Feb-2017/03.%20DB-Advanced-EntityFramework-EntityFramework-Code-First+OOP-Intro/Users-Sample-Data.sql) in **Users** database then write program that print all usernames and emails of users by given email provider.

### Example

|  |  |
| --- | --- |
| **Input** | **Output** |
| d.edu | Hu Sed@Sedid.edu  Roary Praesent.luctus@velfaucibusid.edu |
| t.org | Lee odio@lectusNullamsuscipit.org  Denton nonummy@sit.org |

## Remove Inactive Users

Write a program that set IsDeleted field to true for all users that has not been logged in after given date. Print the number of user that has been set as deleted. Then delete all users that have been marked for removal.

### Example

|  |  |
| --- | --- |
| **Input** | **Output** |
| 01 May 2013 | 3 users have been deleted |
| 10 Apr 2014 | 21 user has been deleted |
| 12 Dec 2013 | No users have been deleted |

\*Note that the example is valid only in this order.