# Lab: Introduction to .NET Core and EF Core

This document defines the **in-class exercises** assignments for the ["C# Web Development Basics" course @ Software University](https://softuni.bg/courses/csharp-web-development-basics). You can submit your code in the course page.

1. **School Competition**

You are given a sequence of students with their points from competitions in different categories.

The input comes in the following format: “{student name} {category} {points}”.

For each student print the **sum of the points** and a list of **unique categories** they have participated.

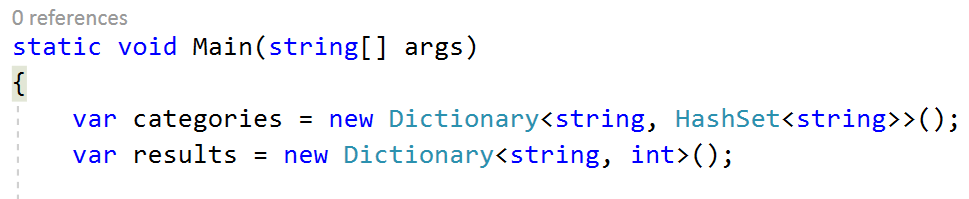
Order students by their **points** in **descending** order. If two students have **equal points** order them by **name** in **ascending** order. Order the categories by **name** in **ascending** order.

### Examples

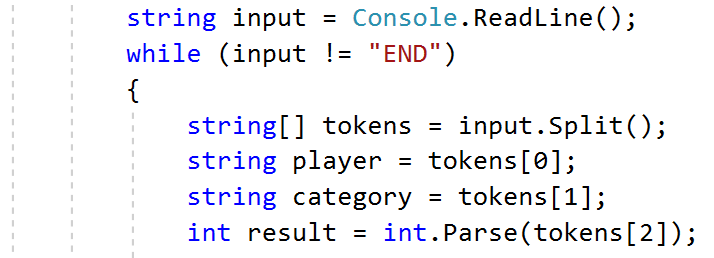
|  |  |
| --- | --- |
| **Input** | **Output** |
| Ani Math 80  Gosho Math 10  Pesho History 100  Gosho Geography 90  END | Gosho: 100 [Geography, Math]  Pesho: 100 [History]  Ani: 80 [Math] |
| Ivan Literature 50  Martin History 100  Ivan Math 60  Ivan Math 78  Petya English 120  END | Ivan: 188 [Literature, Math]  Petya: 120 [English]  Martin: 100 [History] |

### Hints

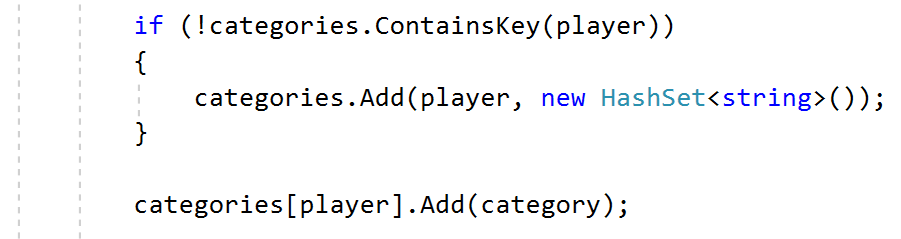
Create collections to store categories and results



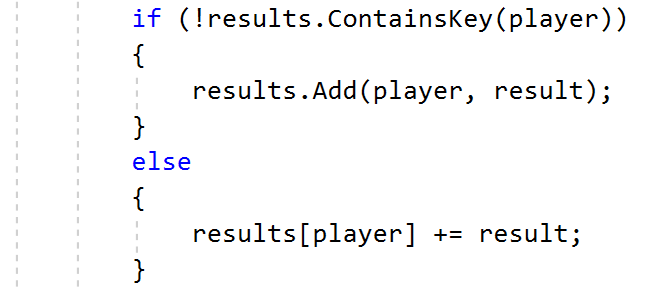
Read and parse the input in a **while loop**



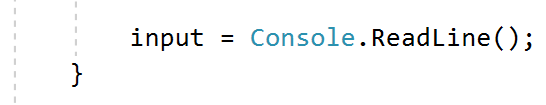
Fill category into categories collection:



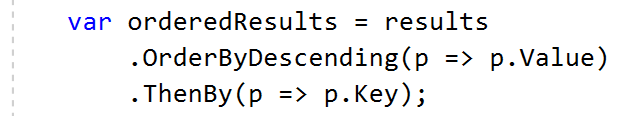
Add result to results collection:



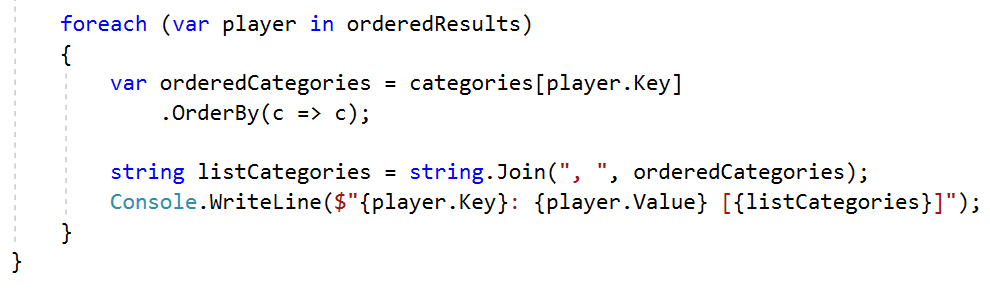
Read next input:



After reading the input order the results by value descending and by student name ascending:



Order categories in ascending order by name for each student and print result:



## One-to-Many Relation

Create a database and implement one-to-many relation with the following entities:

##### \

* Id–int
* Name – text with max length 50 (**required**)

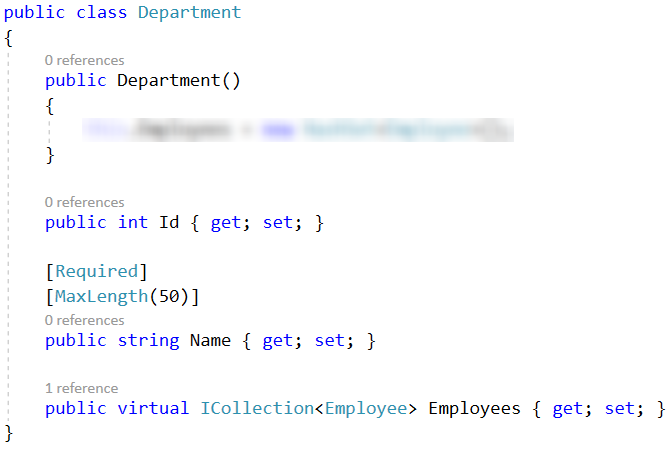
##### Department

* Id – int
* Name – text with max length 50 (**required**)

Each **department** has a **collection** of Employee objects.

### Hints

* Create class Department



* Create the Employee class and add a **navigation property** for the **department**.
* To create the database use migrations or context.Database.EnsureCreated().

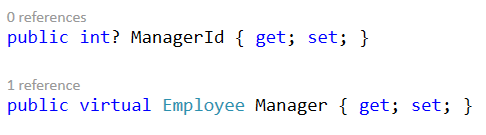
## Self-Referenced Table

Use the database from the previous problem.

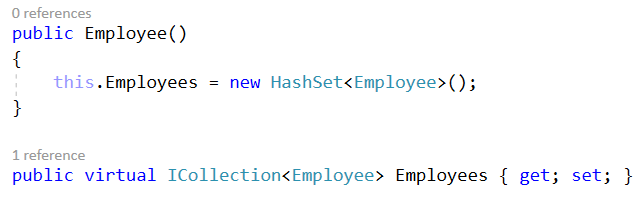
Each employee can have a **manager**. The **manager** is an object of type Employee and is **optional**.

### Hints

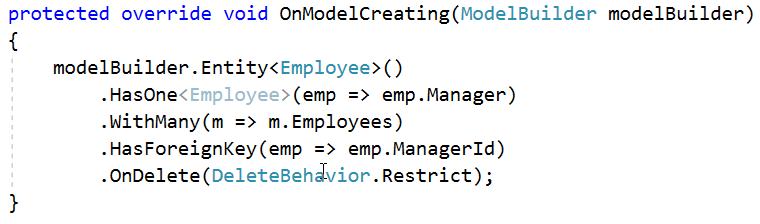
* Add optional virtual property Manager to the Employee class



* Add collection of employees and initialize it in the constructor



* You can use migrations to alter the database.
* Create one-to-many relation in the DbContext class



## Many-to-Many Relation

Create database and implement one-to-many relation with the following entities

##### Student

* Id–integer
* Name – text with max length 50 (**required**)

##### Course

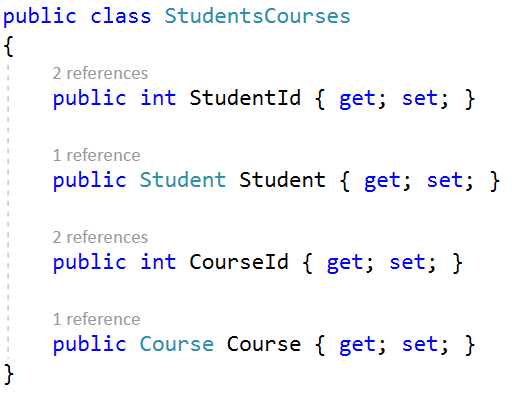
* Id – integer
* Name – text with max length 50 (**required**)

Each Student can participate in many courses and each Course can have many objects of type Student.

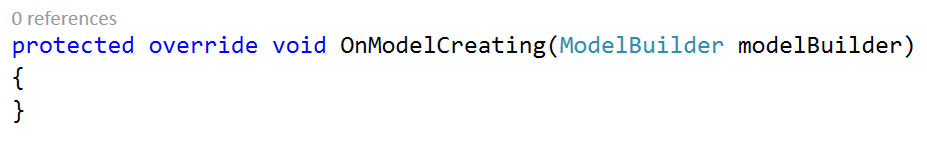
Implement many-to-many relation between the tables Student and Course.

### Hints

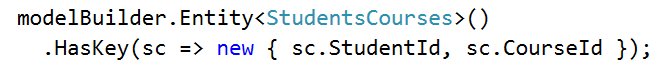
* Create join model StudentsCourses.



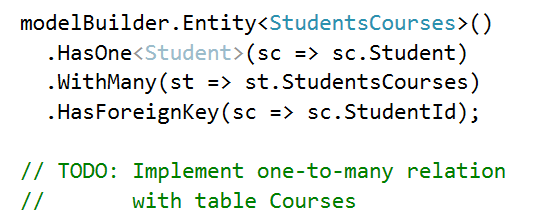
* Override the method OnModelCreating in your DbContext class to implement the many-to-many relation between tables Students and Courses.



* Use the modelBuilder object to set the primary key for the table StudentsCourses. This primary is **composite** and consists of two fields – StudentId and CourseId.

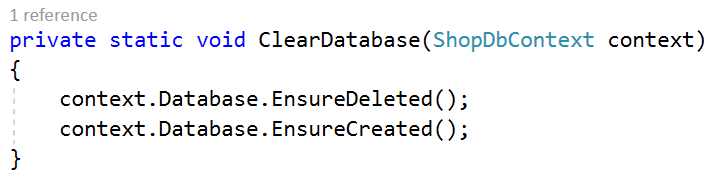


* Implement one-to-many relation between table StudentsCourses and each of the tables Students and Courses.



To test your solution for the rest of the problems locally you will need to clear database and reset identities before running each test.

You can use the following method to delete the database and create it again when starting you project.



## Shop Hierarchy

Create class hierarchy for a small online shop.

The shop hierarchy includes classes Salesman**,** Customer.

##### Customer

* **Id** – integer
* **Name** – text with max length 50 (**required**)

##### Salesman

* **Id** – integer
* **Name** – text with max length 50 (**required**)
* **Customers** – A collection of Customer

Fill table Salesmen with data. You will receive a sequence of salesmen names, separated by ";".

After filling the data, you will receive a sequence of customers to register.

Each line comes in the format "register-{customer name};{salesman id}".

On this command create a customer and add them to the database. Add the created customer to the salesman with the given id. Keep reading lines until you receive a command "END".

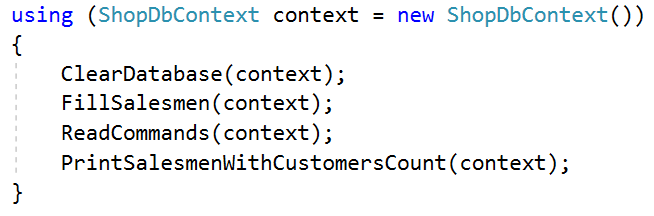
After reading all the registrations print all the salesmen with the count of their customers. Order the results by **number of customers** in **descending** order. If two salesmen have **equal number of customers** order them by **name** in **ascending** order. Format the result as follows: "{salesman name} – {number of customers} customers".

### Examples

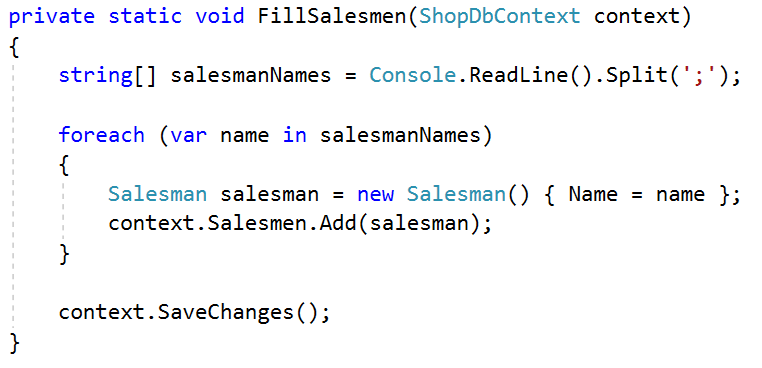
|  |  |  |  |
| --- | --- | --- | --- |
| **Input** | **Output** | **Input** | **Output** |
| Ivan;Pesho;Andrey  register-Ani;1  register-Pavel;1  register-Gosho;2  register-Maria;3  END | Ivan - 2 customers  Andrey - 1 customers  Pesho - 1 customers | Tenko;Valka;Petkan  register-Ani;1  register-Maira;2  register-Gosho;2  register-Stoyan;3  register-Petya;3  register-Andrey;3  END | Petkan - 3 customers  Valka - 2 customers  Tenko - 1 customers |

### Hints

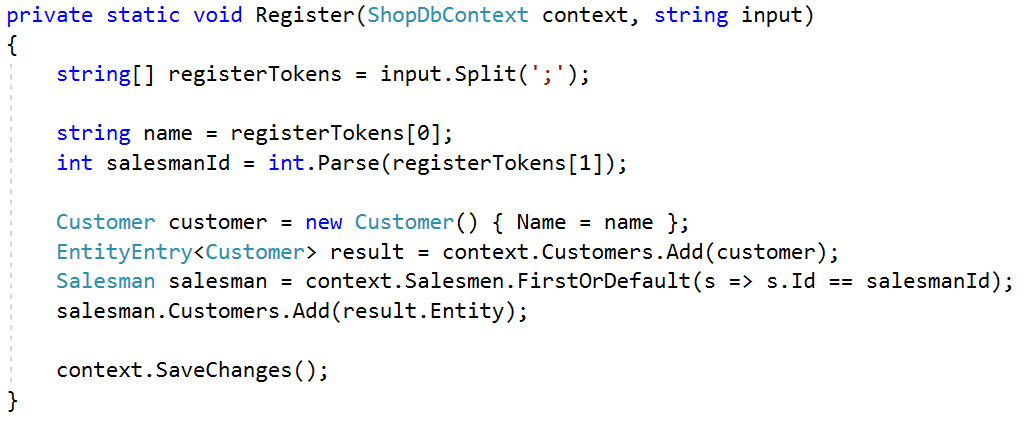
Implement the following methods:



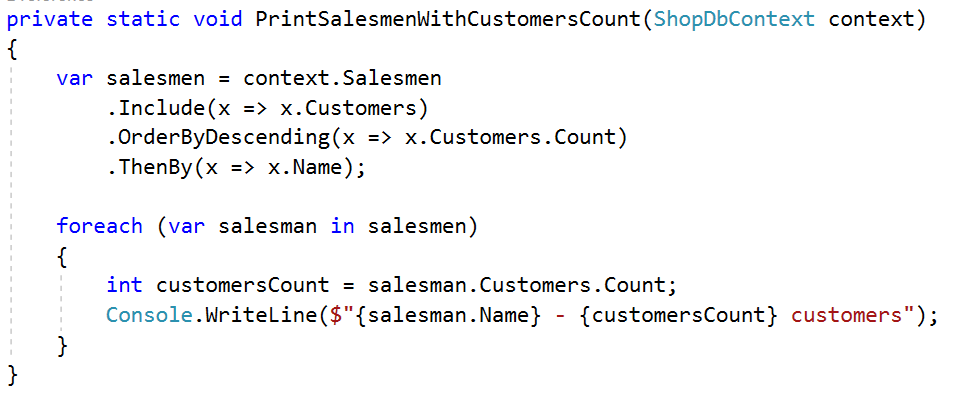
Fill salesmen in the database:



Register customers:



Query the database and print the results:



## Shop Hierarchy – Extended

Extend the shop database from the previous problem. Keep the functionality for filling data in the database as it will be needed for this problem.

Add the following entities to the previously created models:

##### Order

* **Id** – integer

##### Review

* **Id** – integer

**Each customer can make many orders and leave many reviews.**

Fill the **Salesmen** table with data using the logic from the previous problem.

After filling the data, you will receive a sequence of commands:

* "register-{customer name}": Use the registration logic from the previous problem.
* "order-{customer id}": Customer with given id makes an order.
* "review-{customer id}": Customer with given id leaves a review.

Keep reading lines from the console until you receive a command "END".

After reading all the commands print **each customer** with the **count of their orders** and the **count of their reviews** in the following format:

"{customer name}"

"Orders: {number of orders}"

"Reviews: {number of reviews} ".

Order the results by the **number of orders** in **descending** order. If two customers have **equal number of orders**, order them by **reviews count** in **descending** order.

### Examples

|  |  |  |  |
| --- | --- | --- | --- |
| **Input** | **Output** | **Input** | **Output** |
| Ivan;Pesho  register-Ani;1  register-Maria;2  order-1  review-1  order-2  order-2  review-2  END | Maria  Orders: 2  Reviews: 1  Ani  Orders: 1  Reviews: 1 | Martin;Stefo;Siyka  register-Mina;1  register-Velko;1  register-Minko;2  order-1  order-1  review-1  order-2  review-2  order-3  order-3  order-3  review-3  END | Minko  Orders: 3  Reviews: 1  Mina  Orders: 2  Reviews: 1  Velko  Orders: 1  Reviews: 1 |

### Hints

Add a **collection** of orders and a **collection** of reviews to the Customer model.

Use several Include() methods to get the orders and the reviews of each customer.

## Shop Hierarchy – Complex Query

Use the logic for **reading** data and the **models** from the previous problem.

Add the entity item to the Shop entities.

##### Item

* **Id** – integer
* **Name** – text with max length 50 (**required**)
* **Price** – floating point number

Each **order** can have **many items** and each **item** can be part of **many orders**.

Each **item** can have **many reviews**, but each **review** is made for a **single item**.

You will receive **several** items in the format "{item name};{item price}" each on a new line. Add each item to the database. Read lines with items until receiving a command "END".

After filling the items, you will receive several commands "register", "review" and "order". The logic about "register" is the same as in the previous problems.

When you receive a command "order" you will get information about the items that are included in that order.

* "order-{customer id};{item id};{item id}...{item id}": When a customer makes an order, the order includes several items.

When you receive a command "review" you will get information about the item connected to that review.

* "review-{customer id};{item id}": A customer leaves a review for a specific item.

Keep reading commands until you receive a command "END".

After reading all the data, you will receive an **id** of a customer. Print the **number of items** in each of their **orders** in the format "**order {order id}: {items count} items**" each on a **new line**. Order the results by **order id** in **ascending** order. After printing the orders, print the **count of the reviews** that this customer has left in the format "reviews: {reviews count}".

### Examples

|  |  |  |  |
| --- | --- | --- | --- |
| **Input** | **Output** | **Input** | **Output** |
| Ivan;Pesho  Televizor;100  Golqm televizor;200  END  register-Ani;1  order-1;1;2  order-1;1  review-1;2  END  1 | order 1: 2 items  order 2: 1 items  reviews: 1 | Ancho;Tenko  Topka;20  Banichka;10  Chasha;18  END  register-Asen;1  register-Ivan;1  register-Cecko;2  order-1;1;2;3  order-1;2  review-1;2  order-2;3;1  review-1;2  order-3;3;1  order-3;1  END  1 | order 1: 3 items  order 2: 1 items  reviews: 2 |

### Hints

Implement **many-to-many** relationship between Item and Order entities.

Implement **one-to-many** relationship between Item and Review entities.

Use Include() and ThenInclude() methods to retrieve the information needed.

## Explicit Data Loading

Use the models and the logic for reading data from the previous problem.

Read a customer id from the console and print the **name**, the **number of orders** and the **number of reviews** of that customer and the **name of the salesman** who serves that customer, each on a new line.

Use the following format for the output:

"Customer: {customer name}"

"Orders count: {orders count}"

"Reviews: {reviews count}"

"Salesman: {salesman name}"

### Examples

|  |  |  |  |
| --- | --- | --- | --- |
| **Input** | **Output** | **Input** | **Output** |
| Ivan;Pesho  Televizor;100  Golqm televizor;200  END  register-Ani;1  order-1;1;2  order-1;1  review-1;2  END  1 | Customer: Ani  Orders count:2  Reviews count: 1  Salesman: Ivan | Icko;Danka  Malak Televizor;20  Sreden Televizor;50  Golqm Televizor;100  B\*\*\* Televizora;300  END  register-Stela;1  register-Manol;2  order-2;1;2;3;4  review-2;3  review-2;4  review-1;2  END  2 | Customer: Manol  Orders count:1  Reviews count: 2  Salesman: Danka |

### Hints

You can use **explicit** data loading.

## Query Loaded Data

Use the models and the logic for reading data from previous problem.

Read a customer id from the console and print the **number of orders** of that customer that have more than 1 item.

Use the following format for the output: "Orders count: {orders count}".

### Examples

|  |  |  |  |
| --- | --- | --- | --- |
| **Input** | **Output** | **Input** | **Output** |
| Ivan;Pesho  Televizor;100  Golqm televizor;200  END  register-Ani;1  order-1;1;2  order-1;1  review-1;2  END  1 | Orders: 1 | Mecho  Pluvka;12  Vadica;20  Stol;18  END  register-Zayko;1  order-1;1  order-1;1;2;3  order-1;2;1  order-1;3;2  END  1 | Orders: 3 |

### Hints

You can use explicit data loading and make a query on the loaded data.