# More Exercises: Lambda and LINQ

Problems for exercises and homework for the [“Programming Fundamentals Extended” course @ SoftUni](https://softuni.bg/courses/programming-fundamentals).

Check your solutions here: <https://judge.softuni.bg/Contests/436>.

## Lambada Expressions

IT Researchers at the MIT (Massachusetts Institute of Technology) have discovered that, at certain (room) temperatures, the lambda expressions in their code are acting strange. In particular, a dancing behavior has been observed in them. The specialists choose to call this phenomenon – The Lambada expressions.

You will be given several input lines in the following format:

{selector} => {selectorObject}.{property}

All elements of the input are **strings**. You need to store every **lambada expression**, as it is **given**.

In some rare cases you will receive the input “**dance**”, which indicates that the lambada expressions are starting to dance. When that happens, you must **COPY** the **selectorObject ONCE** with a **dot** (“**.**”) **between** it., so that it becomes “{selectorObject}.{selectorObject}”

in **every** lambada expression’s **condition**.   
In other words, if you have “x => x.Value”, and you say “**dance**”, you should get “x => x.x.Value”.

Check the examples for more info.

If you meet a **selectorObject** which already, you must **replace** its **property** with the **given new one**.

When you receive the command “lambada” you must end the input sequence. When that happens, you must print all **lambada expressions**.

### Examples

|  |  |
| --- | --- |
| **Input** | **Output** |
| x => x.Key  object => object.Property  entry => entry.Name  lambada | x => x.Key  object => object.Property  entry => entry.Name |
| x => x.Key  x => x.Value  x => x.Name  dance  dance  dance  lambada | x => x.x.x.x.Name |

## Ordered Banking System

You have been given the task to write software for banking – in particular, an optimized program, which stores only the bank account’s name, the bank account’s balance and the bank it is stored at.

You will be given several input lines in the following format:

{bank} -> {account} -> {balance}

The **bank** and the **account** are **strings**, and the **balance** is a **VARIABLE TYPE** which is **SUITABLE** for storing **MONEY**.

You should store every bank, each of its accounts, and their balances.

If you are given a **bank** which **already exists**, you must **ADD** the **new account** to it.  
If **EVEN** the **account** which **already exists**, you must **INCREASE** the **old balance**, with the new **given one**.

When you receive the command “**end**”, you must end the input sequence. Then you must print the data in a specified ordered format.

You must print each bank’s accounts and balances, ordering the **BANKS** by the **sum** of **all account balances**, in **descending order**. If **2 banks** have the **same sum**, you should print the **one** with the **highest maximum balance** in its **accounts**, **first**.

The **bank’s accounts** must be ordered in **descending order**, by **their balance**.

The balances, must be printed, as they are given. “**500**” = “**500**” … “**500.00**” = “**500.00**”.

### Examples

|  |  |
| --- | --- |
| **Input** | **Output** |
| RaifaizenBank -> Toshko -> 200.4456  DSK -> Toshko -> 500.00  PIB -> Ivan -> 400.521  Piraeus -> Ivan -> 212.345  end | Toshko -> 500.00 (DSK)  Ivan -> 400.521 (PIB)  Ivan -> 212.345 (Piraeus)  Toshko -> 200.4456 (RaifaizenBank) |
| DSK -> Tesla -> 2015125.45623  DSK -> Tesla -> 2000000  DSK -> Microsoft -> 400000  DSK -> Apple -> 200000  DSK -> Microsoft -> 350000  DSK -> Apple -> 550000  end | Tesla -> 4 015 125.45623 (DSK)  Microsoft -> 750 000 (DSK)  Apple -> 750 000 (DSK) |
| DSK -> Microsoft -> 50  DSK -> Tesla -> 200  DSK -> Tesla -> 300  DSK -> Apple -> 100  DSK -> Microsoft -> 150  DSK -> Apple -> 100  RaifaizenBank -> Toshko -> 850  RaifaizenBank -> Ivan -> 100  end |  |

## LINQuistics

LINQ is the greatest .NET component of all time. You can do almost anything with it. That’s why you have been tasked to do almost everything you can with it.

You will be given several input lines containing information about collections, and LINQ methods that have been called on them, in the following format:

{collection}.{method1}().{method2}(). . .

The **count** of **methods**, may **vary**. Your task is to store every collection and the methods that have been executed on them. If the collection already exists, you must **ADD** the new methods to it. **Duplicate** methods should be **REMOVED**.

If you are given a **collection name ONLY**, you must print the **methods** that have been executed on the collection, in **descending order** by their **length**. If **2 methods** have the **same length**, **order** them in **descending order**, by the **count** of **unique symbols** they have in **their names**. The methods must be printed – **each** on a **new line**, with an **asterisk** (“\*”) and a **space** (““) **prefix**.  
If the collection name does **NOT** exist, you should **IGNORE** that line of input.

If you are given a **digit** **ONLY**, you must take the collection, with the most methods, and print the **first N methods**, with the **LOWEST** **length** (**N** being the **digit** **given** from the **input**). If there are **less** than **N** methods you must print **all** of them, in the **same order**.

**NOTE**: When printing, you must print **only** the **method name**, without its brackets.

The input sequence ends when you receive the command “**exit**”. After the ending command, you will receive one last line in the following format:  
{method} {selection}

You must **take all** collections, which **CONTAIN** the **given method**, and print them. The selection will either be “**collection**” or “**all**”.   
If you have “collection”, you must print only the collections’ names in the final output.  
If you have “all”, you must print the collections and their methods in the following format:

“{collection}  
 \* {method1}  
 \* {method2}  
 . . .”

The collections, must be printed in **descending order**, by the **count of methods**. If **2 collections** have the **same** **count**, print the **one** with the **higher** **MINIMAL method length**, first. The **methods** must be printed in **descending order** by their **length**. (sort – 4, ThenBy – 6. So “key” is before “elements”)

### Examples

|  |  |
| --- | --- |
| **Input** | **Output** |
| participants.Max().Reverse().ThenBy()  participants.OrderBy.Select()  participants  participants.ToDictionary()  collection.Max()  collection.Break()  exit  Max all | \* OrderBy – 5 unique symbols  \* Reverse – 4 unique symbols  \* ThenBy  \* Select  \* Max  participants  \* ToDictionary  \* Reverse  \* OrderBy - doesn’t matter here.  \* ThenBy  \* Select  \* Max  collection  \* Break  \* Max |
| elements.Sort()  elements.OrderBy()  bound  elements.Reverse().Select().ThenBy()  keys.Reverse().OrderByDescending()  keys.Reverse().ThenByDescending()  3  keys.Reverse().OrderBy().ThenBy()  values.ToString().ToString().ThenBy()  exit  Reverse collection | \* Sort  \* Select  \* ThenBy  keys  elements |
| 3  keys.Reverse().OrderBy().ThenBy()  values.ToString().ToString().ThenBy()  exit  Reverse collection | myTest  (it give exeption now) |