**Exercise: Iterators and Comparators**

Problems for exercise and homework for the ["C# Advanced" course @ Software University](https://softuni.bg/courses/csharp-advanced).

You can check your solutions here: [Iterators and Comparators Exercise](https://judge.softuni.bg/Contests/1490/Iterators-and-Comparators-Exercise)

## ListyIterator

Create a **generic** class **ListyIterator**. The collection, which it will iterate through, should be received in the **constructor**. You should **store** the elements in a **List**. The class should have **three** main functions:

* **Move** - should move an internal index position to the next index in the list. The method should return **true**, if it had successfully moved the index and **false** if there is **no** **next** **index**.
* **HasNext** - should return true, if there is a next index and false, if the index is already at the last element of the list.
* **Print** - should **print** the **element** at the **current** internal **index**. Calling **Print** on a collection without elements should **throw** an appropriate **exception** with the message "**Invalid Operation!**".

By default, the internal index should be pointing to the **0th index** of the **List**. Your program should support the following commands:

|  |  |  |
| --- | --- | --- |
| **Command** | **Return Type** | **Description** |
| Create {e1 e2 …} | void | Creates a ListyIterator from the specified collection. In case of a Create command without any elements, you should create a ListyIterator with an empty collection. |
| Move | boolean | This command should move the internal index to the next index. |
| Print | void | This command should print the element at the current internal index. |
| HasNext | boolean | Returns whether the collection has a next element. |
| END | void | Stops the input. |

Your program should **catch** any **exceptions** thrown because of the described validations - calling **Print** on an **empty** **collection** - and **print** **their messages** instead.

### Input

* Input will come from the console as **lines** of **commands**.
* The first line will always be the **Create** command in the input.
* The last command received will always be the **END** command.

### Output

* For every command from the input (with the exception of the **END** and **Create** commands), print the result of that command on the console, each on a new line.
* In case of **Move** or **HasNext** commands, print the return value of the methods.
* In case of a **Print** command you don’t have to do anything additional as the method itself should already print on the console.

### Constraints

* There will always be only **one** **Create** command and it will always be the first command passed.
* The number of commands received will be between **[1…100]**.
* The last command will always be the only **END** command.

### Examples

|  |  |
| --- | --- |
| **Input** | **Output** |
| Create  Print  END | Invalid Operation! |
| Create Stefcho Goshky  HasNext  Print  Move  Print  END | True  Stefcho  True  Goshky |
| Create 1 2 3  HasNext  Move  HasNext  HasNext  Move  HasNext  END | True  True  True  True  True  False |

## Collection

Using the **ListyIterator** from the last problem, extend it by **implementing** the **IEnumerable<T>** interface, implement all methods desired by the interface manually. Use **yield return** for the **GetEnumerator()** method. Add a new command **PrintAll** that should **foreach** the collection and **print all of the elements** on a **single line separated by a space**. Your program should catch any **exceptions** **thrown** because of **validations** and print their **messages** instead.

### Input

* Input will come from the console as lines of commands.
* The first line will always be the **Create** command in the input.
* The last command received will always be the **END** command.

### Output

* For every command from the input (with the exception of the **END** and **Create** commands), print the result of that command on the console, each on a new line.
* In case of **Move** or **HasNext** commands print the return value of the method
* In case of a **Print** command you don’t have to do anything additional as the method itself should already print on the console.
* In case of a **PrintAll** command you should print all of the elements on a single line separated by spaces.

### Constraints

* **Do NOT use the GetEnumerator() method from the base class. Use your own implementation using "yield return".**
* There will always be only one **Create** command and it will always be the **first** command passed.
* The number of commands received will be between **[1…100]**.
* The **last** command will always be the only **END** command.

### Examples

|  |  |
| --- | --- |
| **Input** | **Output** |
| Create 1 2 3 4 5  Move  PrintAll  END | True  1 2 3 4 5 |
| Create Stefcho Goshky Peshu  PrintAll  Move  Move  Print  HasNext  END | Stefcho Goshky Peshu  True  True  Peshu  False |

## Stack

Create your **custom** **stack**. You are aware of the **Stack's structure**. There is a collection to store the elements and **two** **functions** (not from the functional programming) - to **push** an **element** and to **pop** it. Keep in mind that the **first** **element**, which is popped **is the last** in the collection. The **Push** method **adds an element** at the **top** of the collection and the **Pop** method returns the **top** **element** and **removes** it. Push and Pop will be the only commands and they will come in the following format:

**"Push {element1}, {element2}, … {elementN}**

**Pop**

**… "**

Write your custom implementation of **Stack<T>** and implement **IEnumerable<T>** interface. Your implementation of the **GetEnumerator()** method should follow the rules of the Abstract Data Type – **Stack** (return the elements in reverse order of adding them to the stack).

### Input

* The input will come from the console as lines of commands.
* **Push** and **pop** will be the only possible commands, followed by integers for the **push** command and no another input for the **pop** command.

### Output

* When you receive **END**, the input is over. Foreach the stack **twice** and print all elements each on new line.

### Constraints

* The elements in the push command will be valid integers between **[2-31…231-1]**.
* The commands will always be valid (always be either **Push, Pop** or **END**).
* If Pop command could not be executed as expected (e.g. no elements in the stack), print on the console: "**No elements**".

### Examples

|  |  |
| --- | --- |
| **Input** | **Output** |
| Push 1, 2, 3, 4  Pop  Pop  END | 2  1  2  1 |
| Push 1, 2, 3, 4  Pop  Push 1  END | 1  3  2  1  1  3  2  1 |
| Push 1, 2, 3, 4  Pop  Pop  Pop  Pop  Pop  END | No elements |

## Froggy

Let's play a game. You have a tiny little **Frog**, and a **Lake** that has a path of stones in it. Every **stone has a number**. Our frog must **cross the lake** along that path and **then return**. But there are some rules. First, the frog must **jump on all the stones, which are in even positions** in ascending order and **then on all the odd ones,** but in **reversed order**. The **order** of the **stones** and their **numbers** will be given on the **first line** of input. Then you must **print the order of stones in which our frog jumped** from one to another.



Try to achieve this functionality by creating a **class Lake** (it will hold **all stone numbers in order**) that implements the **IEnumerable<int>** interface and overrides its **GetEnumerator()** methods.

### Examples

|  |  |
| --- | --- |
| **Input** | **Output** |
| 1, 2, 3, 4, 5, 6, 7, 8 | 1, 3, 5, 7, 8, 6, 4, 2 |
| 1, 2, 3, 4, 5 | 1, 3, 5, 4, 2 |
| 13, 23, 1, -8, 4, 9 | 13, 1, 4, 9, -8, 23 |

## Comparing Objects

Create a **class Person**. Each person should have a **name**, an **age** and a **town**. You should implement the interface – **IComparable<T>** and implement the **CompareTo** method. When you compare two people, first you should **compare their names**, after that – **their age** and finally – **their towns**. You will be receiving input with information about the people, until you receive the "**END**" command in the following format:

**"{name} {age} {town}"**

**After that, you will receive n – the n'th person from your collection, starting from 1.** You should bring statistics, how many people are equal to him, how many people are not equal to him and the total people in your collection in the following format:

**"{count of matches} {number of not equal people} {total number of people}"**

If there are no equal people print: **"No matches".**

### Input

* You will be receiving lines in the format described above, until the "END" command.
* After the "END" command, you will receive the position of the person you should compare the others to.

Note: Start counting the people in your collection from 1.

### Output

* Print a single line of output in the format described above.

### Constraints

* Input names, ages and addresses will be valid.
* Input number will always be а valid integer in range **[2…100]**

### Examples

|  |  |
| --- | --- |
| **Input** | **Output** |
| Pesho 22 Vraca  Gogo 14 Sofeto  END  2 | No matches |
| Pesho 22 Vraca  Gogo 22 Vraca  Gogo 22 Vraca  END  2 | 2 1 3 |