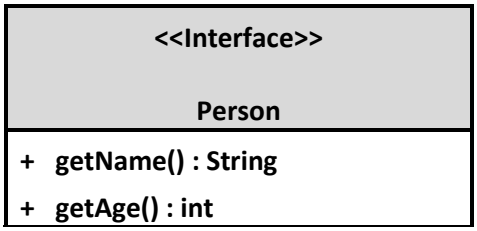
Exercises: Interfaces and Abstraction

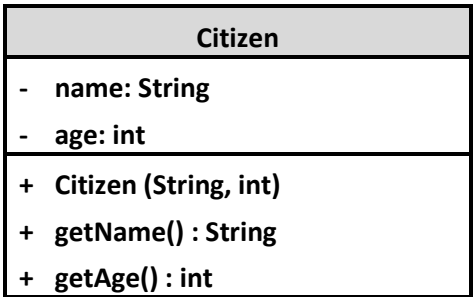
This document defines the exercises for the "Java Advanced" course @ Software University. Please submit your solutions (source code) to all below-described problems in Judge.

Problem 1. Define an Interface Person

Define an interface **Person** with methods **`getName`** and **`getAge`**. Define a class **Citizen** which implements **Person** and has a constructor which takes a **`String` name** and an **`int` age**.

Add the following code to your main method and submit it to Judge.






```
public static void main(String[] args) {
    Class[] citizenInterfaces = Citizen.class.getInterfaces();
    if(Arrays.asList(citizenInterfaces).contains(Person.class)){
        Method[] fields = Person.class.getDeclaredMethods();
        Scanner scanner = new Scanner(System.in);
        String name = scanner.nextLine();
        int age = Integer.parseInt(scanner.nextLine());
        Person person = new Citizen(name,age);
        System.out.println(fields.length);
        System.out.println(person.getName());
        System.out.println(person.getAge());
    }
}
```

If you defined the interface and implemented it correctly, the test should pass.

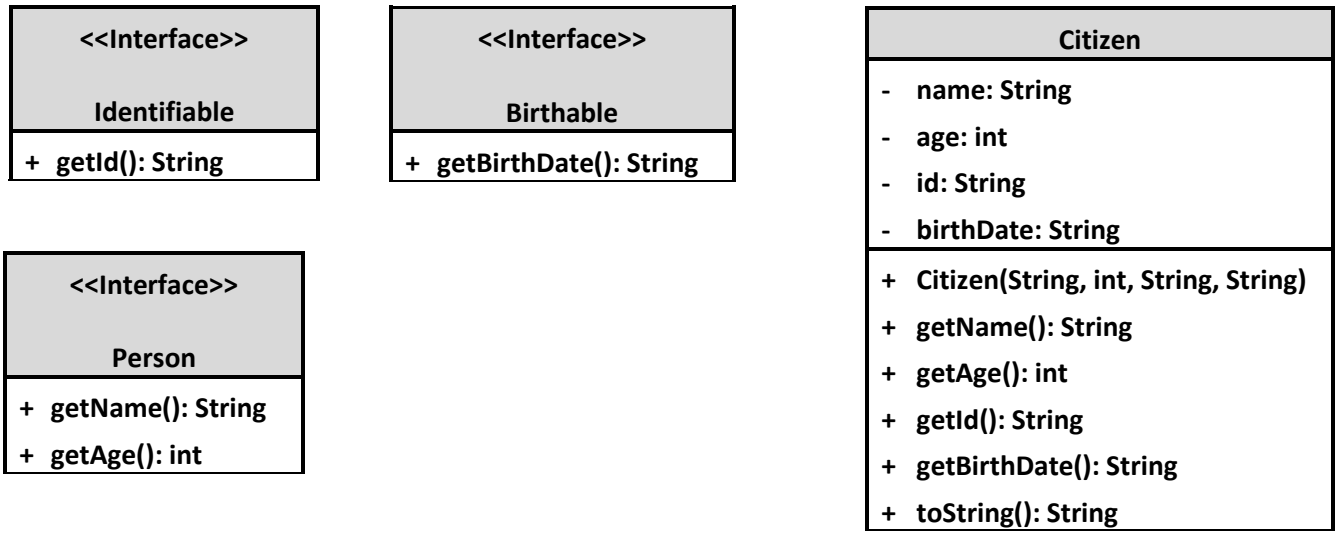
Examples

Input	Output
Peter 25	2 Peter 25
John 34	2 John 34

Problem 2. Multiple Implementation

Using the code from the previous task, define an interface **Identifiable** with a **String** method **getId** and an interface **Birthable** with a **String** method **getBirthDate** and implement them in the **Citizen** class. Rewrite the **Citizen** constructor to accept the new parameters.

Add the following code to your main method and submit it to Judge.




```
public static void main(String[] args) {
    Class[] citizenInterfaces = Citizen.class.getInterfaces();
    if (Arrays.asList(citizenInterfaces).contains(Birthable.class)
            && Arrays.asList(citizenInterfaces).contains(Identifiable.class)) {
        Method[] methods = Birthable.class.getDeclaredMethods();
        Method[] methods1 = Identifiable.class.getDeclaredMethods();
        Scanner scanner = new Scanner(System.in);
        String name = scanner.nextLine();
        int age = Integer.parseInt(scanner.nextLine());
        String id = scanner.nextLine();
        String birthDate = scanner.nextLine();
        Identifiable identifiable = new Citizen(name,age,id,birthDate);
        Birthable birthable = new Citizen(name,age,id,birthDate);
        System.out.println(methods.length);
        System.out.println(methods[0].getReturnType().getSimpleName());
        System.out.println(methods1.length);
        System.out.println(methods1[0].getReturnType().getSimpleName());
    }
}
```

If you defined the interfaces and implemented them, the test should pass.

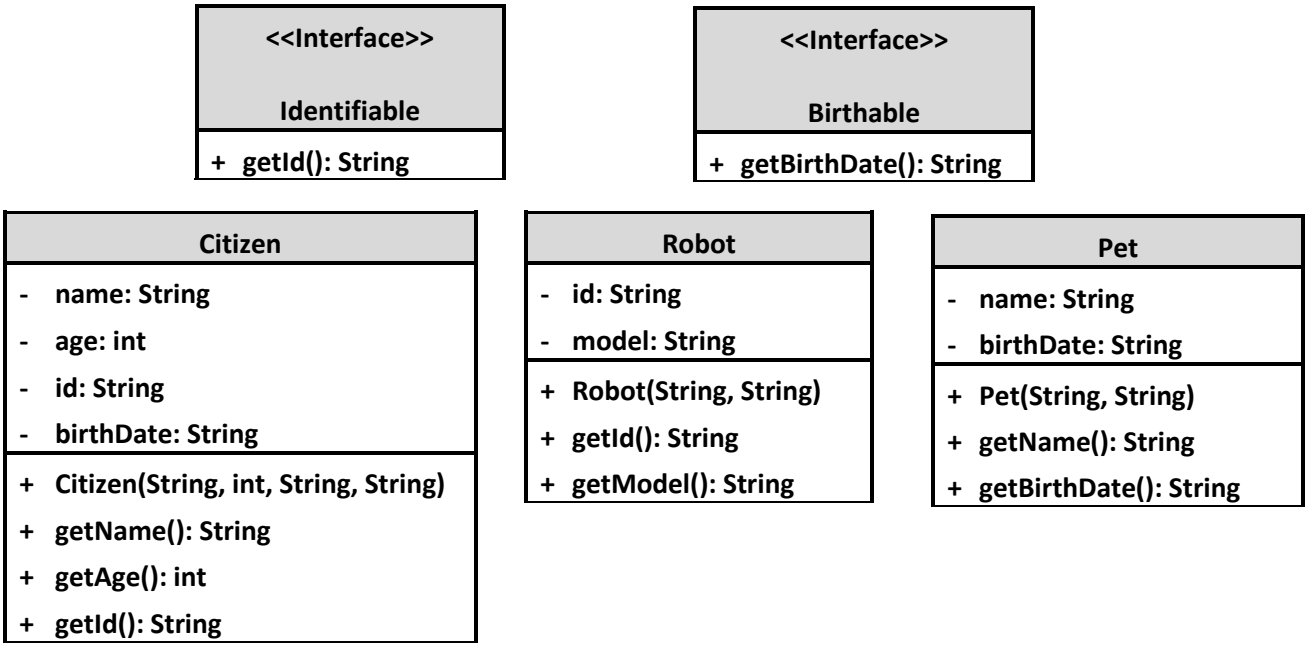
Examples

Input	Output
Peter 25 9105152287 15/05/1991	1 String 1 String

Problem 3. Birthday Celebrations

It is a well-known fact that people celebrate birthdays, it is also known that some people also celebrate their pet's birthdays. Extend the program from your last task to add **birthdates** to citizens and include a class **Pet**, pets have a **name** and a **birthdate**. Also, create a class **Robot** that has an **id** and **model**. Encompass repeated functionality into interfaces and implement them in your classes.

You will receive from the console an unknown amount of lines until the command "**End**" is received, each line will contain information in one of the following formats **"Citizen {name} {age} {id} {birthdate}"** for citizens, "**Robot {model} {id}**" for robots or "**Pet {name} {birthdate}**" for pets. After the end command on the next line, you will receive a single number representing **a specific year**, your task is to print all birthdates (of both citizens and pets) in that year in the format "**{day}/{month}/{year}**" (the order of printing doesn't matter).



Examples

Input	Output
Citizen Peter 22 9010101122 10/10/1990 Pet Sharo 13/11/2005 Robot MK-13 558833251 End 1990	10/10/1990
Citizen Stamo 16 0041018380 01/01/2000 Robot MK-10 12345678 Robot PP-09 00000001 Pet Topcho 24/12/2000 Pet Kosmat 12/06/2002 End 2000	01/01/2000 24/12/2000
Robot VV-XYZ 11213141	<no output>

Citizen Penk 35 7903210713 21/03/1979 Citizen Kane 40 7409073566 07/09/1974 End 1975	

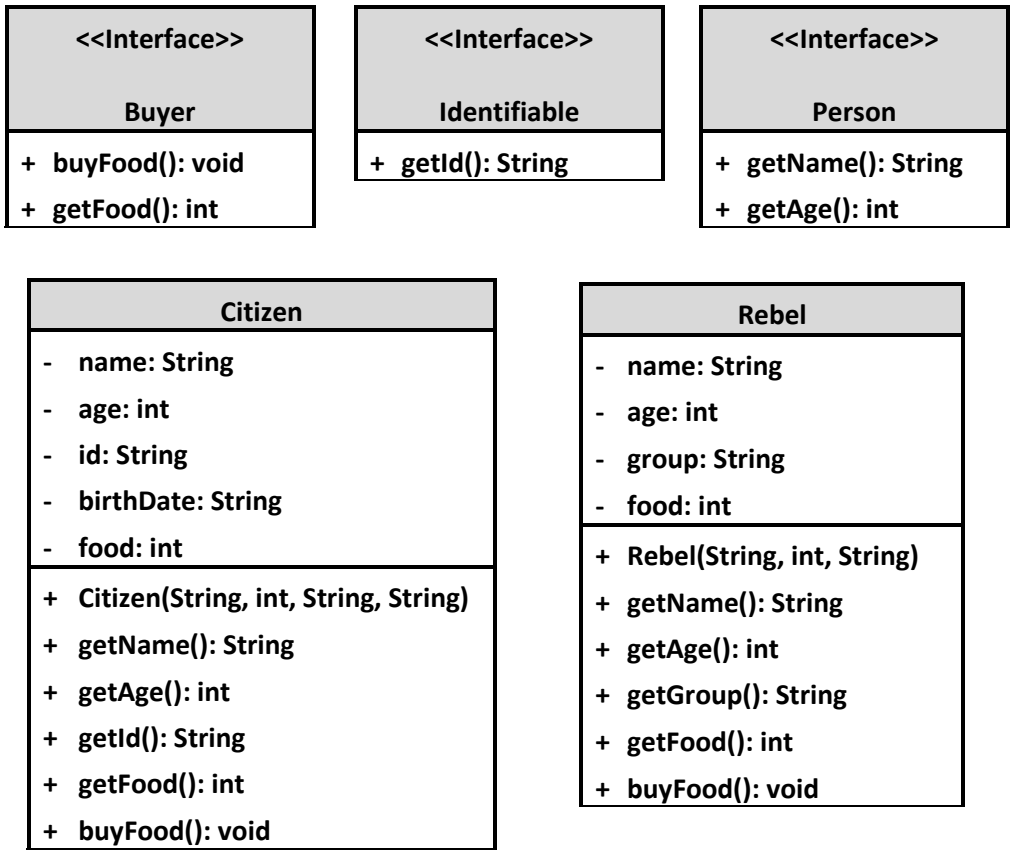
Problem 4. Food Shortage

Your totalitarian dystopian society suffers a shortage of food, so many rebels appear. Extend the code from your previous (Problem 2. **Multiple Implementation**) tasks with new functionality to solve this task.

Define a class **Rebel** which has a **name**, **age,** and **group** (String)**,** names are **unique -** there will never be 2 Rebels/Citizens or a Rebel and Citizen with the same name**.** Define an interface **Buyer** which defines the methods **buyFood()** and a **getFood()**. Implement the **Buyer** interface in the **Citizen** and **Rebel** class, both Rebels and Citizens **start with 0 food**, when a Rebel buys food his **food** increases by **5**, when a Citizen buys food his **food** increases by **10**.

On the first line of the input you will receive an integer **N** - the number of people, on each of the next **N** lines you will receive information in one of the following formats "**{name} {age} {id} {birthdate}**" for a Citizen or "**{name} {age}{group}**" for a Rebel. After the **N** lines, until the command "**End**" is received, you will receive names of people who bought food, each on a new line. Note that not all names may be valid, in the case of an incorrect name - nothing should happen.

On the only line of output, you should print the total amount of food purchased.



Examples

Input	Output

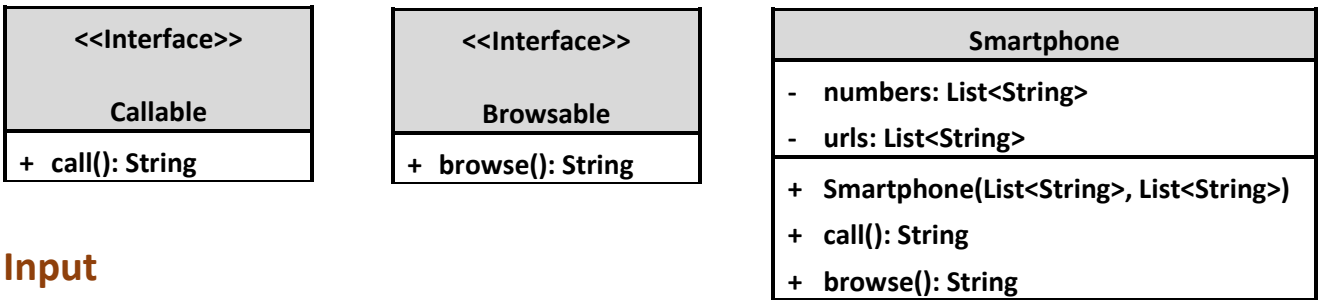
2 Peter 25 8904041303 04/04/1989 Stan 27 WildMonkeys Peter George Peter End	20
4 Stam 23 TheSwarm Tony 44 7308185527 18/08/1973 Joro 31 Terrorists Peny 27 881222212 22/12/1988 Jaguar Joro Jaguar Joro Stam Peny End	25

Problem 5. Telephony

You have a business - **manufacturing cell phones**. But you have no software developers, so you call your friends and ask them to help you create cell phone software. They agree and you start working on the project. The project consists of one main **model - a Smartphone**. Each of your smartphones should have functionalities of **calling other phones** and **browsing on the world wide web.**

Your friends are very busy, so you decide to write the code on your own. Here is the mandatory assignment:

You should have a **model** - **Smartphone** and two separate functionalities which your smartphone has - to **call other phones** and to **browse the world wide web**. You should end up with **one class** and **two interfaces**.



Input

The input comes from the console. It will hold two lines:

- **First line**: **phone numbers** to call (String), separated by spaces.
- **Second line: sites** to visit (String), separated by spaces.

Output

- First, **call all numbers** in the order of input then **browse all sites** in order of input.
- The functionality of calling phones is printing on the console the number which is being called in the format: **`"Calling... {number}"`**.

- The functionality of the browser should print on the console the site in the format:

  `"Browsing: {site}!"` (pay attention to the exclamation mark when printing URLs).
- If there is a number in the input of the URLs, print: **`"Invalid URL!"`** and continue printing the rest of the URLs.
- If there is a character different from a digit in a number, print: **`"Invalid number!"`** and continue to the next number.

Constraints

- Each site's URL should consist only of letters and symbols (**No digits are allowed** in the URL address).

Examples

Input	Output

`0882134215 0882134333 08992134215 0558123 3333 1`
`http://softuni.bg http://youtube.com http://www.g00gle.com`
	`Calling... 0882134215`
	`Calling... 0882134333`
	`Calling... 08992134215`
	`Calling... 0558123`
	`Calling... 3333`
	`Calling... 1`
	`Browsing: http://softuni.bg!`
	`Browsing: http://youtube.com!`
	`Invalid URL!`

`0884542465 0895321654 25632 06014532 123`
`http://softuni.bg http://www.g00gle.com http://facebook.com`
	`Calling... 0884542465`
	`Calling... 0895321654`
	`Calling... 25632`
	`Calling... 06014532`
	`Calling... 123`
	`Browsing: http://softuni.bg!`
	`Invalid URL!`
	`Browsing: http://facebook.com!`

Problem 6. *Military Elite

Create the following class hierarchy:

- **SoldierImpl** – general class for soldiers, holding **id** (int), **first name,** and **last name**
  - **PrivateImpl** – lowest base soldier type, holding the field **salary**(double)
    - **LieutenantGeneralImpl** – holds a set of **PrivatesImpl** under his command
      - **`public void addPrivate(Private priv)`**
    - **SpecialisedSoldierImpl –** general class for all specialized soldiers – holds the **corps** of the soldier. The corps can only be one of the following: **`"Airforces"`** or **`"Marines"`** (Enumeration)
      - **EngineerImpl** – holds a set of **repairs**. A **repair** holds a **part name** and **hours worked** (int)
        - **`public void addRepair(Repair repair)`**
        - **`public Collection<Repair> getRepairs()`**

- **CommandoImpl** – holds a set of **missions**. A mission holds a **code name** and a **state** (Enumeration: **"inProgress"** or **"finished"**). A mission can be finished through the method **completeMission()**
    - **`public void addMission(Mission mission)`**
    - **`public Collection<Mission> getMissions()`**
- **SpyImpl** – holds the **code number** of the spy.

Extract **interfaces** for each class. (e.g. **Soldier**, **Private**, **LieutenantGeneral**, etc.) The interfaces should hold their public get methods (e.g. **Soldier** should hold **getId**, **getFirstName,** and **getLastName**). Each class should implement its respective interface. Validate the input where necessary (corps, mission state) - input should match **exactly** one of the required values, otherwise, it should be treated as **invalid**. In the case of **invalid corps,** the entire line should be skipped, in the case of an **invalid mission state,** only the mission should be skipped.

You will receive from the console an unknown amount of lines containing information about soldiers until the command "**End**" is received. The information will be in one of the following formats:

- Private: "**`Private {id} {firstName} {lastName} {salary}`**"
- LieutenantGeneral: "**`LieutenantGeneral {id} {firstName} {lastName} {salary} {private1Id} {private2Id} … {privateNId}`**" where privateXId will **always** be an **Id** of a private already received through the input
- Engineer: "**`Engineer {id} {firstName} {lastName} {salary} {corps} {repair1Part} {repair1Hours} … {repairNPart} {repairNHours}`**" where repairXPart is the name of a repaired part and repairXHours the hours it took to repair it (the two parameters will always come paired)
- Commando: "**`Commando {id} {firstName} {lastName} {salary} {corps} {mission1CodeName} {mission1state} … {missionNCodeName} {missionNstate}`**" a missions code name, description and state will always come together
- Spy: "**`Spy {id} {firstName} {lastName} {codeNumber}`**"

Define proper constructors. Avoid code duplication through abstraction. Override **toString()** in all classes to print detailed information about the object.

Privates:

```
"Name: {firstName} {lastName} Id: {id} Salary: {salary}"
```

Spy:

```
"Name: {firstName} {lastName} Id: {id}
Code Number: {codeNumber}"
```

LieutenantGeneral:

```
"Name: {firstName} {lastName} Id: {id} Salary: {salary}
Privates:
  {private1 ToString()}
  {private2 ToString()}
  …
  {privateN ToString()}"
```

Note: privates must be sorted by id in **descending order**.

Engineer:

```
"Name: {firstName} {lastName} Id: {id} Salary: {salary}
Corps: {corps}
Repairs:
  {repair1 ToString()}
```

```
  {repair2 ToString()}
  …
  {repairN ToString()}"
```

Commando:

```
"Name: {firstName} {lastName} Id: {id} Salary: {salary}
Corps: {corps}
Missions:
  {mission1 ToString()}
  {mission2 ToString()}
  …
  {missionN ToString()}"
```

Repair:

```
"Part Name: {partName} Hours Worked: {hoursWorked}"
```

Mission:

```
"Code Name: {codeName} State: {state}"
```

NOTE: Salary should be printed and rounded to **two decimal places** after the separator.

Examples

Input	Output

Private 1 Peter Petrov 22.22
Commando 13 Stam Stamov 13.1 Airforces
Private 222 Tom Tomson 80.08
LieutenantGeneral 3 John Johnson 100 222 1
End
	Name: Peter Petrov Id: 1 Salary: 22.22
	Name: Stam Stamov Id: 13 Salary: 13.10
	Corps: Airforces
	Missions:
	Name: Tom Tomson Id: 222 Salary: 80.08
	Name: John Johnson Id: 3 Salary: 100.00
	Privates:
	  Name: Tom Tomson Id: 222 Salary: 80.08
	  Name: Peter Petrov Id: 1 Salary: 22.22

Engineer 7 Peter Petrov 12.23 Marines Boat 2 Crane 17
Commando 19 Sara Johnson 150.15 Airforces HairyFoot finished Freedom inProgress
End
	Name: Peter Petrov Id: 7 Salary: 12.23
	Corps: Marines
	Repairs:
	  Part Name: Boat Hours Worked: 2
	  Part Name: Crane Hours Worked: 17
	Name: Sara Johnson Id: 19 Salary: 150.15
	Corps: Airforces
	Missions:
	  Code Name: HairyFoot State: finished
	  Code Name: Freedom State: inProgress

LieutenantGeneral 17 No Units 500.01 Spy 7 James Bond 007 Spy 8 James Boned 008 End	Name: No Units Id: 17 Salary: 500.01 Privates: Name: James Bond Id: 7 Code Number: 007

	Name: James Boned Id: 8 Code Number: 008

Problem 7. *Collection Hierarchy

Create 3 different string collections – **AddCollection**, **AddRemoveCollection** and **MyListImpl**.

The **AddCollection** should have:

- Only a single method **`add(String)`** which adds an item to the **end** of the collection.

The **AddRemoveCollection** should have:

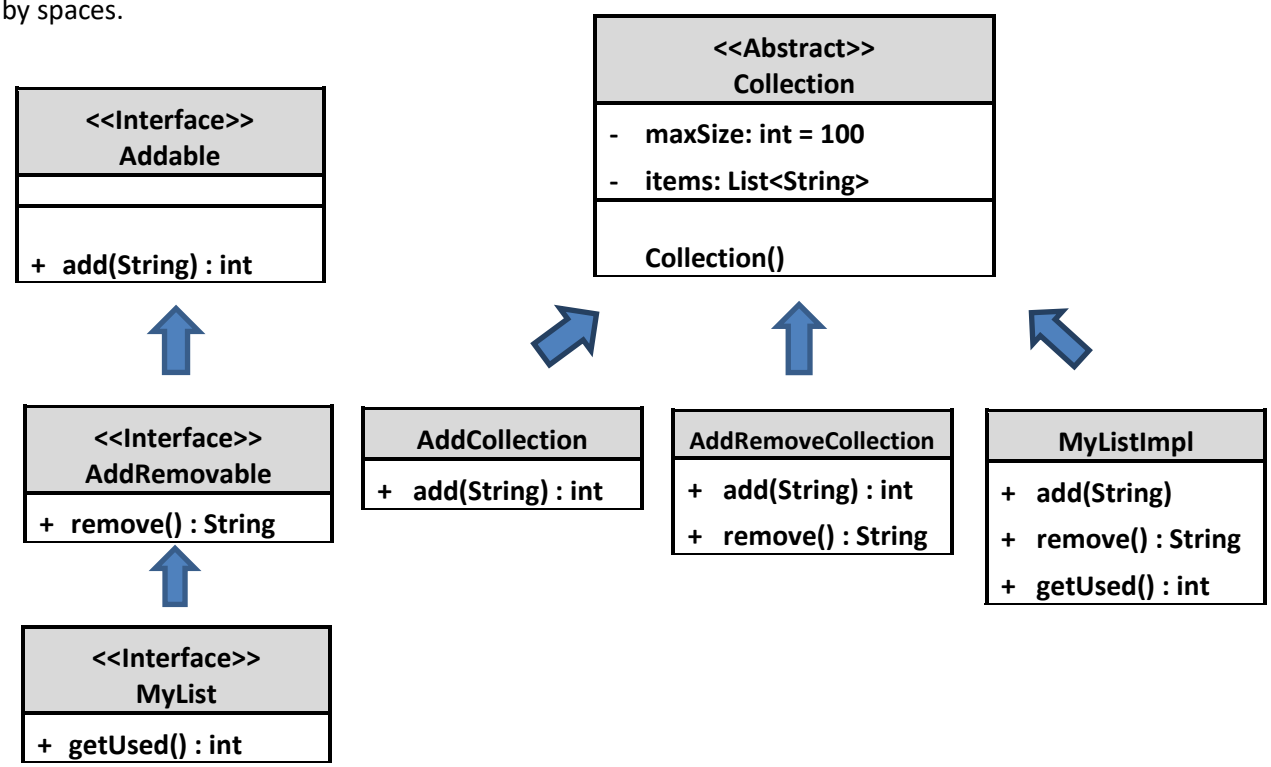
- An **`add(String)`** method – which adds an item to the **start** of the collection.
- A **`remove()`** method removes the **last** item in the collection.

The **MyListImpl** collection should have:

- An **`add(String)`** method adds an item to the **start** of the collection.
- A **`remove()`** method removes the **first** element in the collection.
- A **used** field that displays the size of elements currently in the collection.

Create interfaces that define the functionality of the collection, and think about how to model the relations between interfaces to reuse code. Add an extra bit of functionality to the methods in the custom collections, **add** methods should return the index in which the item was added, **remove** methods should return the item that was removed.

Your task is to create a single copy of your collections, after which on the first input line you will receive a random amount of strings in a single line separated by spaces - the elements you have to add to each of your collections. For each of your collections write a single line in the output that holds the results of all **add operations** separated by spaces (check the examples to better understand the format). On the second input line, you will receive a single number - the amount of **remove operations** you have to call on each collection. In the same manner as with the **add** operations for each collection (except the AddCollection), print a line with the results of each **remove** operation separated by spaces.



Input

The input comes from the console. It will hold two lines:

- The first line will contain a random amount of strings separated by spaces - the elements you have to **add** to each of your collections.
- The second line will contain a single number - the amount of **removed** operations.

Output

The output will consist of 5 lines:

- The first line contains the results of all **add** operations on the **AddCollection** separated by spaces.
- The second line contains the results of all **add** operations on the **AddRemoveCollection** separated by spaces.
- The third line contains the result of all **add** operations on the **MyListImpl** collection separated by spaces.
- The fourth line contains the result of all **remove** operations on the **AddRemoveCollection** separated by spaces.
- The fifth line contains the result of all **removes** operations on the **MyListImpl** collection separated by spaces.

Constraints

- All collections should have a **length of 100.**
- There will never be **more than 100** added operations.
- The number of removed operations will never be more than the amount of added operations.

Examples

Input	Output
apple orange peach 3	0 1 2 0 0 0 0 0 0 apple orange peach peach orange apple
one two three four five six seven 4	0 1 2 3 4 5 6 0 0 0 0 0 0 0 0 0 0 0 0 0 0 one two three four seven six five four

Hint

Create an interface hierarchy representing the collections. You can use a List as the underlying collection and implement the methods using the List's add, remove and insert methods.