Exercises: Interfaces

This document defines the **exercise assignments** for the "Java OOP Advanced" course @ Software University. Please submit your solutions (source code) of all below described problems in https://judge.softuni.bg/Contests/246/Interfaces-and-Abstraction-Exercises .

Problem 1. Define an Interface Person

Define an interface **Person** with properties for **Name** and **Age**. 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
Pesho 25	2 Pesho 25

Problem 2. Multiple Implementation

Using the code from the previous task, define an interface **Identifiable** with a **string** property **Id** and an interface **Birthable** with a **string** property **Birthdate** 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();
        methods = 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(methods.length);
        System.out.println(methods[0].getReturnType().getSimpleName());
    }
}
```

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

Examples

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

Problem 3. Ferrari

Model an application which contains a **class Ferrari** and an **interface**. Your task is simple, you have a **car - Ferrari**, its model is **"488-Spider"** and it has a **driver**. Your Ferrari should have functionality to **use brakes** and **push the gas pedal**. When the **brakes** are pushed down **print "Brakes!"**, and when the **gas pedal** is pushed down - **"Zadu6avam sA!"**. As you may have guessed this functionality is typical for all cars, so you should **implement an interface** to describe it.

Your task is to **create a Ferrari** and **set the driver's name** to the passed one in the input. After that, print the info. Take a look at the Examples to understand the task better.

Input

On the **single input line**, you will be given the **driver's name**.

Output

On the **single output line**, print the model, the messages from the brakes and gas pedal methods and the driver's name. In the following format:

<model>/<brakes>/<gas pedal>/<driver's name>

Constraints

The input will always be valid, no need to check it explicitly! The Driver's name may contain any ASCII characters.

Example

Input	Output
Bat Giorgi	488-Spider/Brakes!/Zadu6avam sA!/Bat Giorgi
Dinko	488-Spider/Brakes!/Zadu6avam sA!/Dinko

Note

To check your solution, copy the code below and paste it to the bottom of the code in your main method.

Reflection

```
String ferrariName = Ferrari.class.getSimpleName();
String carInterface = Car.class.getSimpleName();

boolean isCreated = Car.class.isInterface();

if (!isCreated) {
    throw new IllegalClassFormatException("No interface created!");
}
```

Problem 4. 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 a 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 in 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 in 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 are being called in the format:
  `Calling... <number>`
- The functionality of the browser should print on the console the site in 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`	`Calling... 0882134215`

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

Problem 5. Border Control

It's the future, you're the ruler of a totalitarian dystopian society inhabited by **citizens** and **robots**, since you're afraid of rebellions you decide to implement strict control of who enters your city. Your soldiers check the **Id**s of everyone who enters and leaves.

You will receive from the console an **unknown** amount of lines until the command "**End**" is received, on each line there will be the information for either **a citizen** or **a robot** who tries to enter your city in the format **"<name> <age> <id>**" for citizens and "**<model> <id>**" for robots. After the end command on the next line you will receive a single number representing **the last digits of fake ids**, all citizens or robots whose **Id** ends with the specified digits must be detained.

The output of your program should consist of all detained **Id**s each on a separate line (the order of printing doesn't matter).

Examples

Input	Output
Pesho 22 9010101122 MK-13 558833251 MK-12 33283122 End 122	9010101122 33283122
Toncho 31 7801211340 Penka 29 8007181534 IV-228 999999 Stamat 54 3401018380 KKK-666 80808080 End 340	7801211340

Problem 6. Birthday Celebrations

It is a well known fact that people celebrate birthdays, it is also known that some people also celebrate their pets 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**. 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 Pesho 22 9010101122 10/10/1990 Pet Sharo 13/11/2005 Robot MK-13 558833251 End 1990	10/10/1990
Citizen Stamat 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 Citizen Penka 35 7903210713 21/03/1979 Citizen Kane 40 7409073566 07/09/1974 End 1975	<no output>

Problem 7. Food Shortage

Your totalitarian dystopian society suffers a shortage of food, so many rebels appear. Extend the code from your previous task 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 a method **buyFood()** and a integer property **Food**. 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 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 Pesho 25 8904041303 04/04/1989 Stancho 27 WildMonkeys Pesho Gosho Pesho	20

End	
4 Stamat 23 TheSwarm Toncho 44 7308185527 18/08/1973 Joro 31 Terrorists Penka 27 881222212 22/12/1988 Jiraf Joro Jiraf Joro Stamat Penka End	25

Problem 8. Military Elite

Create the following class hierarchy:

- **Soldier** – general class for soldiers, holding **id**, **first name** and **last name.**
  - **Private** – lowest base soldier type, holding the field **salary**(double).
    - **LeutenantGeneral** – holds a set of **Privates** under his command.
    - **SpecialisedSoldier –** general class for all specialised soldiers – holds the **corps** of the soldier. The corps can only be one of the following: **Airforces** or **Marines**.
      - **Engineer** – holds a set of **repairs**. A **repair** holds a **part name** and **hours worked**(int).
      - **Commando** – holds a set of **missions**. A mission holds **code name** and a **state** (***inProgress*** or ***Finished***). A mission can be finished through the method **CompleteMission()**.
  - **Spy** – holds the **code number** of the spy.

Extract **interfaces** for each class. (e.g. **ISoldier**, **IPrivate**, **ILeutenantGeneral**, etc.) The interfaces should hold their public properties and methods (e.g. **Isoldier** should hold **id**, **first name** and **last name**). 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 case of **invalid corps** the entire line should be skipped, in 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>**"
- LeutenantGeneral: "**LeutenantGeneral <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>

LeutenantGeneral:
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 rounded to **two decimal places** after the separator.

Examples

Input	Output
Private 1 Pesho Peshev 22.22 Commando 13 Stamat Stamov 13.1 Airforces	Name: Pesho Peshev Id: 1 Salary: 22.22 Name: Stamat Stamov Id: 13 Salary: 13.10

Private 222 Toncho Tonchev 80.08
LeutenantGeneral 3 Joro Jorev 100 222 1
End
	Corps: Airforces
	Missions:
	Name: Toncho Tonchev Id: 222 Salary: 80.08
	Name: Joro Jorev Id: 3 Salary: 100.00
	Privates:
	  Name: Toncho Tonchev Id: 222 Salary: 80.08
	  Name: Pesho Peshev Id: 1 Salary: 22.22

Engineer 7 Pencho Penchev 12.23 Marines Boat 2 Crane 17
Commando 19 Penka Ivanova 150.15 Airforces
HairyFoot finished Freedom inProgress
End
	Name: Pencho Penchev Id: 7 Salary: 12.23
	Corps: Marines
	Repairs:
	  Part Name: Boat Hours Worked: 2
	  Part Name: Crane Hours Worked: 17
	Name: Penka Ivanova Id: 19 Salary: 150.15
	Corps: Airforces
	Missions:
	  Code Name: Freedom State: inProgress

Problem 9. *Collection Hierarchy

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

The **AddCollection** should have:

- Only a single method **Add** which adds an item to the **end** of the collection.

The **AddRemoveCollection** should have:

- An **Add** method – which adds an item to the **start** of the collection.
- A **Remove** method which removes the **last** item in the collection.

The **MyList** collection should have:

- An **Add** method which adds an item to the **start** of the collection.
- A **Remove** method which removes the **first** element in the collection.
- A **Used** property which displays the amount of elements currently in the collection.

Create interfaces which define the collections functionality, think 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 **Remove** 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 **MyList** 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 **Remove** operations on the **MyList** collection separated by spaces.

Constraints

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

Examples

Input	Output
banichka boza tutmanik 3	0 1 2 0 0 0 0 0 0 banichka boza tutmanik tutmanik boza banichka
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.

Problem 10. MooD 3

You are an owner of the most epic video game of the world - **3 MooD**. Your employees have gone on summer vacation. But there is a problem in the application and you are on your own. So the problem is how to store the information for the players. The best approach to you, seems to be, storing them in **GameObjects**.

In your game, there are two types of characters - **Demon** and **Archangel**. All characters in the game have:

- **username**
- **hashedPassword**

- **level**
- **special points**.

The **main difference** between the Demon and the Archangel is that the **Demon has an energy** (as special points) and the **Archangel has a mana** (as special points). Your task is to model the application.

When you receive the username and the character type, you should generate the hashed password by the formulas below:

- For a **Demon**: **username length * 217**
- For an **Archangel**: **(username's characters in reversed order) + (username's characters' length * 21)**

Your task is to print the info as it is written in the Output.

Input

The input consists of **single line**. First, you will get the username of a player. The second parameter is its character type. The next two parameters are his mana / energy points and his level. Format:
<**username**> | <**character type**> | <**special points**> | <**level**>

Output

Print the info on two lines, for a single entry in the database (player) in the format:
<"**username**"> | <"**hashed password**"> -> <**character type**>
<**special points * level**>

Constraints

- **Username** – alphabetical letters (**Latin**), no more than 10 characters and you do not need to check it explicitly.
- **Character type** – String, Demon or Archangel, no need to check it explicitly.
- **Special points (Mana)** – a valid Integer, no need to check it explicitly.
- **Special points (Energy)** – a valid Double, no need to check it explicitly.
- **Level** – a valid Integer, no need to check it explicitly.

Example

Input	Output
`"KoHaH" | Demon | 100.0 | 100`	`""KoHaH"" | "1519" -> Demon` `10000.0`
`"Akasha" | Archangel | 5 | 100`	`""Akasha"" | ""ahsakA"168" -> Archangel` `500`

Note

Implement **interface**, holding the **main functionalities of all characters**. Create an **abstract class** to hold all the same characteristics of the characters. If you need to declare a character object, be sure to declare it of type character's

interface to the left side and the specific implementation to the right side of the declaration. You should not override the setter for the hashedPassword and instead, use generics to pass them the type for the password and the special points.