Exercise: Interfaces and Abstraction

Problems for exercise and homework for the "C# OOP" course @ SoftUni".

You can check your solutions here: https://judge.softuni.org/Contests/1502/Interfaces-and-Abstraction-Exercise

1. Define an Interface IPerson

NOTE: You need a public **StartUp** class with the namespace **PersonInfo**.

Define an **interface IPerson** with properties for **Name** and **Age**. Define a class **Citizen** that implements **IPerson** and has two properties **string name** and an **int age.** The **Citizen** should accept **name** and **age** upon initialization.

Try to create a new **Person** like this:

```
string name = Console.ReadLine();
int age = int.Parse(Console.ReadLine());
IPerson person = new Citizen(name, age);
Console.WriteLine(person.Name);
Console.WriteLine(person.Age);
```

Examples

Input	Output
Peter 25	Peter 25

2. Multiple Implementation

NOTE: You need a public **StartUp** class with the namespace **PersonInfo**.

Using the code from the previous task, define an **interface IIdentifiable** with a **string** property **Id** and an **interface IBirthable** with a **string** property **Birthdate** and implement them in the **Citizen** class. Rewrite the **Citizen** constructor to accept the new parameters.

Test your class like this:

```
string name = Console.ReadLine();
int age = int.Parse(Console.ReadLine());
string id = Console.ReadLine();
string birthdate = Console.ReadLine();
IIdentifiable identifiable = new Citizen(name, age,id, birthdate);
IBirthable birthable = new Citizen(name, age, id, birthdate);
Console.WriteLine(identifiable.Id);
Console.WriteLine(birthable.Birthdate);
```

Examples

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

3. Telephony

You have a small business - **manufacturing phones** and to run your business you need to create phone software. The software should support two main phone **models with the following functionality:**

- **`Smartphone`:**
  - Can **calling other phones.**
  - Can **browsing in the world wide web.**
- **`Stationary phone`**:
  - Can **only call other phones.**

You should start the project by implementing two **classes**:

- **The `Smartphone`** can **call other phones** and **browse the world wide web**.
- **The `StationaryPhone`** can only **call other phones**.

You should also implement **interfaces for each class with the appropriate methods**.

Input

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

- **The First line** consists of **phone numbers**: a **`string`**, separated by spaces.
- **The Second** line consists **of websites: a `string`**, separated by spaces.

Output

1. First, **call all valid numbers** in the order of input:
   - If there is a character different from a digit in a number, print: **`"Invalid number!"`** and continue with the next number.
   - If the number is **10 digits long**, you are making a call from your smartphone and print: "**`Calling... {number}"`**
   - If the number is 7 **digits long**, you are making a call from your stationary phone and print: " **`Dialing... {number}"`**
2. Next, **browser all valid websites** in the order of input:
   - If there is a number in the input of the URLs, print: **`"Invalid URL!"`** and continue with the next URLs.
   - If the URL is valid, print on the console the website in the format: **`"Browsing: {site}!"`**

Constraints

- Each site's URL should consist only of letters and symbols (**No digits are allowed** in the URL address).
- The phone numbers will always be 7 or 10 digits long.

Examples

Input	Output
0882134215 0882134333 0899213421 0558123 3333123 http://softuni.bg http://youtube.com http://www.g00gle.com	Calling... 0882134215 Calling... 0882134333 Calling... 0899213421 Dialing... 0558123 Dialing... 3333123 Browsing: http://softuni.bg! Browsing: http://youtube.com! Invalid URL!

4. 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 an unknown amount of lines from the console until the command "**End**" is received, on each line, there will be a piece of 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 in the **order** of **input**.

Input

The input comes from the console. Every commands' parameters before the command "**End**" will be separated by a **single space**.

Examples

Input	Output
Peter 22 9010101122 MK-13 558833251 MK-12 33283122 End 122	9010101122 33283122
Teo 31 7801211340 Peter 29 8007181534 IV-228 999999 Sam 54 3401018380 KKK-666 80808080	7801211340

End 340	
George 954614 Ron 124610 VI-228 999999 Mike 13 7604128614 Peter 90 5602142414 T500 131313130 End 14	954614 7604128614 5602142414

5. 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 number 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 **Citizen**, "**Robot <model> <id>**" for **Robot** or "**Pet <name> <birthdate**" for **Pet**. 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 **Citizen** and **Pet**) in that year in the format **day/month/year** in the **order** of **input**.

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 Stam 16 0041018380 01/01/2000 Robot MK-10 12345678 Robot PP-09 00000001 Pet Topcho 24/12/2000 Pet Rex 12/06/2002 End 2000	01/01/2000 24/12/2000
Robot VV-XYZ 11213141 Citizen Corso 35 7903210713 21/03/1979 Citizen Kane 40 7409073566 07/09/1974 End 1975	<empty output>

6. 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 **Rebel**s/**Citizen**s or a **Rebel** and **Citizen** with the same name**.** Define an interface **IBuyer** which defines a method **BuyFood()** and an integer property **Food**. Implement the **IBuyer** interface in the **Citizen** and **Rebel** class, both **Rebel**s and **Citizen**s **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.

Output

The **output** consists of only **one line** on which 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 Ton 44 7308185527 18/08/1973 George 31 Terrorists Pen 27 881222212 22/12/1988 John Geo rge John Joro Stam Pen End	15

7. *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 **salary**(**decimal**).
    - **LieutenantGeneral** - holds a set of **Privates** under his command.
    - **SpecialisedSoldier** - general class for all specialized **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 a **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** (**int**).

Extract **interfaces** for each class. (e.g. **ISoldier**, **IPrivate**, **ILieutenantGeneral**, 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**: "**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()>

- **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 Peter Johnson 22.22
Commando 13 Sam Peterson 13.1 Airforces
Private 222 Tony Samthon 80.08
LieutenantGeneral 3 George Stevenson 100 222 1
End
	Name: Peter Johnson Id: 1 Salary: 22.22
	Name: Sam Peterson Id: 13 Salary: 13.10
	Corps: Airforces
	Missions:
	Name: Tony Samthon Id: 222 Salary: 80.08
	Name: George Stevenson Id: 3 Salary: 100.00
	Privates:
	  Name: Tony Samthon Id: 222 Salary: 80.08
	  Name: Peter Johnson Id: 1 Salary: 22.22

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

8. *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 number of elements currently in the collection.

Create **interfaces**, which define the functionality of the collection, 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 number of strings** in a single line **separated by spaces** - the **elements** you must **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 number 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 removed operations will never be more than the number of added operations.

Examples

Input	Output
popcorn cola donuts 3	0 1 2 0 0 0 0 0 0 popcorn cola donuts donuts cola popcorn
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.

9. *Explicit Interfaces

Create 2 interfaces **IResident** and **IPerson**. **IResident** should have a **name**, **country,** and method **GetName()**. **IPerson** should have a **name**, an **age,** and a method **GetName()**. Create a class **Citizen** which implements both **IResident** and **IPerson**, explicitly declare that **IResident**'s **GetName()** method should **return** "**Mr/Ms/Mrs** " before the name while **IPerson**'s **GetName()** method should **return just the name**. You will receive lines of **Citizen** information from the console until the command "**End**" is received. Each will be in the format "**<name> <country> <age>**" for each line create the corresponding **Citizen** and print his **IPerson's GetName()** and his **IResitent's GetName()**.

Examples

Input	Output
PeterDavies Bulgaria 20 End	PeterDavies Mr/Ms/Mrs PeterDavies
GeorgeSmith Bulgaria 33 EricAnderson GreatBritain 28 PeterArmstrong USA 19 End	GeorgeSmith Mr/Ms/Mrs GeorgeSmith EricAnderson Mr/Ms/Mrs EricAnderson PeterArmstrong Mr/Ms/Mrs PeterArmstrong

Hint

Check online about Explicit Interface Implementation.