

Exercises: Communication and Events

This document defines the exercises for ["Java OOP Advanced" course @ Software University](#). Please submit your solutions (source code) of all below described problems in <https://judge.softuni.bg/Contests/256/Object-Communication-and-Events-Exercises>

Problem 1. Event Implementation

Create a class **Dispatcher** with a property **name** and a class **Handler**. Create an **Event** (a class that extends **EventArgs**) **NameChange** - which holds a string **changed name** that it receives from the constructor and has a getter for it. Create also an interface **NameChangeListener** that defines a method **handleChangedName(NameChange event)**.

Implement the **NameChangeListener** in the **Handler** class, the implementation should write on the console **"Dispatcher's name changed to <newName>"**. Create in the **Dispatcher** a list of **NameChangeListeners** and 3 methods - **addNameChangeListener**, **removeNameChangeListener** and **fireNameChangeEvent**. As the names imply the add method should add a new **NameChangeListener** to the list the remove method should remove an existing **NameChangeListener** and the fire method should fire the event to all listeners in the list (call their **handleChangedName** method).

At the start of your program create a new **Dispatcher** and **Handler**, then add the **Handler** to the list of **NameChangeListeners** in the **Dispatcher**.

Input

From the console you will receive lines containing a name until the **"End"** command is received. For each name change the dispatcher's name to it. Everytime the Dispatcher's name is changed, you should fire an event to all observers.

Output

For each name change of the dispatcher the handler should print **"Dispatcher's name changed to <newName>."** on the console.

Constraints

- Names will contain only alphanumerical characters.
- The number of commands will be a positive integer between [1...100].
- The last command will always be the only **"End"** command.

Examples

Input	Output
Pesho Gosho Stefan End	Dispatcher's name changed to Pesho. Dispatcher's name changed to Gosho. Dispatcher's name changed to Stefan.

Prakash	Dispatcher's name changed to Prakash.
Stamat	Dispatcher's name changed to Stamat.
MuadDib	Dispatcher's name changed to MuadDib.
Ivan	Dispatcher's name changed to Ivan.
Joro	Dispatcher's name changed to Joro.
End	

Problem 2. King's Gambit

Implement 3 classes - **King**, **Footman** and **Royal Guard**. All of them have a **name** (names are **unique** there will never be two units with the same name), Footmen and Royal Guards can also be **killed** (killed units are removed from the program), while the king is **attackable** - should have a method to respond to attacks. Whenever the king is attacked, he should print to the console "**King <kingName> is under attack!**" and all **alive** Footmen and Royal guards should respond to the attack:

- **Footman** respond by writing to the console "**Footman <footmanName> is panicking!**".
- **Royal Guards** instead write "**Royal Guard <guardName> is defending!**".

Input

On the first line of the console you will receive a single string - the name of the **king**. On the second line you will receive the names of his **Royal Guards** separated by spaces. On the third the names of his **Footmen** separated by spaces. On the next lines until the command "**End**" is received, you will receive commands in one of the following format:

- "**Attack King**" - calls the king's respond to attack method.
- "**Kill <name>**" - the Footman or Royal Guard with the given name is killed.

Output

Whenever the king is attacked you should print on the console "**King <kingName> is under attack!**" and each **living** Footman and Royal Guard should print **their response message** - first all Royal Guards should respond (in the order that they were received from the input) and then all Footmen should respond (in the order that they were received from the input). Every message should be printed on a new line.

Constraints

- Names will contain only alphanumerical characters.
- There will **always** be a **king** and at least **one Footman** and **one Royal Guard**.
- The king **cannot be killed** - there will never be a kill command for the king.
- Kill commands will be received only for living soldiers.
- All commands received will be valid commands in the formats described.
- The number of commands will be a positive integer between **[1...100]**.
- The last command will always be the only "**End**" command.

Examples

Input	Output
-------	--------

Pesho Krivogled Ruboglav Gosho Pencho Stamat Attack King End	King Pesho is under attack! Royal Guard Krivogled is defending! Royal Guard Ruboglav is defending! Footman Gosho is panicking! Footman Pencho is panicking! Footman Stamat is panicking!
HenryVIII Thomas Oliver Mark Kill Oliver Attack King Kill Thomas Kill Mark Attack King End	King HenryVIII is under attack! Royal Guard Thomas is defending! Footman Mark is panicking! King HenryVIII is under attack!

Problem 3. Dependency Inversion

You are given a skeleton of a simple project. The project contains a class Primitive Calculator which supports two methods - **changeStrategy(char operator)** and **performCalculation(int firstOperand, int secondOperand)**. The **performCalculation** method should perform a mathematical operation on the two operands based on the Primitive Calculator's current Strategy and the **changeStrategy** should change the calculator's current Strategy. Currently the calculator supports only adding and subtracting strategies, think how to refactor the **changeStrategy** and **performCalculation** method to allow the Primitive Calculator to support any strategy. Add functionality to the Primitive calculator to support multiplying and dividing of elements.

The calculator should start by **default** in **addition** mode. Currently the **changeStrategy** method can switch only between 2 Strategies based on a character received by the method. The currently supported strategies are:

- "+" for addition
- "-" for subtraction

Input

From the console you will receive lines in one of the following formats until the command **"End"** is received:

- "**<number> <number>**" - perform calculation on the current numbers based on the current mode of the calculator.
- "**mode <operator>**" - changes the mode of the calculator to the specified one.

Output

Print the results of the calculation of all number lines - each result on a new line.

Constraints

- You are allowed to refactor the Primitive Calculator class, but you're **NOT** allowed to add additional methods to it like Addition method, Subtraction method and so on.
- The **operators** received from the console will always be valid ones specified in the calculator modes section.
- The **result of the calculations** should also be an **integer**.
- There will **never** be a 0 divisor.

- The last command will always be the “End” command.

Examples

Input	Output
10 15 mode / 20 5 17 7 mode - 30 31 End	25 4 2 -1
mode * 1 1 3 21 -5 -6 mode - -30 -50 mode / -28 4 mode + 1 10 End	1 63 30 20 -7 11

Problem 4. **Work Force

Create two classes - **StandartEmployee** and **PartTimeEmployee**, both of which have a **name** and **work hours per week**. The **StandartEmployee**’s work hours per week are always **40** and the **PartTimeEmployee**’s work hours per week are always **20**. Create a class **Job** which should receive an employee through its constructor, have fields - **name** and **hours of work required** and a method **update** which should subtract from its **hours of work required** the employee’s **work hours per week**. Whenever a job’s **hours of work required** reaches **0 or less** it should print “**Job <jobName> done!**” and find a way to notify the collection you hold all jobs in, that it is done and should be deleted from the collection.

Input

From the console you will receive lines in one of the following formats until the command “End” is received:

- “**Job <nameOfJob> <hoursOfWorkRequired> <employeeName>**” - should create a Job with the specified name, hours of work required and employee.
- “**StandartEmployee <name>**” - should create a Standart Employee with the specified name.
- “**PartTimeEmployee <name>**” - should create a Part Time Employee with the specified name.
- “**Pass Week**” - should call each job’s **update** method.
- “**Status**” - should print the status of all jobs in the following format “**Job: <jobName> Hours Remaining: <hoursOfWorkRequired>**”.

Output

Every time a job ends the message “**Job <jobName> done!**” should be printed on the console. Every time a **Status** command is received all jobs **currently active** (not completed) should be printed on the console in the format specified on the **Status**,

in order of being receiving them from the input - each message on a new line.

Constraints

- All names will consist of alphanumerical characters.
- All **hours of work required** will be valid positive integers between [1...1000].
- The employee specified in the Job input line will **always** be a valid existing employee.
- Employee and Job names are **unique** - there will never be two Employee/Jobs with the same name.
- The last command will always be "End".

Examples

Input	Output
StandartEmployee Pesho PartTimeEmployee Penka Job FeedTheFishes 45 Pesho Pass Week Status Pass Week End	Job: FeedTheFishes Hours Remaining: 5 Job FeedTheFishes done!
PartTimeEmployee Penka PartTimeEmployee Vanka PartTimeEmployee Stanka Job Something 177 Stanka Pass Week Job AnotherThing 33 Vanka Status Pass Week Pass Week Pass Week Status End	Job: Something Hours Remaining: 157 Job: AnotherThing Hours Remaining: 33 Job AnotherThing done! Job: Something Hours Remaining: 97

Hint

Find a way to have your collection respond to events. Create your own class extending the ArrayList and implementing an EventListener to a custom event which is triggered when a job is done. Use abstraction in the Job class to allow for different type of employees to be accepted - i.e. extract an interface for employees and have the Job class accept an object from that implements the interface instead of a concrete class.

Problem 5. **King's Gambit Extended

Extend your code from **Problem 2 King's Gambit** - normal **Footmen** should now die in **2 hits** (you would have to receive 2 Kill commands with their name from the input to kill them), while **Royal Guards** should die from **3 hits**. Dead Footmen and Royal Guards should still not respond to the king being attacked and be deleted from the collection of units. Find a way for the dying soldiers to communicate their deaths to the king and the collection holding them without you manually checking their state at each Kill command (i.e. use Events).

Input

On the first line of the console you will receive a single string - the name of the **king**. On the second line you will receive the names of his

royal guards separated by spaces. On the third the names of his **Footmen** separated by spaces. On the next lines until the command **“End”** is received, you will receive commands in one of the following format:

- **“Attack King”** - calls the king’s respond to attack method.
- **“Kill <name>”** - the Footman or Royal Guard with the given name is attacked, if this is the second Kill command for Footmen or the third for Royal Guards - they are killed.

Output

Whenever the king is attacked you should print on the console **“King <kingName> is under attack!”** and each **living** footman and royal guard should print **their response message** - first all royal guards should respond (in the order that they were received from the input) and then all footmen should respond (in the order that they were received from the input). Every message should be printed on a new line.

Constraints

- Names will contain only alphanumerical characters.
- There will **always** be a **king** and at least **one Footman** and **one Royal Guard**.
- The king **cannot be killed** - there will never be a kill command for the king.
- All commands received will be valid commands in the formats described.
- Kill commands will be received only for living soldiers.
- The number of commands will be a positive integer between **[1...100]**.
- The last command will always be the only **“End”** command.

Examples

Input	Output
Pesho Ruboglav Gosho Stamat Kill Gosho Kill Stamat Attack King Kill Gosho Attack King End	King Pesho is under attack! Royal Guard Ruboglav is defending! Footman Gosho is panicking! Footman Stamat is panicking! King Pesho is under attack! Royal Guard Ruboglav is defending! Footman Stamat is panicking!
HenryVIII Thomas Mark Kill Thomas Kill Mark Attack King Kill Thomas Kill Thomas Kill Mark Attack King End	King HenryVIII is under attack! Royal Guard Thomas is defending! Footman Mark is panicking! King HenryVIII is under attack!

Problem 6: *Mirror Image

You are given Mirror images, wizards, reflections... or not.

Basically, you have a Wizard which has an **id**, **name** and **magical power**. The id of the initial wizard is always zero.

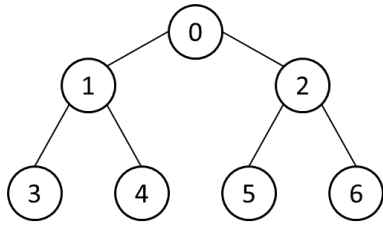
He can cast **two spells** - **Reflection** and **Fireball**.

Reflection creates two mirror images of the wizard with **ids representing the instance of a reflection being created starting from 1**. A mirror image has half the magical power of its creator. Every wizard/mirror image **cannot have more than two own mirror images**.

Fireball creates a ball of fire which deals damage equal to its caster's magical power.

When a wizard casts a spell, all of his images cast the same spell after him. The same is true for the mirror images.

Consider the picture below.

Picture	Explanation
	If the initial wizard (number 0) casts Reflection two times you will first end up with mirror images 1 and 2, then with 3, 4 followed by 5 and 6. After that, if wizard 0 casts Fireball, the sequence in which his mirror images will cast fireball after him would be 1, 3, 4 and then 2, 5 and lastly 6.

Create a program which by a given id and a spell name makes the corresponding wizard or mirror image cast the spell.

Input

On the first line you will get the name and magical power of the initial wizard. Next the input will come as commands from the console and will be one of the following, until an "END" command is given:

- "<id> REFLECTION" - The wizard casts a Reflection spell
- "<id> FIREBALL" - The wizard casts a Fireball spell
- "END" - Program execution stops

Output

The output should be printed as follows:

- For Reflection spell - "<name> <id> created a <name> <id>"
- For Fireball spell - "<name> <id> casts Fireball for <magical power> damage"

Constraints

- The id of the initial wizard always starts from zero.

- Every magical power division is made on integers, so half the magical power of 11 is 5.

Examples

Input	Output
Oz 12	Oz 0 casts Fireball for 12 damage
0 FIREBALL	Oz 0 casts Reflection
0 REFLECTION	Oz 0 casts Reflection
0 REFLECTION	Oz 1 casts Reflection
3 FIREBALL	Oz 2 casts Reflection
4 FIREBALL	Oz 3 casts Fireball for 3 damage
5 FIREBALL	Oz 4 casts Fireball for 3 damage
6 FIREBALL	Oz 5 casts Fireball for 3 damage
END	Oz 6 casts Fireball for 3 damage

Problem 7: *1984

The institutions are always watching you.

You are given three different entities - **Employees**, **Companies** and **Institutions**. Employees have **name** and **income**. Companies have **name**, **turnover** and **revenue**. Employees and Companies have an **id** which is unique and never changes. Everything else can change.

Institutions have **name** and one or more **subjects which they are monitoring**. Every institution always watches if a company or an employee has changed some of their attributes and saves the change if it is within its subject.

For example, an Institution named "NAP" monitors "income". If an Employee changes its income, the Nap Institution saves this change.

A change should have an information about the entity that has changed (e.g. employee or company), the type of the changed property (e.g. string or double), the name of the changed property (e.g. income or name), the old value and the new value.

Save all changes in format "--<entity>(ID:<id>) changed <field name>(<field type>) from <old value> to <new value>". For example, "--Employee(ID:1) changed name(String) from Gosho to Misho".

Input

The first line consists of 3 numbers - **M**, **N** and **P**.

M - the number of entities, **N** - the number of institutions and **P** - the number of changes entities will make.

- On the next M lines, you will get the information about all entities in one of the following formats "**Employee <Id> <Name> <Income>**" for employees or "**Company <Id> <Name> <Turnover> <Revenue>**" for companies.
- On the next N lines, you will get the information about all institutions in the format "**Institution <Id> <Name> <subject 1> <subject 2> .. <subject n>**".
- On the next P lines, you will get an id which changes some of its fields in the format "<Id> <Field name> <New value>".

Output

Print all Institution logs starting with the name of the institution and the number of saved changes:

"<Institution name>: <Number of changes> changes registered"

"--<entity type>(ID:<id>) changed <field name>(<field type>) from <old value> to <new value>"

For example:

"NAP: 1 changes registered"

"--Employee changed int field "income" from 1200 to 1300"

Print the change logs of all Institutions in the order of their appearance.

Constraints

- Id is unique, cannot be changed and should be saved in a string variable.
- Name can be changed and should be saved in a string variable.
- All number fields (income, turnover and revenue) can be changed and should be saved in an int variables.
- The input will always be valid and in the format described. You will never receive a non-existent attribute to monitor, field name to change or a non-existent id.

Examples

Input	Output
2 3 2 Employee 1 Gosho 900 Company 2 Magazin 5000 1000 Institution 3 MVR name Institution 4 NAP turnover Institution 5 DANS income 1 name Misho 2 turnover 6000	MVR: 1 changes registered --Employee(ID:1) changed name(String) from Gosho to Misho NAP: 1 changes registered --Company(ID:2) changed turnover(int) from 5000 to 6000 DANS: 0 changes registered
2 3 4 Employee 1 Gosho 900 Company 2 Magazin 5000 1000 Institution 3 MVR name turnover Institution 4 NAP turnover Institution 5 DANS income 1 name Misho 2 name BookShop 2 turnover 4000 1 income 1200	MVR: 3 changes registered --Employee(ID:1) changed name(String) from Gosho to Misho --Company(ID:2) changed name(String) from Magazin to BookShop --Company(ID:2) changed turnover(int) from 5000 to 4000 NAP: 1 changes registered --Company(ID:2) changed turnover(int) from 5000 to 4000 DANS: 1 changes registered --Employee(ID:1) changed income(int) from 900 to 1200

Hint

Try using Observer pattern, reflection, annotations and generics for a cleaner solution.