# **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 Judge.

# **Problem 1. Event Implementation**

Create a class Dispatcher with a property name and a class Handler. Create an Event (a class that extends EventObject) 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

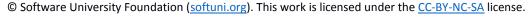
For each name change of the dispatcher the handler should print "Dispatcher's name changed to <newName>." on the console.

#### **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
Gosho	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

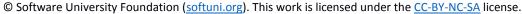
Whenever the king is attacked you should print on the console "King <kingName> is under attack!" and each living Footman and Royal Guard should print their response message - first all Royal Guards should respond (in the order that they were received from the input) and then all Footmen should respond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

© Software University Foundation (softuni.org). This work is licensed under the CC-BY-NC-SA license.



















The last command will always be the "End" command.

### **Examples**

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

### 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,

© Software University Foundation (softuni.org). This work is licensed under the CC-BY-NC-SA license.



















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

© Software University Foundation (softuni.org). This work is licensed under the CC-BY-NC-SA license.



















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	King HenryVIII is under attack! Royal Guard Thomas is defending! Footman Mark is panicking! King HenryVIII is under attack!



















