Exercises: Enumerations and Annotations

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

Problem 1. Card Suit

Create an **enumeration type** that has as its constants the **four suits** of a deck of playing cards (CLUBS, DIAMONDS, HEARTS, SPADES). Iterate over the values of the enumeration type and print all **ordinal values** and **names**.

Examples

Input	Output

`Card Suits`	`Card Suits:`
	`Ordinal value: 0; Name value: CLUBS`
	`Ordinal value: 1; Name value: DIAMONDS`
	`Ordinal value: 2; Name value: HEARTS`
	`Ordinal value: 3; Name value: SPADES`

Problem 2. Card Rank

Create an **enumeration type** that has as its constants the **fourteen ranks** of a deck of playing cards (ACE, TWO, THREE, FOUR, FIVE, SIX, SEVEN, EIGHT, NINE, TEN, JACK, QUEEN, KING). Iterate over the values of the enumeration type and print all ordinal values and names.

Examples

Input	Output

`Card Ranks`	`Card Ranks:`
	`Ordinal value: 0; Name value: ACE`
	`Ordinal value: 1; Name value: TWO`
	`Ordinal value: 2; Name value: THREE`
	`Ordinal value: 3; Name value: FOUR`
	`Ordinal value: 4; Name value: FIVE`
	`Ordinal value: 5; Name value: SIX`
	`Ordinal value: 6; Name value: SEVEN`
	`Ordinal value: 7; Name value: EIGHT`
	`Ordinal value: 8; Name value: NINE`
	`Ordinal value: 9; Name value: TEN`
	`Ordinal value: 10; Name value: JACK`
	`Ordinal value: 11; Name value: QUEEN`
	`Ordinal value: 12; Name value: KING`

Problem 3. Cards with Power

Create a program that generates a **deck of cards (class Card)** which have a power. The power of a card is calculated by adding the power of its rank plus the power of its suit.

Rank powers are as follows: (ACE - 14, TWO - 2, THREE - 3, FOUR - 4, FIVE - 5, SIX - 6, SEVEN - 7, EIGHT - 8, NINE - 9, TEN - 10, JACK - 11, QUEEN - 12, KING - 13).

Suit powers are as follows: (CLUBS - 0, DIAMONDS - 13, HEARTS - 26, SPADES - 39).

You will get a command consisting of two lines. On the first line you will receive the Rank of the card and on the second line you will get the suit of the card.

Print the output in the format "Card name: ACE of SPADES; Card power: 53".

Note

Try using the enumeration types you have created in the previous problems but extending them with constructors and methods. Try using the **Enum.valueOf().**

Examples

Input	Output
TWO CLUBS	Card name: TWO of CLUBS; Card power: 2
ACE SPADES	Card name: ACE of SPADES; Card power: 53

Problem 4. Card toString()

If you haven't done it already, try using built-in annotations to override the **toString()** of your Card class you've created earlier. Make it so it returns the same information as before e.g. in format:

"Card name: {Rank} of {Suit}; Card power: {Card power}"

Note

Pay attention to the actual overriding of the method.

Examples

Input	Output
TWO CLUBS	Card name: TWO of CLUBS; Card power: 2
ACE SPADES	Card name: ACE of SPADES; Card power: 53

Problem 5. Card compareTo()

As your cards have power you can safely add a functionality for comparing them. Try using the ready available interface and the built-in annotations to override the **compareTo().**

Read two cards from the console and print the greater of the two. In the given format:

"Card name: {Rank} of {Suit}; Card power: {Card power}"

Note

Pay attention to the actual overriding of the method.

Examples

Input	Output
TWO CLUBS ACE SPADES	Card name: ACE of SPADES; Card power: 53

Problem 6. Custom Enum Annotation

Create a custom annotation that can be applied to **classes** and can be accessed at **runtime**. The annotation **type** elements it should contain are **category** and **description**. Apply the annotation to both enumeration types you have created for the previous problems (**Rank** and **Suit**). Provide them these exact values:

Rank:

- **type** = "Enumeration"
- **category** = "Rank"
- **description** = "Provides rank constants for a Card class."

Suit:

- **type** = "Enumeration"
- **category** = "Suit"
- **description** = "Provides suit constants for a Card class."

Create a program which gets the description and type by a given category: “Rank” or “Suit”.

Note

Try using the **getAnnotation()** method.

Examples

Input	Output
Rank	Type = Enumeration, Description = Provides rank constants for a Card class.

Problem 7. Deck of Cards

Create a program that generates **all cards** of a card playing deck. First print the clubs, starting from the ace, ending with a king. Continue with the same cards from diamonds, hearts and spades. Print them in the format below.

Note

Try using the enumeration types you have created in the previous problems.

Examples

Input	Output
Card Deck	ACE of CLUBS TWO of CLUBS THREE of CLUBS

	FOUR of CLUBS FIVE of CLUBS ... ... KING of SPADES

Problem 8. Card Game

Simulate a card game in which you have two players. Each player has a hand of five cards. The winning player is the player which holds the highest powered card in his hand.

Rank powers are as follows: (ACE - 14, TWO - 2, THREE - 3, FOUR - 4, FIVE - 5, SIX - 6, SEVEN - 7, EIGHT - 8, NINE - 9, TEN - 10, JACK - 11, QUEEN - 12, KING - 13).

Suit powers are as follows: (CLUBS - 0, DIAMONDS - 13, HEARTS - 26, SPADES - 39).

Input

On the first two lines you will get the names of the players.

On the next lines, you should **read cards** from the console in the format **{ACE of CLUBS}** for a certain player until he has **exactly 5 cards in his hand**. If he receives a card that is not in the deck, you should print "Card is not in the deck.". If he receives an invalid card name, for example "spades of ace", print "No such card exists.".

Output

Print the name of the winner and his winning card in the format "`{Player name} wins with {Card name}.`".

Examples

Input	Output	Comments
Ivo Gosho QUEEN of DIAMONDS KING of DIAMONDS **ACE of HEARTS** ACE of HEARTS spades of ace TWO of HEARTS THREE of HEARTS FOUR of HEARTS FIVE of HEARTS SIX of HEARTS SEVEN of HEARTS EIGHT of HEARTS	Card is not in the deck. No such card exists. Ivo wins with ACE of HEARTS.	Player Ivo receives cards (in orange) from the deck, until he has exactly five of them. When he is given ACE of HEARTS for a second time, error message is printed and his hand stays the same size. When a card with invalid name is given, error message is printed and his hand stays the same size. When Ivo's hand has 5 cards, Gosho starts receiving cards from the deck. When Gosho has 5 cards, the hands are evaluated and one of the players wins.

Problem 9. Traffic Lights

Implement a simple state machine in the form of a traffic light. Every traffic light has **three** possible signals - **red**, **green** and **yellow**. Each traffic light can be updated, which changes the color of its signal (e.g. if it is currently red, it changes to green, if it is green it changes to yellow). The order of signals is red -> green -> yellow -> red and so on.

On the first line you will be given multiple traffic light signals in the format "RED GREEN YELLOW". They may be 3, more or less than 3. You need to make as many traffic lights as there are signals in the input.

On the second line, you will receive the **n** number of times you need to change each traffic light's signal.

Your output should consist of **n** number of lines, including each updated traffic light's signal. To better understand the problem, see the example below.

Examples

Input	Output
GREEN RED YELLOW 4	YELLOW GREEN RED RED YELLOW GREEN GREEN RED YELLOW YELLOW GREEN RED

Problem 10. *Inferno Infinity

If you've been involved with the creation of Inferno III last year, you may be informed of the disastrous critic reception it has received. Nevertheless, your company is determined to satisfy its fan base, so a sequel is coming and yeah, you will develop the crafting module of the game using the latest OOP trends.

You have three different weapons (Axe, Sword and Knife) which have **base stats** and a **name**. The base stats are **min** damage, **max** damage and number of **socket**s (sockets are basically holes, in which you can insert gems). Below are the base stats for the three weapon types:

- **Axe** (5-10 damage, 4 sockets)
- **Sword** (4-6 damage, 3 sockets)
- **Knife** (3-4 damage, 2 sockets)

Additionally, every weapon provides a **bonus** to three magical stats - **strength**, **agility** and **vitality**. At first the bonus of every magical stat is zero and can be increased with gems which are inserted into the weapon.

Every gem provides a bonus to all three of the magical stats. There are three different kind of gems:

- **Ruby** (+7 strength, +2 agility, +5 vitality)
- **Emerald** (+1strength, +4 agility, +9 vitality)
- **Amethyst** (+2 strength, +8 agility, +4 vitality)

Every point of strength adds +2 to **min** damage and +3 to **max** damage. Every point of agility adds +1 to **min** damage and +4 to **max** damage. Vitality does not add damage.

Your job is to implement the functionality to read some weapons from the console and optionally to insert or remove gems at different socket indexes until you receive the END command.

Note

If you add gem on top of another, just overwrite it. If you add a gem to an invalid index, nothing happens. If you try to remove a gem from an empty socket or from invalid index, nothing happens. Upon receiving the END command print the weapons in order of their appearance in the format provided below.

Input

Each line consists of three types of commands in which the tokens are separated by ";".

Command types:

- Create;{weapon type};{weapon name}
- Add;{weapon name};{socket index};{gem type}
- Remove;{weapon name};{socket index}
- Print;{weapon name}

Output

Print weapons in the given format:

"{weapon's name}: {min damage}-{max damage} Damage, +{points} Strength, +{points} Agility, +{points} Vitality"

Examples

Input	Output
Create;AXE;Axe of Misfortune Add;Axe of Misfortune;0;RUBY Print;Axe of Misfortune END	Axe of Misfortune: 21-39 Damage, +7 Strength, +2 Agility, +5 Vitality
Create;AXE;Axe of Misfortune Add;Axe of Misfortune;0;RUBY Remove;Axe of Misfortune;0 Print;Axe of Misfortune END	Axe of Misfortune: 5-10 Damage, +0 Strength, +0 Agility, +0 Vitality

Problem 11. *Inferno Infinity - @Override the toString() Method

If you haven't already, override the **toString()** method of the **Weapon** class you have created for the Inferno Infinity problem. Try using the **@Override** annotation.

Note

Pay attention to the actual overriding of the method.

Examples

Input	Output
Create;AXE;Axe of Misfortune Add;Axe of Misfortune;0;RUBY Print;Axe of Misfortune END	Axe of Misfortune: 21-39 Damage, +7 Strength, +2 Agility, +5 Vitality
Create;AXE;Axe of Misfortune Add;Axe of Misfortune;0;RUBY	Axe of Misfortune: 5-10 Damage, +0 Strength, +0 Agility, +0 Vitality

Remove;Axe of Misfortune;0 Print;Axe of Misfortune END	

Problem 12. *Inferno Infinity - @Override the compareTo() Method

Extend your solution a bit further by making your **Weapon** class to be **comparable** to other weapons. Every weapon should have an **item level** which is calculated by the average of the min and max damage, plus every additional stat it has. Consider the Axe of Misfortune imbued with a Ruby from the zero tests:

Axe of Misfortune Item Level: ((21 + 39) / 2) + 7 + 2 + 5 = 44.0

Implement **additional Print** (prints the **greater** weapon with its item level) and Compare command, which compares two weapons by their **non-rounded item level** and prints the **greater** of two weapons' name and its item level displaying **one numbers** after the decimal separator (e.g. 54.40123 == 54.4):

- **Compare**;{weapon name};{weapon name}

Print the greater of the two weapons in the following format:

"{weapon's name}: {min damage}-{max damage} Damage, +{points} Strength, +{points} Agility, +{points} Vitality (Item Level: {items level})"

If both weapons have equal item level, print the **first** one.

Note

Pay attention to the actual overriding of the method.

Examples

Input	Output
Create;AXE;Axe of Misfortune Add;Axe of Misfortune;0;RUBY Create;KNIFE;Thieves Blade Add;Thieves Blade;0;AMETHYST Add;Thieves Blade;1;AMETHYST Compare;Axe of Misfortune;Thieves Blade END	Thieves Blade: 27-80 Damage, +4 Strength, +16 Agility, +8 Vitality **(Item Level: 81.5)**

Problem 13. *Create Custom Class Annotation

Create a custom annotation that can be applied to **classes** and can be accessed at **runtime**. The annotation type elements it should contain are **author**, **revision**, **description** and **reviewers**. Apply the annotation to the **Weapon** class you have created for the Inferno Infinity problem. Provide these exact values:

- **author** = "Pesho"
- **revision** = 3
- **description** = "Used for Java OOP Advanced course - Enumerations and Annotations."
- **reviewers** = "Pesho", "Svetlio"

Implement additional commands for extracting different annotation values:

- **Author** - prints the author of the class
- **Revision** - prints the revision of the class
- **Description** - prints the class description
- **Reviewers** - prints the reviewers of the class

Examples

Input	Output

Author
Revision
Description
Reviewers
END
	`Author: Pesho`
	`Revision: 3`
	Class description: Used for Java OOP Advanced course - Enumerations and Annotations.
	Reviewers: Pesho, Svetlio

Problem 14. **Refactoring - Bonus

Refactor your Inferno Infinity problem code according to all **HQC standards**.

- Think about the **proper naming** of all your variables, methods, classes and interfaces.
- Review all of your methods and make sure they are doing **only one** highly **concrete thing**.
- Review your class **hierarchy** and make sure you have no duplicating code.
- Consider making your classes **less dependent** of each other. If you have the **new** keyword anywhere inside the body of a non-factory or main class, think about how to remove it. Read about **dependency injection.**
- Consider adding **independent** classes for reading **input** and writing **output**.
- Create **repository** class that stores all weapon data.
- Create an **engine**, weapon **creator** and so on. Try using **design patterns** like command and factory.
- Make you classes **highly cohesive** and **loosely coupled**.

Examples

Input	Output

`Create;AXE;Axe of Misfortune`
`Add;Axe of Misfortune;0;RUBY`
`Create;KNIFE;Thieves Blade`
`Add;Thieves Blade;0;AMETHYST`
`Add;Thieves Blade;1;AMETHYST`
`Compare;Axe of Misfortune;Thieves Blade`
Description
END
	`Thieves Blade: 27-80 Damage, +4 Strength, +16 Agility, +8 Vitality (Item Level: 81.5)`
	Class description: Used for Java OOP Advanced course - Enumerations and Annotations.

Problem 15. Bonus - Generate a JavaDoc file

Choose a problem solution of yours and try documenting every class and its members. Use annotations with @Documented meta-tags and learn how to generate a JavaDoc file using the IDE you are using.