# Exercise: Encapsulation and Inheritance

You can check your solutions here: <https://judge.softuni.bg/Contests/473/Defining-Classes-Exercise>

## Part I: Encapsulation

### Class Box

You are given a geometric figure box with parameters **length**, **width** and **height**. Model a class **Box** that that can be instantiated by the same **three parameters**. **Expose** to the outside world only **methods for its surface area, lateral surface area and its volume** (formulas: <http://www.mathwords.com/r/rectangular_parallelepiped.htm>).

The **lateral surface area** of **a three-dimensional object** is the **surface area of the object minus the area of its bases**.

On the first three lines you will get the **length**, **width** and **height**. Print three lines with the surface area, lateral surface area and the volume of the box:

#### Note

Print exactly two digits after every double value's decimal separator (e.g. 10.00). See the built-in PHP function **round()** (<http://php.net/manual/en/function.round.php>).

#### Examples

|  |  |
| --- | --- |
| **Input** | **Output** |
| 2  3  4 | Surface Area – 52.00  Lateral Surface Area – 40.00  Volume – 24.00 |
| 1.3  1  6 | Surface Area - 30.20  Lateral Surface Area - 27.60  Volume - 7.80 |
| 15  23  8 | Surface Area - 1298.00  Lateral Surface Area - 608.00  Volume - 2760.00 |

### Class Box Data Validation

A box’s side should not be zero or a negative number. **Expand** **your** **class** from the **previous** **problem** by adding **data validation by using setters for each parameter given** to the constructor (this makes 3 setters). Make a **private** **setter** that performs data validation internally.

#### Examples

|  |  |
| --- | --- |
| **Input** | **Output** |
| 2  -3  4 | Width cannot be zero or negative. |

### Shopping Spree

Create two classes: class **Person** and class **Product**. Each person should have a **name**, **money** and a **bag** **of products**. Each product should have **name** and **cost**. Name cannot be an empty string. Money cannot be a negative number.

Create a program in which **each command corresponds to a person buying a product**:

* If the person **can afford** a product **add it to his bag**.
* If a person **doesn’t have** **enough** **money**, print an **appropriate** **message** ("[Person name] can't afford [Product name]").

On the first two lines you are given **all people and all products**. Next you will be given **all purchases people made until END** is reached. Print a message when someone makes a purchase. After all purchases print every person in the order of appearance and all products that he has bought also in order of appearance. If nothing is bought, print the name of the person followed by "**Nothing bought**".

In case of invalid input (negative money exception message: "**Money cannot be negative**") or empty name: (empty name exception message "Name cannot be empty") break the program with an appropriate message. See the examples below:

#### Examples

|  |  |
| --- | --- |
| **Input** | **Output** |
| Pesho=11;Gosho=4;  Bread=10;Milk=2;  Pesho Bread  Gosho Milk  Gosho Milk  Pesho Milk  END | Pesho bought Bread  Gosho bought Milk  Gosho bought Milk  Pesho can't afford Milk  Pesho - Bread  Gosho - Milk,Milk |
| Mimi=0;  Kafence=2;  Mimi Kafence  END | Mimi can't afford Kafence  Mimi – Nothing bought |
| Jeko=-3;  Chushki=1;  Jeko Chushki  END | Money cannot be negative |

### \*Mordor’s Cruelty Plan

Gandalf the Gray is a great wizard but he also loves to eat and the food makes him loose his capability of fighting the dark. The Mordor’s orcs have asked you to design them a program which **calculates** **Gandalf’s** **mood**. So they could predict the battles between them and try to beat The Gray Wizard. **When Gandalf is hungry he gets angry** and he could not fight well. Because the orcs have a spy, he has told them the foods that Gandalf is eating and the result on his mood after he has eaten some food. So here is the list:

* **Cram**: 2 points of happiness;
* **Lembas**: 3 points of happiness;
* **Apple**: 1 point of happiness;
* **Melon**: 1 point of happiness;
* **HoneyCake**: 5 points of happiness;
* **Mushrooms**: -10 points of happiness;
* **Everything** **else**: -1 point of happiness;

Gandalf moods are:

* **Angry** - below -5 points of happiness;
* **Sad** - from -5 to 0 points of happiness;
* **Happy** - from 0 to 15 points of happiness;
* **PHP** - when happiness points are more than 15;

The task is simple. Model an **application which is calculating the happiness points**, Gandalf has after eating **all the food passed in the input**. After you have done, print on the first line – **total** **happiness** **points** Gandalf had collected. On the second line – print the **Mood’s** name which is corresponding to the points.

#### Input

The input comes from the console. It will hold single line: all the Gandalf`s foods he has eaten. Comma-separated and case insensitive.

#### Output

Print on the console Gandalf`s happiness points and the **Mood’s** name which is corresponding to the points.

#### Note

Try to implement **factory** **pattern**. You should have two factory classes – **FoodFactory** and **MoodFactory**. And their task is to produce objects (e.g. **FoodFactory**, produces – **Food** and the **MoodFactory** - **Mood**).

Simply put the purpose of the **factory** **design** **pattern** is to **create** **objects** without exposing the object creation to the outer world. Instead of constructors creating the objects it is done by a **factory** **method** in the factory classes.

#### Hint

The factory method returns something like**: return new Food();**

This way when you call the factory method it returns an object of the class it is created to produce. This is why you don’t use a constructor. For more go to: [https://en.wikipedia.org/wiki/Factory\_method\_pattern#PHP](https://en.wikipedia.org/wiki/Factory_method_pattern%23PHP)

#### Examples

|  |  |
| --- | --- |
| **Input** | **Output** |
| Cram,banica,Melon!\_,HonEyCake,!HoneYCake,hoNeyCake\_ | 3  Happy |
| gosho,pesho,meze,Melon,HoneyCake@ | -3  Sad |
| HoneyCake,honeyCake,honeyCake,HoneyCakE,HoneYCake,HonEyCake,HoneyCake,HoneyCake,HoneyCake,HoNeyCake | 50  PHP |

### \*Pizza Calories

A Pizza is made of dough and different toppings. You should model a **class** **Pizza** which should have a **name**, **dough** and **toppings** as properties. **Every type of ingredient should have its own class**. Every ingredient has different properties: the **dough** can be **white** or **wholegrain** and in addition it can be **crispy**, **chewy** or **homemade**. The **toppings** can be of type **meat**, **veggies**, **cheese** or **sauce**. Every ingredient should have a **weight in grams** and a **method for calculating its calories according its type**. Calories per gram are calculated through modifiers. Every ingredient has **2 calories per gram** as a base and a modifier that gives the exact calories. For example, **white** **dough** **has a modifier of 1.5, chewy dough has a modifier of 1.1**, which means that a white chewy dough weighting 100 grams will have (2 \* 100) \* 1.5 \* 1.1 = 330.00 total calories.

**Your job** is to **model the classes** in such a way that they are **properly encapsulated** and to provide a public method for every pizza that **calculates its calories according to the ingredients it has.**

#### Step 1. Create a Dough Class

The base ingredient of a Pizza is the dough. First you need to create a class for it. It has a flour type which can be white or wholegrain. In addition, it has a baking technique which can be crispy, chewy or homemade. Dough should have weight in grams. The calories per gram of dough are calculated depending on the flour type and the baking technique. Every dough has 2 calories per gram as a base and a modifier that gives the exact calories. For example, white dough has a modifier of 1.5, chewy dough has a modifier of 1.1, which means that white chewy dough weighting 100 grams will have (2 \* 100) \* 1.5 \* 1.1 = 330.00 total calories. You are given the modifiers below:

Modifiers:

* White – 1.5;
* Wholegrain – 1.0;
* Crispy – 0.9;
* Chewy – 1.1;
* Homemade – 1.0;

Everything that the class should expose is a getter for the calories per gram. Your task is to create the class with a proper constructor, properties, getters and setters. Make sure you use the proper access modifiers.

#### Step 2. Validate Data for the Dough Class

Change the internal logic of the Dough class by adding a data validation in the setters.

Make sure that if invalid flour type or an invalid baking technique is given a proper exception is thrown with the message "Invalid type of dough.".

The allowed weight of dough is in the range [1..200] grams. If it is outside of this range throw an exception with the message "Dough weight should be in the range [1..200]."

##### Hint

An exception is actually an object of **class** **Exception** thrown this way:

**"throw new Exception('Division by zero.')**;**"**

If an exception is not caught, then it stops programs execution and displays the error. You can catch an exception (then the program will not stop) and do something meaningful by using **try** and **catch**. See for more: <http://php.net/manual/en/language.exceptions.php>

##### **Exception** **Messages**

* "Invalid type of dough."
* "Dough weight should be in the range [1..200]."

##### Examples

|  |  |
| --- | --- |
| **Input** | **Output** |
| Dough White Chewy 100  END | 330.00 |
| Dough Tip500 Chewy 100  END | Invalid type of dough. |
| Dough White Chewy 240  END | Dough weight should be in the range [1..200]. |

#### Step 3. Create a Topping Class

Next you need to create a **Topping** **class**. It can be of four different types – **meat**, **veggies**, **cheese** or a **sauce**. A topping has a **weight in grams**. The calories per gram of a topping are calculated depending on its type. **The base calories per gram are 2**. Every different type of topping has a modifier. For example**, meat has a modifier of 1.5**, so a **meat topping will have 3 calories per gram (2 \* 1.5)**. Everything that the class should expose is a **getter for calories per gram**. You are given the modifiers below:

Modifiers:

* Meat – 1.2;
* Veggies – 0.8;
* Cheese – 1.1;
* Sauce – 0.9;

Your task is to create the class with a proper **constructor**, **properties**, **getters** and **setters**. Make sure you use the proper access modifiers.

#### Step 4. Validate Data for the Topping Class

Change the internal logic of the Topping class by **adding a data validation in the setter.**

Make sure the topping is one of the provided types, otherwise **throw a proper exception with the message**

**"Cannot place [name of invalid argument] on top of your pizza."**

The **allowed** **weight** of a topping is in the **range [1..50] grams**. If it is outside of this range throw an exception with the message "[Topping type name] weight should be in the range [1..50].".

##### Exception Messages

* "Cannot place [name of invalid argument] on top of your pizza."
* "[Topping type name] weight should be in the range [1..50]."

##### Examples

|  |  |
| --- | --- |
| **Input** | **Output** |
| Dough White Chewy 100  Topping meat 30  END | 330.00  72.00 |
| Dough White chewy 100  Topping Krenvirshi 500  END | 330.00  Cannot place Krenvirshi on top of your pizza. |
| Dough White Chewy 100  Topping Meat 500  END | 330.00  Meat weight should be in the range [1..50]. |

#### Step 5. Create a Pizza Class!

A Pizza should have a name, some toppings and dough. Make use of the two classes you’ve made earlier. Also a pizza should have public getters for its name, number of toppings and the total calories. The **total** **calories** are **calculated by summing the calories of all the ingredients a pizza has**. Create the class using a proper constructor, expose a method for adding a topping, a public setter for the dough and a getter method for the total calories.

The input for a pizza consists of several lines. On the first line is the **pizza** **name** and the **number of toppings it has**. On the second line you will get input for the **dough**. On the next lines, you will receive every topping the pizza has. **The number of lines for the toppings** will correspond to the number of toppings declared on the first line.

If creation of the pizza was **successful** print on a single line the name of the pizza and the **total** **calories** it has.

#### Step 6. Validate Data for the Pizza Class

The name of the pizza should not be an empty string. Also it should not be longer than 15 symbols. If it doesn’t fit this throw and exception with the message "Pizza name should be between 1 and 15 symbols."

The number of toppings should be in range [0...10]. If not, throw an exception with the message "Number of toppings should be in range [0..10]."

Your task is to print the name of the pizza and the total calories it has according to the examples below.

##### Examples

|  |  |
| --- | --- |
| **Input** | **Output** |
| Pizza Meatless 2  Dough Wholegrain Crispy 100  Topping Veggies 50  Topping Cheese 50  END | Meatless – 370.00 Calories. |
| Pizza Meatfull 5  Dough White cheWy 200  Topping Meat 50  Topping Cheese 50  Topping meat 20  Topping sauce 10  Topping Meat 30  END | Meatfull – 1028.00 Calories. |
| Pizza Bulgarian 11  Dough Tip500 Balgarsko 100  Topping Sirene 50  Topping Cheese 50  Topping Krenvirsh 20  Topping Next1 10  Topping Next2 12  Topping Next3 41  Topping Next4 33  Topping Next5 45  Topping Next6 18  Topping Next7 25  Topping Next8 15  END | Number of toppings should be in range [0..10]. |
| Pizza Bulgarian 4  Dough Tip500 Balgarsko 100  Topping Sirene 50  Topping Cheese 50  Topping Krenvirsh 20  Topping Meat 10  END | Invalid type of dough. |
| Pizza Bulgarian 4  Dough White Chewy 100  Topping Sirene 50  Topping Cheese 50  Topping Krenvirsh 20  Topping Meat 10  END | Cannot place Sirene on top of your pizza. |

## Part II: Inheritance

### Person

You are asked to model an application for storing data about people. You should be able to have a person and a child. The child is derived of the person. Your task is to model the application. The only constraints are:

* **Person** – represents the base class by which all others are implemented
  + People should **not** be able to have **negative** **age**
* **Child** - represents a class which is derived by the class **Person**.
  + Children should **not** be able to have age **greater** **than** **15**

#### Note

Your classes' names **MUST** be the same as the names shown above!!!

Create a new empty class and name it **Person**. Every person has a name, and age.

|  |
| --- |
| **Sample Code** |
| class Person {    *// 1. Add the Properties  // 2. Add the Constructor  // 3. Add the Methods* } |

#### Step 1. Define the Properties of a Person

Define the **name** and **age** properties of a Person. Ensure that they can only be **changed by the class itself or its descendants** (pick the most appropriate access modifier).

#### Step 2. Define the Constructor

Define a constructor that accepts **name** and **age** arguments.

|  |
| --- |
| **Sample Code** |
| **function** \_\_construct(string $name, int $age){  $**this**->setName($name);  $**this**->setAge($age);  } |

#### Step 3. Perform Validations

After you have created the **properties** (e.g. **name** and **age**). Next step is to **perform** **validations** for each one. The **getter should return the corresponding property’s** **value** and the **setter** **should** **validate** the input data before setting it. Do this for each property.

|  |
| --- |
| **Sample Code** |
| **protected function** setAge(int $age) {  **if** ($age < 1) {  **throw new** Exception(**"Age must be positive!"**);  }   *//* ***TODO: Set the age*** } |

##### Constraints

* If the age of a person is negative – exception’s message is:
  + **"Age must be positive!"**
* If the age of a child is bigger than 15 – exception’s message is:
  + **"Child’s age must be less than 16!"**
* If the name of a child or a person is no longer than 3 symbols – exception’s message is:
  + **"Name’s length should not be less than 3 symbols!"**

#### Step 5. Create a Child

Create a **Child** class that **inherits** **Person** and has the same constructor definition. However, do not copy the code from the Person class - **reuse the Person class’s constructor.**

### Book Shop

You are working in a library. And you are pissed off by writing descriptions for books by hand, so you wanted to use the computer to make them faster. So the task is simple. Your program should have two classes – one for the ordinary books – **Book**, and another for the special ones – **GoldenEditionBook**. So let’s get started! We need two classes:

* **Book** - represents a book that holds **title**, **author** and **price**. A book should offer **information** about itself in the format shown in the output below.
* **GoldenEditionBook** - represents a special book holds the same properties as any **Book**, but its **price** is always **30% higher**.

#### Input

The input comes from the console. It will hold four lines. On the first you will be given the name, on the second the title and third will be the standard price. Fourth line will hold the type of book - **GOLD** or **STANDARD**

#### Output

On the first line print OK, if all inputs are in the correct format. If not print every error message on a single line.

If input is valid print a second line with the price of the book.

#### Constraints

* If the author’s second name is starting with a digit– exception’s message is:
  + **"Author not valid!"**
* If the title’s length is less than 3 symbols – exception’s message is:
  + **"Title not valid!"**
* If the price is zero or it is negative – exception’s message is:
  + **"Price not valid!"**
* If book type is invalid – exception’s message is:
  + **"Type is not valid!"**

#### Example

|  |  |
| --- | --- |
| **Input** | **Output** |
| Ivo 4ndonov  Under Cover  9999999999999999999  STANDARD | Author not valid! |
| Georgi Petrov  Summer Moon  22  STANDARD | OK  22 |
| Vasil Ivanov  UML Basics  10  GOLD | OK  13 |

#### Step 1. Create a Book Class

Create a new class and name it **Book**.

|  |
| --- |
| **Sample Code** |
| **public class** Book {   *// 1. Add the Properties  // 2. Add the Constructor  // 3. Add the Methods* } |

#### Step 2. Define the Properties of a Book

Define the **title**, **author** and **price** properties of a Book. Ensure that they can only be **changed by the class itself** **or its descendants** (pick the most appropriate access modifier).

#### Step 3. Define a Constructor

Define a constructor that accepts **author**, **title** and **price** arguments.

#### Step 4. Perform Validations

Create the **properties** (**price**, **title** and **author**) and **perform** **validations** for each one. **The getter should return the corresponding property** and the **setter should validate** the input data before setting it. Do this for every property.

#### Step 5. Define magic method \_\_toString

The \_\_toString() method allows a class to decide how it will react when it is treated like a string. For example, what **echo $obj**; will print.

And voila! If everything is correct, we can now create **Book** **objects** and display information about them.

#### Step 6. Create a GoldenEditionBook

Create a **GoldenEditionBook** class that inherits **Book** and has the same constructor definition. However, do not copy the code from the Book class - **reuse the Book class constructor.**

There is **no need** to rewrite the **price**, **title** and **author** properties since **GoldenEditionBook** inherits **Book** and by default has them.

#### Step 7. Override the Price Method

Golden edition books should return a **30%** higher **price** than the original price. In order for the getter to return a different value, we need to override the method.

Back to the **GoldenEditionBook** class, let's override the getter method.

|  |
| --- |
| **Sample Code** |
| **public function** getPrice() {  **return parent::**getPrice() \* 1.3; } |

### Mankind

Your task is to model an application. It is very simple. The mandatory models of our application are 3: Human, Worker and Student.

The parent class – Human should have **first** **name** and **last** **name**. Every student has a **faculty** **number**. Every worker has a **week** **salary** and **work** **hours** **per** **day**. It should be able to calculate the money he earns by hour. You can see the constraints below.

#### Input

On the first input line you will be given info about a single student - first name, last name and faculty number.

On the second input line you will be given info about a single worker - first name, last name, salary and working hours.

#### Output

You should first print the info about the student following a single blank line and the info about the worker in the given formats:

* Print the student info in the following format:
  + **First Name: {student's first name}**
  + **Last Name: {student's last name}**
  + **Faculty number: {student's faculty number}**
* Print the worker info in the following format:
  + **First Name: {worker's first name}**
  + **Last Name: {worker's second name}**
  + **Week Salary: {worker's salary}**
  + **Hours per day: {worker's working hours}**
  + **Salary per hour: {worker's salary per hour}**

Print exactly two digits after every double value's decimal separator (e.g. 10.00)

#### Constraints

|  |  |  |
| --- | --- | --- |
| **Parameter** | **Constraint** | **Exception** **Message** |
| Human first name | Should start with a capital letter | "Expected upper case letter!Argument: firstName" |
| Human first name | Should be at least 4 symbols | "Expected length at least 4 symbols!Argument: firstName" |
| Human last name | Should start with a capital letter | "Expected upper case letter!Argument: lastName" |
| Human last name | Should be at least 3 symbols | "Expected length at least 3 symbols!Argument: lastName " |
| Faculty number | Length should be in range [5..10] digits | "Invalid faculty number!" |
| Worker last name | Should be more than 3 symbols | "Expected length more than 3 symbols!Argument: lastName" |
| Week salary | Should be more than 10 | "Expected value mismatch!Argument: weekSalary" |
| Working hours | Should be in the range [1..12] | "Expected value mismatch!Argument: workHoursPerDay" |

#### Example

|  |  |
| --- | --- |
| **Input** | **Output** |
| Ivan Ivanov 08  Pesho Kirov 1590 10 | Invalid faculty number! |
| Stefo Mk321 0812111  Ivcho Ivancov 1590 10 | First Name: Stefo  Last Name: Mk321  Faculty number: 0812111  First Name: Ivcho  Last Name: Ivancov  Week Salary: 1590.00  Hours per day: 10.00  Salary per hour: 22.71 |

### Online Radio Database

Create an online radio station database. It should keep information about all added songs. On the first line you are going to get the number of songs you are going to try adding. On the next lines you will get the songs to be added in the format **<artist name>;<song name>;<minutes:seconds>**. To be valid, every song should have an artist name, a song name and length.

Design a **custom** **exception** **hierarchy** for invalid songs:

* InvalidSongException
  + InvalidArtistNameException
  + InvalidSongNameException
  + InvalidSongLengthException
    - InvalidSongMinutesException
    - InvalidSongSecondsException

#### Validation

* Artist name should be between 3 and 20 symbols.
* Song name should be between 3 and 30 symbols.
* Song length should be between 0 second and 14 minutes and 59 seconds.
* Song minutes should be between 0 and 14.
* Song seconds should be between 0 and 59.

#### Exception Messages

|  |  |
| --- | --- |
| **Exception** | **Message** |
| InvalidSongException | "Invalid song." |
| InvalidArtistNameException | "Artist name should be between 3 and 20 symbols." |
| InvalidSongNameException | "Song name should be between 3 and 30 symbols." |
| InvalidSongLengthException | "Invalid song length." |
| InvalidSongMinutesException | "Song minutes should be between 0 and 14." |
| InvalidSongSecondsException | "Song seconds should be between 0 and 59." |

Note: Check validity in the order artist name -> song name -> song length

#### Output

If the song is added, print "**Song added.**". If you **can’t add a song**, print an **appropriate exception message**. On the last two lines print the **number of songs added** and **the total length of the playlist** in format **{Playlist length: 0h 07m 47s}.**

#### Examples

|  |  |
| --- | --- |
| **Exception** | **Message** |
| 3  ABBA;Mamma Mia;3:35  Nasko Mentata;Shopskata salata;4:123  Nasko Mentata;Shopskata salata;4:12 | Song added.  Song seconds should be between 0 and 59.  Song added.  Songs added: 2  Playlist length: 0h 07m 47s |
| 5  Nasko Mentata;Shopskata salata;14:59  Nasko Mentata;Shopskata salata;14:59  Nasko Mentata;Shopskata salata;14:59  Nasko Mentata;Shopskata salata;14:59  Nasko Mentata;Shopskata salata;0:5 | Song added.  Song added.  Song added.  Song added.  Song added.  Songs added: 5  Playlist length: 1h 00m 01s |

### \*Animals

Create a hierarchy of **Animals**. Your task is simple: there should be a base class which all others derive from. Your program should have 3 different animals – **Dog**, **Frog** and **Cat**. Let’s go deeper in the hierarchy and create two additional classes – **Kitten** and **Tomcat. Kittens are female and Tomcats are male!** We are ready now, but the task is not complete. Along with the animals, there should be a class which classifies its derived classes as sound producible. You may guess that all animals are sound producible. The only one mandatory functionality of all sound producible objects is to **produceSound()**. For instance, the dog should bark.

Your task is to model the hierarchy and test its functionality. Create an animal of all kinds and make them produce sound.

On the console, you will be given some lines of code. Each two lines of code, represents **animals and their names, age** and **gender**. On the first line there will be the kind of animal, you should **instantiate**. And on the next line, you will be given the **name**, the age and the gender. Stop the process of gathering input, when the command "**Beast**!" is given.

#### Output

* On the console, print for each animal you’ve instantiated, its info on two lines. On the first line, print:  
   {**Kind** **of** **animal**} {**name**} {**age**} {**gender**}
* On the second line, print: {**produceSound()**}

#### Constraints

* Each Animal should have **name**, **age** and **gender**
* All properties’ values should not be blank (e.g. name, age and so on…)
* If you enter invalid input for one of the properties’ values, throw exception with message: "**Invalid input!**"
* Each animal should have a functionality to **produceSound()**
* Here is example of what each kind of animal should produce when, **produceSound()** is called
  + **Dog: "BauBau"**
  + **Cat: "MiauMiau"**
  + **Frog: "Frogggg"**
  + **Kittens: "Miau"**
  + **Tomcat: "Give me one million b\*\*\*h"**
  + **Message from the Animal class: "Not implemented!"**

#### Examples

|  |  |
| --- | --- |
| **Input** | **Output** |
| Cat  Macka 12 Female  Dog  Sharo 132 Male  Beast! | Cat Macka 12 Female  MiauMiau  Dog Sharo 132 Male  BauBau |
| Frog  Sashky 12 Male  Beast! | Frog Sashky 12 Male  Frogggg |
| Frog  Sashky -2 Male  Beast! | Invalid input! |

### \*\*\*Family Tree

You want to build your family tree, so you went to ask your grandmother, sadly your grandmother keeps remembering information about your predecessors in pieces, so it falls to you to group the information and build the family tree.

On the first line of the input you will receive either a name or a birthdate in the format **"<FirstName> <LastName>**" or "**day/month/year**" – your task is **to find the person’s information in the family tree**. On the next lines until the command "**End**" is received you will receive information about your predecessors that you will use to build the family tree.

The information will be in one of the following formats:

* **"FirstName LastName - FirstName LastName"**
* **"FirstName LastName - day/month/year"**
* **"day/month/year - FirstName LastName"**
* **"day/month/year - day/month/year"**
* **"FirstName LastName day/month/year"**

The first 4 formats reveal a family tie **– the person on the left is parent to the person** **on the right** (as you can see the format does not need to contain names, for example the 4th format means the person born on the left date is parent to the person born on the right date). The last format ties different information together – i.e. **the person with that name was born on that date**. **Names** and **birthdates** are **unique** – there won’t be 2 people with the same name or birthdate, there will **always** be enough entries to construct the family tree (all people’s names and birthdates are known and they have at least one connection to another person in the tree).

After the command "**End**" is received you should print all information about the person whose name or birthdate you received on the first line – his **name**, **birthday**, **parents** and **children** (check the examples for the format). The people in the parents and childrens lists should be ordered by their first appearance in the input (regardless if they appeared as a birthdate or a name, for example in the first input Stamat is before Penka because he first appeared in the second line, while she appears in the third.).

#### Examples

|  |  |
| --- | --- |
| **Input** | **Output** |
| Pesho Peshev  11/11/1951 - 23/05/1980  Penka Pesheva - 23/05/1980  Penka Pesheva 09/02/1953  Pesho Peshev - Gancho Peshev  Gancho Peshev 01/01/2005  Stamat Peshev 11/11/1951  Pesho Peshev 23/05/1980  End | Pesho Peshev 23/05/1980  Parents:  Stamat Peshev 11/11/1951  Penka Pesheva 09/02/1953  Children:  Gancho Peshev 01/01/2005 |
| 13/12/1993  25/03/1934 - 04/04/1961  Poncho Tonchev 25/03/1934  04/04/1961 - Moncho Tonchev  Toncho Tonchev - Lomcho Tonchev  Moncho Tonchev 13/12/1993  Lomcho Tonchev 07/07/1995  Toncho Tonchev 04/04/1961  End | Moncho Tonchev 13/12/1993  Parents:  Toncho Tonchev 04/04/1961  Children: |

### \*\*Cat Lady

Ginka has many cats in her house of various breeds, since some breeds have specific characteristics, Ginka needs some way to catalogue the cats, help her by creating a **class** **hierarchy** with all her breeds of cats so that she can easily check on their characteristics. Ginka has 3 specific breeds of cats "**Siamese**", "**Cymric**" and the very famous bulgarian breed "**Street** **Extraordinaire**", each breed has a specific characteristic about which information should be kept. For the Siamese cats their ear size should be kept, for Cymric cats - the length of their fur in milimeters and for the Street Extraordinaire the decibels of their meowing during the night.

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

* **"Siamese <name> <earSize>"**
* **"Cymric <name> <furLength>"**
* **"StreetExtraordinaire <name> <decibelsOfMeows>"**

On the last line after the "End" command you will receive the name of a cat, you should print that cat’s information in the same format in which you received it.

#### Example

|  |  |
| --- | --- |
| **Input** | **Output** |
| StreetExtraordinaire Maca 85  Siamese Sim 4  Cymric Tom 28  End  Tom | Cymric Tom 28 |
| StreetExtraordinaire Koti 80  StreetExtraordinaire Maca 100  Cymric Tim 31  End  Maca | StreetExtraordinaire Maca 100 |

#### Hint

Use class inheritance to represent the cat hierarchy and define **\_\_toString** methods of concrete breeds to allow for easy printing of the cat, regardless what breed it is.