# Lab: Interfaces and Abstraction

Problems for exercises and homework for the ["Java OOP" course @ SoftUni.](https://softuni.bg/trainings/2245/java-oop-february-2019#lesson-10496)

You can check your solutions here: <https://judge.softuni.bg/Contests/1581/Interfaces-and-Abstraction-Lab> .

## Shapes Drawing

Build hierarchy of **interfaces** and **classes**:

|  |
| --- |
| <<Drawable>> |
| **+draw()** |

|  |
| --- |
| **Circle** |
| **-radius: Integer** |

|  |
| --- |
| Rectangle |
| **-width: Integer**  **-height: Integer** |

You should be able to use the class like this:

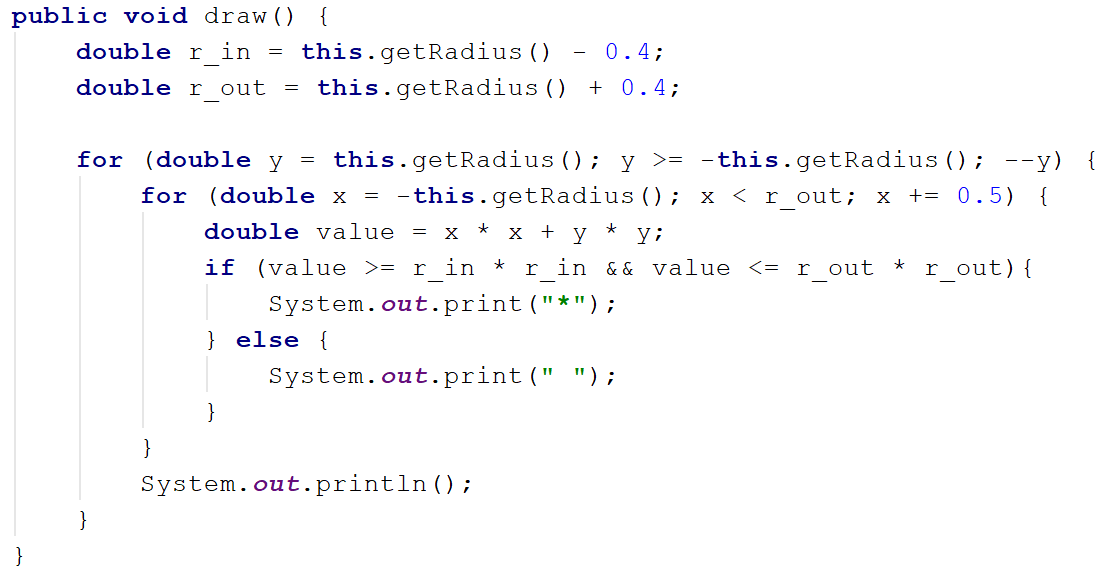
|  |
| --- |
| Main.java |
| **public static void** main(String[] args) {  Scanner scanner = **new** Scanner(System.***in***);  Queue<Integer> queue = **new** ArrayDeque<>();  **for** (**int** i = 0; i < 5; i++) {  queue.add(Integer.*parseInt*(scanner.nextLine()));  }   Drawable circle = **new** Circle(queue.poll());  Drawable rect = **new** Rectangle(queue.poll(), queue.poll());   circle.draw();  rect.draw(); } |

### Examples

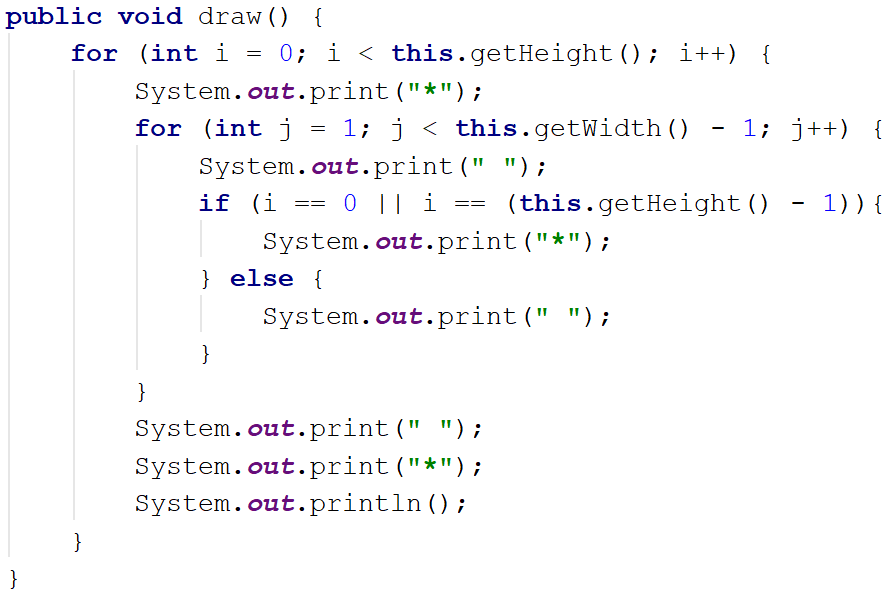
|  |  |
| --- | --- |
| **Input** | **Output** |
| 4  5  4 | \*\*\*\*\*\*\*  \*\*\* \*\*\*  \*\* \*\*  \*\* \*\*  \* \*  \*\* \*\*  \*\* \*\*  \*\*\* \*\*\*  \*\*\*\*\*\*\*  \* \* \* \* \*  \* \*  \* \*  \* \* \* \* \* |

### Solution

For **circle** drawing you can use this algorithm:



For **rectangle** drawing algorithm will be:



## Car Shop

Build hierarchy from **classes** and **interfaces** for this UML diagram

|  |
| --- |
| <<inteface>>  <<Car>> |
| **+TIRES: Integer** |
| **+getModel(): String**  **+getColor(): String**  **+getHorsePower(): Integer** |

|  |
| --- |
| **<<Serializable>>** |

|  |
| --- |
| Seat |
| **-countryProduced: String** |
| **+toString(): String** |

Your hierarchy have to be used with this code

|  |
| --- |
| Main.java |
| **public static void** main(String[] args) {  Car seat = **new** Seat(**"Leon"**, **"gray"**, 110, **"Spain"**);   System.***out***.println(String.*format*(  **"%s is %s color and have %s horse power"**,  seat.getModel(),  seat.getColor(),  seat.getHorsePower()));  System.***out***.println(seat.toString()); } |

### Examples

|  |  |
| --- | --- |
| **Input** | **Output** |
|  | Leon is gray and have 110 horrse powers  This is Leon produced in Spain and have 4 tires |

### Solution



**Note:** consider using the wrapper classes in the **Seat** constructor**.**

## Car Shop Extend

|  |
| --- |
| **<<Car>>** |

Extend previous problem:

|  |
| --- |
| <<Rentable>> |
| **+getMinRentDay(): Integer**  **+getPricePerDay(): Double** |

|  |
| --- |
| <<Sellable>> |
| **+getPrice(): Double** |

|  |
| --- |
| Audi |
| **-minRentDay: Integer**  **-pricePerDay: Double**  **-countryProduced: String** |
| **+toString(): String** |

|  |
| --- |
| Seat |
| **-price: Double**  **-countryProduced: String** |
| **+toString(): String** |

Your hierarchy have to be used with this code

|  |
| --- |
| Main.java |
| **public static void** main(String[] args) {  Sellable seat = **new** Seat(**"Leon"**, **"Gray"**, 110, **"Spain"**, 11111.1);  Rentable audi = **new** Audi(**"**A4**"**, **"Gray"**, 110, **"Germany"**, 3, 99.9);   *printCarInfo*(seat);  *printCarInfo*(audi); }  **private static void** printCarInfo(Car car) {  System.***out***.println(String.*format*(  **"%s is %s color and have %s horse power"**,  car.getModel(),  car.getColor(),  car.getHorsePower()));  System.***out***.println(car.toString()); } |

### Examples

|  |  |
| --- | --- |
| **Input** | **Output** |
|  | Leon is Gray color and have 110 horse power  This is Leon produced in Spain and have 4 tires  Leon sells for 11111,100000  A4 is Gray color and have 110 horse power  This is A4 produced in Germany and have 4 tires  Minimum rental period of 3 days. Price per day 99,900000 |

## Say Hello

Build hierarchy from classes and interfaces for this UML diagram

|  |
| --- |
| <<Person>> |
| **+getName(): String**  **+sayHello(): void** |

|  |
| --- |
| Chinese |
| **-name: String** |
| **+sayHello(): String** |

|  |
| --- |
| Bulgarian |
| **-name: String** |
| **+sayHello(): String** |

|  |
| --- |
| European |
| **-name: String** |
|  |

Your hierarchy have to be used with this code

|  |
| --- |
| Main.java |
| **public static void** main(String[] args) {  List<Person> persons = **new** ArrayList<>();  persons.add(**new** Bulgarian(**"Pesho"**));  persons.add(**new** European(**"Pesho"**));  persons.add(**new** Chinese(**"Pesho"**));   **for** (Person person : persons) {  *print*(person);  } }  **private static void** print(Person person) {  System.***out***.println(person.sayHello()); } |

### Examples

|  |  |
| --- | --- |
| **Input** | **Output** |
|  | Здравей  Hello  Djydjybydjy |

## Say Hello Extend

Build hierarchy from classes and interfaces for this UML diagram

|  |
| --- |
| Bulgarian |
| **+sayHello(): String** |

|  |
| --- |
| <<Person>> |
| **+getName(): String**  **+sayHello(): String** |

|  |
| --- |
| *BasePerson* |
| **-name: String** |
| **+getName(): String**  **-setName(): void** |

|  |
| --- |
| European |
| +sayHello(): String |

|  |
| --- |
| Chinese |
| **+sayHello(): String** |

Your hierarchy have to be used with this code

|  |
| --- |
| Main.java |
| **public static void** main(String[] args) {  List<Person> persons = **new** ArrayList<>();  persons.add(**new** Bulgarian(**"Pesho"**));  persons.add(**new** European(**"Pesho"**));  persons.add(**new** Chinese(**"Pesho"**));   **for** (Person person : persons) {  *print*(person);  } }  **private static void** print(Person person) {  System.***out***.println(person.sayHello()); } |

### Examples

|  |  |
| --- | --- |
| **Input** | **Output** |
|  | Здравей  Hello  Djydjybydjy |

## Ferrari

Model an application which contains a **class Ferrari** and an **interface**. Your task is simple, you have a **car - Ferrari**, its model is **"488-Spider"** and it has a **driver**. Your Ferrari should have functionality to **use brakes** and **push the gas pedal**. When the **brakes** are pushed down **print "Brakes!"**, and when the **gas pedal** is pushed down - **"Zadu6avam sA!"**. As you may have guessed this functionality is typical for all cars, so you should **implement an interface** to describe it.

|  |  |
| --- | --- |
| **<<Interface>>**  **Car** | |
|  |  |
| + | brakes() : String |
| + | gas() : String |

Your task is to **create a Ferrari** and **set the driver's name** to the passed one in the input. After that, print the info. Take a look at the Examples to understand the task better.

|  |  |
| --- | --- |
| **Ferrari** | |
| - | driverName: String |
| - | model: String |
| + | Ferrari (String) |
| + | brakes() : String |
| + | gas() : String |

### Input

On the **single input line**, you will be given the **driver's name**.

### Output

On the **single output line**, print the model, the messages from the brakes and gas pedal methods and the driver's name. In the following format:

<**model**>/<**brakes**>/<**gas** **pedal**>/<**driver's** **name**>

### Constraints

The input will always be valid, no need to check it explicitly! The Driver's name may contain any ASCII characters.

### Example

|  |  |
| --- | --- |
| **Input** | **Output** |
| Bat Giorgi | 488-Spider/Brakes!/Zadu6avam sA!/Bat Giorgi |
| Dinko | 488-Spider/Brakes!/Zadu6avam sA!/Dinko |

### Note

To check your solution, copy the code below and paste it to the bottom of the code in your main method.

|  |
| --- |
| **Reflection** |
| String ferrariName = Ferrari.**class**.getSimpleName();  String carInterface = ***Car***.**class**.getSimpleName();  **boolean** isCreated = ***Car***.**class**.isInterface();  **if** (!isCreated) {  **throw new** IllegalClassFormatException(**"No interface created!"**);  } |

## 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 from the console an **unknown** amount of lines until the command “**End**” is received, on each line there will be the 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 (the order of printing doesn’t matter).

|  |  |
| --- | --- |
| **Robot** | |
| - | id: String |
| - | model: String |
| + | Robot(Stirng, String) |
| + | getId() : String |
| + | getModel() : String |

|  |  |
| --- | --- |
| **Citizen** | |
| - | name: String |
| - | age: int |
| - | id: String |
| + | Citizen(Stirng, int, String) |
| + | getName() : String |
| + | getAge() : int |
| + | getId() : String |

|  |  |
| --- | --- |
| **<<Interface>>**  **Identifiable** | |
|  |  |
|  |  |
| + | getId() : String |

### Examples

|  |  |
| --- | --- |
| **Input** | **Output** |
| Pesho 22 9010101122  MK-13 558833251  MK-12 33283122  End  122 | 9010101122  33283122 |
| Toncho 31 7801211340  Penka 29 8007181534  IV-228 999999  Stamat 54 3401018380  KKK-666 80808080  End  340 | 7801211340 |

## \*MooD 3

You are an owner of the most epic video game of the world - **3 MooD**. Your employees have gone on summer vacation. But there is a problem in the application and you are on your own. So the problem is how to store the information for the players. The best approach to you, seems to be, storing them in **GameObjects**.

In your game, there are two types of characters - **Demon** and **Archangel**. All characters in the game have:

* **username**
* **hashedPassword**
* **level**
* **special** **points**.

The **main difference** between the Demon and the Archangel is that the **Demon has an energy** (as special points) and the **Archangel has a mana** (as special points). Your task is to model the application.

When you receive the username and the character type, you should generate the hashed password by the formulas below:

* For a **Demon**: **username length \* 217**
* For an **Archangel**: **(username’s characters in reversed order) + (username’s characters' length \* 21)**

Your task is to print the info as it is written in the Output.

|  |  |
| --- | --- |
| **<<Interface>>**  **GameObject** | |
|  |  |
|  |  |
| + | getUsername() : String |
| + | getCharacterType() : String |
| + | getSpecialPoints() : Number |
| + | getLevel() : int |
| + | getHashedPassword() : T |
| + | setHashedPassword(T):void |

|  |  |
| --- | --- |
| **<<Abstract>>**  **Character<P>** | |
| - | username: String |
| - | characterType: String |
| - | level: int |
| - | specalPoints: Number |
| - | hashedPassword: P |
| + | Citizen(Stirng, String, int, Number) |
| + | getUsername() : String |
| + | getCharacterType() : String |
| + | getSpecialPoints() : Number |
| + | getLevel () : int |
| + | getHashedPassword() : P |
| + | setHashedPassword(P):void |
| + | toString(): String |

|  |  |
| --- | --- |
| **Archangel** | |
| - | characterType: String = “Archangel” |
| + | Archangel(String, int , Integer) |
| + | toString() : String |

|  |  |
| --- | --- |
| **Demon** | |
| - | characterType: String = “Demon” |
| + | Demon( String , int , Double) |
| + | toString() : String |

### Input

The input consists of **single line**. First, you will get the username of a player. The second parameter is its character type. The next two parameters are his mana / energy points and his level. Format:

<**username**> | <**character type**> | <**special points**> | <**level**>

### Output

Print the info on two lines, for a single entry in the database (player) in the format:

<”**username”>** | **<”hashed password”>** -> <**character type**>

<**special points \* level**>

### Constraints

* **Username** – alphabetical letters (**Latin**), no more than 10 characters and you do not need to check it explicitly.
* **Character** **type** – String, Demon or Archangel, no need to check it explicitly.
* **Special points (Mana)** – a valid Integer, no need to check it explicitly.
* **Special points (Energy)** – a valid Double, no need to check it explicitly.
* **Level** – a valid Integer, no need to check it explicitly.

### Example

|  |  |
| --- | --- |
| **Input** | **Output** |
| **KoHaH** | **Demon** | **100.0** | **100** | **""KoHaH"" | "1519" -> Demon**  **10000.0** |
| **Akasha | Archangel | 5 | 100** | **""Akasha"" | ""ahsakA"168" -> Archangel**  **500** |

### Note

Implement **interface**, holding the **main functionalities of** **all characters**. Create an **abstract class** to hold all the same characteristics of the characters. If you need to declare a character object, be sure to declare it of type character's interface to the left side and the specific implementation to the right side of the declaration. You should not override the setter for the hashedPassword and instead, use generics to pass them the type for the password and the special points.