# Java OOP Retake Exam – Robot Service

## Overview

In this exam, you need to build a robot service project, which has support for **robots**, **procedures** for storing procedures of robots,and a **garage** for storing robots in the robot service. The project will consist of **model classes** and a **controller class**, which manages the **interaction** between the **robots**, **procedures** and **garage**.

## Setup

* Upload **only the** robotServiceproject in every problem **except** **Unit Tests**
* **Do not modify the interfaces or their packages**
* Use **strong cohesion** and **loose coupling**
* **Use inheritance and the provided interfaces wherever possible**.
  + This includes **constructors**, **method parameters** and **return types**
* **Do not** violate your **interface** **implementations** by adding **more public methods** in the concrete class than the interface has defined
* Make sure you have **no public fields** anywhere

## Task 1: Structure (50 points)

For this task’s evaluation logic in the methods isn’t included.

You are given interfaces, and you have to implement their functionality in the **correct classes**.

There are **3** types of entities in the application: **Robot, Procedure, Garage**.

### Robots

The BaseRobot is a **base class** for any **robot** and it **should not be able to be instantiated**.

#### Data

* name – **String**
* happiness – **int** 
  + can't be **less** **than** **0** or **more than** **100**. In these cases throw **IllegalArgumentException** with message "Invalid happiness."
* **energy – int** 
  + can't be **less than** **0** or **more than** **100**. In these cases throw **IllegalArgumentException** with message "Invalid energy."
* **procedureTime – int**
* **owner – String** 
  + by default: **"**Service**"**
* **isBought – boolean** 
  + by default: **false**
* **isRepaired – boolean** 
  + by default: **false**

#### Constructor

A **robot** should take the following values upon initialization:

(String name, int energy, int happiness, int procedureTime)

#### Child Classes

There are several concrete types of **robots**:

* Housekeeper
* Cleaner

**Override** toString() **method in the format:**

**"**Robot type: {robotType} - {robotName} - Happiness: {happiness} - Energy: {energy}**"**

**Note: There is a space in the beginning of the sentence!**

### Procedures

The BaseProcedure is a **base class** for any **procedures** and it **should not be able to be instantiated**.

#### Data

* robots – collection of **Robots** accessible only by the child classes. Collection should contains all robots which has **visited specific procedure**.

#### Constructor

A **procedure** should not take any values upon initialization

#### Behavior

##### String history()

Returns a String with information about **current procedure type** and its robots. The returned string must be in the following format:

**"**{procedureType}**"**

**"** Robot type: {robotType} - {robotName} - Happiness: {happiness} - Energy: {energy}**"**

**void doService(Robot robot, int procedureTime)**

Each procedure must check if the robot procedure time is **more than or equal to** the time each procedure will take. If robot procedure time is lower than the time for the current procedure throw **IllegalArgumentException** with message **"**Robot doesn't have enough procedure time**"**

**NOTE: Every time when doService() method has called, current robot is added to the robot's collection and the time the procedure took is subtracted from the robot’s allowed procedure time.**

#### Child Classes

There are several concrete types of **procedures**, which execute **different logic** when doService() is called:

* Repair class
  + doSerice()– removes **5** happiness and repairs current robot. Robot can be repaired once. If robot is already repaired throw an **IllegalArgumentException** with message **"**{robotName} is already repaired**"**
* Charge class
  + doSerice() – adds **12** happiness and **10** energy
* Work class
  + doSerice() – removes **6** energy and adds **12** happiness

### Garage

The GarageImpl is a **class** which **should be able to be instantiated**.

The **garage** is a building, which holds **robots**.

#### Data

* CAPACITY – **int** with a **constant** value of **10**
* robots – Map with the robot's **name** as the **key** and the **robot itself** as the **value**

**NOTE: You should keep robots in order of adding**

#### Constructor

A **garage** should not take any values upon initialization:

#### Behavior

##### void manufacture(Robot robot)

If there isn't enough capacity in the garage throw an IllegalStateException with the message **"**Not enough capacity**"**

If a robot with this name already exists in the garage, throw an **IllegalArgumentException** with the message **"**Robot {robot name} already exist**"**

In any other case, add the current robot to the garage.

##### void sell(String robotName, String owner)

If the provided robot name does not exist in the garage, throw an **IllegalArgumentException** with the message **"**Robot {robot name} does not exist**"**

If a robot with that name exists, **change its owner**, its **bought status** and **remove the robot from the garage**.

## Task 2: Business Logic (150 points)

### The Controller Class

The business logic of the program should be concentrated around several **commands**. You are given interfaces, which you have to implement in the correct classes.

**Note: The** Controller **class SHOULD NOT handle exceptions! The tests are designed to expect exceptions, not messages!**

The first interface is Controller. You must create a ControllerImplclass, which implements the interface and implements all of its methods. The constructor of Controllerdoes not take any arguments. The given methods should have the logic described for each in the Commands section.

### Data

You need to keep track of some things; this is why you need some private fields in your controller class:

* **garage - Garage**
* **charge - Procedure**
* **repair – Procedure**
* **work - Procedure**

### Commands

There are several commands which control the business logic of the application. They are stated below.

**NOTE:** For each command except for **"**Manufacture**" and "History"**, you must check if a robot with that name exist in the garage. If it doesn't, throw an **IllegalArgumentException** with the message **"**Robot {robotName} does not exist**"**.

#### Manufacture Command

##### Parameters

* robotType – String
* name – String
* energy – int
* happiness – int
* procedureTime – int

##### Functionality

Creates a robot with the correct type and **registers** it in the garage. **Valid** types are: "**Cleaner**" and "**Housekeeper**"

If the robot type is invalid throw **IllegalArgumentException** with the message "{robotName} type doesn't exist"

If successful, returns "Robot {robotName} registered successfully".

#### Repair Command

##### Parameters

* robotName - String
* procedureTime - int

##### Functionality

Calls the Repair procedure with parameters **Robot** with the given name and **procedureTime**.

Returns "{robotName} had repair procedure".

#### Work Command

##### Parameters

* robotName - String
* procedureTime - int

##### Functionality

Calls the **Work** procedure with parameters **Robot** with the given name and **procedureTime**.

Returns "{current robot name} was working for {procedure time} hours".

#### Charge Command

##### Parameters

* robotName - String
* procedureTime - int

##### Functionality

Calls the **Charge** procedure with parameters **Robot** with the given name and **procedureTime**.

Returns "{current robot name} had charge procedure".

#### Sell Command

##### Parameters

* robotName - String
* ownerName - String

##### Functionality

Finds the **Robot** with the given name in the garage and sells it.

Returns:

**"**{ownerName} bought {robotName} robot**"**

#### History Command

##### Parameters

* procedureType - String

##### Functionality

Returns information about **all robots** which had current procedure type in the format:

"{procedureType}"

" Robot type: {robotType} - {robotName} - Happiness: {happiness} - Energy: {energy}"

#### Exit Command

##### Functionality

Ends the program.

## Input / Output

You are provided with one interface, which will help you with the correct execution process of your program. The interface is Engine and the class implementing this interface should read the input and when the program finishes, this class should print the output.

You are given the **EngineImpl** class with written logic in it. In order the code to be **compiled**, some parts are **commented**, **don’t forget to comment them out**. The **try-catch block** is also **commented** in order for the program to **throw exceptions and for you to see them**, **comment it out** when you are **ready** with this too.

### Input

Below, you can see the **format** in which **each command** will be given in the input:

* Manufacture {type} {name} {energy} {happiness} {procedureTime}
* Repair {robotName} {procedureTime}
* Work {robotName} {procedureTime}
* Charge {robotName} {procedureTime}
* Sell {robotName} {owner}
* History {procedureType}
* Exit

### Output

Print the output from each command when issued. If an exception is thrown during any of the commands' execution, print the exception message.

### Examples

|  |
| --- |
| **Input** |
| Manufacture Housekeeper Cortana 30 50 6  Repair Cortana 3  Sell Cortana John  Manufacture Cleaner Alexa 20 40 5  Repair Alexa 3  Work Alexa 1  Manufacture Lion Cortana 30 50 6  Work Alexa 4  Charge Alexa 6  History Repair  Sell Alexa John  Exit |
| **Output** |
| Robot Cortana registered successfully  Cortana had repair procedure  John bought Cortana robot  Robot Alexa registered successfully  Alexa had repair procedure  Alexa was working for 1 hours  Lion type doesn't exist  Robot doesn't have enough procedure time  Robot doesn't have enough procedure time  Repair  Robot type: Housekeeper - Cortana - Happiness: 45 - Energy: 30  Robot type: Cleaner - Alexa - Happiness: 47 - Energy: 14  John bought Alexa robot |

|  |
| --- |
| **Input** |
| Manufacture Cleaner Siri 100 50 100  Manufacture Cleaner Alexa 50 40 80  Manufacture Cleaner Sophia 60 40 60  Manufacture Housekeeper Cortana 10 20 15  Manufacture Housekeeper Cortana 10 20 14  Manufacture InvalidRobot FalseName 20 40 15  Manufacture Cleaner InvalidEnergy -20 40 15  Manufacture Housekeeper InvalidHappines 20 -40 15  Repair Cortana 10  Repair Cortana 10  Repair Alexa 10  Repair Sophia 10  Charge Sophia 10  Charge InvalidName 20  Work Siri 20  Sell Siri John  Sell Cortana Alex  Sell Invalid Name Alex  Manufacture Cleaner Koly 10 20 13  Manufacture Housekeeper Willy 43 20 100  Manufacture Housekeeper Jack 10 34 55  Manufacture Cleaner Fast 80 20 14  Manufacture Housekeeper Zoom 22 12 90  Manufacture Cleaner Spider 50 60 100  Manufacture Cleaner Robot010 10 44 50  Manufacture Housekeeper Oliver 35 55 10  History Repair  Exit |
| **Output** |
| Robot Siri registered successfully  Robot Alexa registered successfully  Robot Sophia registered successfully  Robot Cortana registered successfully  Robot Cortana already exist  InvalidRobot type doesn't exist  Invalid energy  Invalid happiness  Cortana had repair procedure  Robot doesn't have enough procedure time  Alexa had repair procedure  Sophia had repair procedure  Sophia had charge procedure  Robot InvalidName does not exist  Siri was working for 20 hours  John bought Siri robot  Alex bought Cortana robot  Robot Invalid does not exist  Robot Koly registered successfully  Robot Willy registered successfully  Robot Jack registered successfully  Robot Fast registered successfully  Robot Zoom registered successfully  Robot Spider registered successfully  Robot Robot010 registered successfully  Robot Oliver registered successfully  Repair  Robot type: Housekeeper - Cortana - Happiness: 15 - Energy: 10  Robot type: Cleaner - Alexa - Happiness: 35 - Energy: 50  Robot type: Cleaner - Sophia - Happiness: 47 - Energy: 70 |

## Task 3: Unit testing (100 points)

You will receive a skeleton with **Computer** and **Part** classes inside. The classes will have some methods and fields, which are working properly. You are **NOT ALLOWED** to change any classes. Cover the whole class with unit tests to make sure that the class is working as intended. Submit only "**computers"** package with **ComputerTests** class inside.

You are provided with a **unit test project** in the **project skeleton**. Do **NOT** use **Mocking** in your unit tests!