

# Java OOP Demo Exam - 07 Dec 2019

## Motocross World Championship MXGP

### 1. Overview

The FIM(Fédération Internationale de Motocyclisme) Motocross World Championship (MXGP) is one of the biggest motocross championship ever. You love riding motorcycles and you are the biggest fan on MXGP and for that reason MXGP hired you to create a platform for storing information about riders, motorcycles and races.

### 2. Setup

- Upload **only the MXGP** package in every problem **except Unit Tests**
- **Do not modify the classes, 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

### 3. Task 1: Structure (50 points)

You are given **4** interfaces, and you have to implement their functionality in the **correct classes**.

It is not required to implement your structure with **CommandHandler**, **InputReader**, **OutputWriter** and etc. It's good practice but it's not required.

There are **3** types of entities and 3 repositories in the application: **Motorcycle**, **Rider**, **Race** and **Repository**:

#### Motorcycle

**MotorcycleImpl** is a **base class** for any **type of motorcycle** and it **should not be able to be instantiated**.

#### Data

- **model** – **String** (If the model is **null**, **whitespace** or **less than 4 symbols**, throw an **IllegalArgumentException** with message "**Model {model} cannot be less than 4 symbols.**") – All models are unique
- **horsePower** – **int** (every type of motorcycle has a different range of valid horsepower. If the horsepower is not in the valid range, throw an **IllegalArgumentException** with message "**Invalid horse power: {horsepower}.**")
- **cubicCentimeters** – **double**(every type of motorcycle has different cubic centimeters)

#### Behavior

**double calculateRacePoints(int laps)**

The **calculateRacePoints** calculates the race points in the concrete race with this formula:

**cubic centimeters / horsepower \* laps**

## Constructor

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

**String** model, **int** horsepower, **double** cubicCentimeters

## Child Classes

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

### PowerMotorcycle

The **cubic centimeters** for this type of motorcycle are **450**. Minimum **horsepower** is **70** and maximum **horsepower** is **100**.

If you receive horsepower which is not in the given range throw **IllegalArgumentException** with message "Invalid horse power: {horsepower}."

### SpeedMotorcycle

The **cubic centimeters** for this type of motorcycle are **125**. Minimum **horsepower** is **50** and maximum **horsepower** is **69**.

If you receive horsepower which is not in the given range throw **IllegalArgumentException** with message "Invalid horse power: {horsepower}."

## RiderImpl

### Data

- **name** – **String** (If the name is **null**, **empty** or less than **5 symbols** throw an **IllegalArgumentException** with message "Name {name} cannot be less than 5 symbols.") – All names are unique
- **motorcycle** – **Motorcycle**
- **numberOfWins** – **int**
- **canParticipate** – **boolean** (default behaviour is **false**). A rider can participate in a race, **ONLY** if he has motorcycle (motorcycle is not **null**)

### Behavior

#### **void** addMotorcycle(Motorcycle motorcycle)

This method adds a motorcycle to a rider. If a motorcycle from method parameters is **null**, throw **NullPointerException** with message "**Motorcycle cannot be null**".

If motorcycle is not null, save it and after that rider can participate to race.

#### **void** winRace()

When a rider wins a race, the number of wins should be increased with one.

## Constructor

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

**String** name

## RaceImpl

### Data

- **name** – **String** (If the name is **null**, **empty** or less than **5 symbols** throw an **IllegalArgumentException** with message "**Name {name} cannot be less than {5} symbols.**") – All names are unique
- **laps** – **int** (throws **IllegalArgumentException** with message "**Laps cannot be less than 1.**" If laps are less than 1)
- **riders** – **Collection** of riders

### Behavior

#### **void addRider(Rider rider)**

This method adds a rider to the race if the rider is valid. If the rider is not valid, throw exception with the appropriate message. Exceptions are:

- If the rider is **null** throw **NullPointerException** with message "**Rider cannot be null.**"
- If the rider **cannot** participate to race (rider has not motorcycle) throw **IllegalArgumentException** with message "**Rider {rider name} could not participate in race.**"
- If the rider already **exists** in the race throw **IllegalArgumentException** with message "**Rider {rider name} is already added in {race name} race.**"

## Repository

The repository holds information for the entity.

### Data

- **models** – **collection** of **T (entity)**

### Behavior

#### **void add(T model)**

Adds an entity in the collection.

#### **boolean remove(T model)**

Removes an entity from the collection.

#### **T getByName(String name)**

Returns an entity with that name.

#### **Collection<T> getAll()**

Returns all entities (unmodifiable)

### Child Classes

Create Motorcycle, Rider and Race repositories.

## 4. 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 `ChampionshipControllerImpl` class SHOULD NOT handle exceptions! The tests are designed to expect exceptions, not messages!**

The first interface is **`ChampionshipController`**. You must implement a **`ChampionshipControllerImpl`** class, which implements the interface and implements all of its methods. The given methods should have the following logic:

## Commands

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

### CreateRider Command

#### Parameters

- `riderName` - `String`

#### Functionality

Creates a rider with the given name and adds it to the appropriate repository.

The method should **return** the following message:

**"Rider {name} is created."**

If already have rider with given name in the rider repository, throw **`IllegalArgumentException`** with message **"Rider {name} is already created."**

### CreateMotorcycle Command

#### Parameters

- `type` - `String`
- `model` - `String`
- `horsePower` - `int`

#### Functionality

Create a **motorcycle** with the provided **model** and **horsepower** and add it to the repository. There are two types of motorcycle: **"Speed"** and **"Power"**.

If the **motorcycle** already exists in the appropriate repository throw **`IllegalArgumentException`** with following message:

**"Motorcycle {model} is already created."**

If the motorcycle is successfully created, the method should **return** the following message:

**"{"SpeedMotorcycle"/ "PowerMotorcycle"} {model} is created."**

### AddMotorcycleToRider Command

#### Parameters

- `riderName` - `String`
- `motorcycleModel` - `String`

#### Functionality

Gives the motorcycle with given name to the rider with given name (if exists).

If the rider **does not exist** in rider repository, throw **`NullPointerException`** with message

- **"Rider {name} could not be found."**

If the motorcycle **does not exist** in motorcycle repository, throw **NullPointerException** with message

- "Motorcycle {name} could not be found."

If everything is successful you should add the motorcycle to the rider and return the following message:

- "Rider {rider name} received motorcycle {motorcycle name}."

## AddRiderToRace Command

### Parameters

- `raceName` - `String`
- `riderName` - `String`

### Functionality

Adds rider to the race.

If the race **does not exist** in race repository, throw **NullPointerException** with message

- "Race {name} could not be found."

If the rider **does not exist** in rider repository, throw **NullPointerException** with message

- "Rider {name} could not be found."

If everything is successful you should add the rider to the race and return the following message:

- "Rider {rider name} added in {race name} race."

## CreateRace Command

### Parameters

- `name` - `String`
- `laps` - `int`

### Functionality

Creates race with given name and laps and adds to race repository.

If race with given name already **exists** throw **IllegalArgumentException** with message:

- "Race {name} is already created."

If everything is successful you should return the following message:

- "Race {name} is created."

## StartRace Command

### Parameters

- `raceName` - `String`

### Functionality

This method is the most important one. If everything is valid, you should **arrange** all riders and then return the three fastest riders. In order to do this you should sort all riders in **descending** order by the result of **calculateRacePoints** method in motorcycle object. At the end if everything is valid remove this race from race repository.

If the race **does not exist** in race repository, throw **NullPointerException** with message:

- "Race {name} could not be found."

If the participants in the race are less than 3, throw `IllegalArgumentException` with message:

- "Race {race name} cannot start with less than 3 participants."

If everything is successful you should return the following message:

- "Rider {first rider name} wins {race name} race."  
 "Rider {second rider name} is second in {race name} race."  
 "Rider {third rider name} is third in {race name} race."

## End Command

Exit the program.

## Input / Output

You are provided with one interface, which will help 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.

### Input

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

- `CreateRider {name}`
- `CreateMotorcycle {motorcycle type} {model} {horsepower}`
- `AddMotorcycleToRider {rider name} {motorcycle name}`
- `AddRiderToRace {race name} {rider name}`
- `CreateRace {name} {laps}`
- `StartRace {race name}`
- `End`

### 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

```
CreateRider Michael
CreateRider Peter
CreateMotorcycle Speed Honda 60
CreateMotorcycle Power Suzuki 80
CreateMotorcycle Power Yamaha 70
CreateRace Loket 2
AddMotorcycleToRider Michael Honda
AddMotorcycleToRider Peter Suzuki
AddMotorcycleToRider Michael Yamaha
```

StartRace Locket

AddRiderToRace Locket Michael

AddRiderToRace Locket Peter

StartRace Locket

CreateRider Brian

AddRiderToRace Locket Brian

CreateMotorcycle Speed KTM-SX 55

AddMotorcycleToRider Brian KTM-SX

AddRiderToRace Locket Brian

StartRace Locket

End

### Output

Rider Michael is created.

Rider Peter is created.

SpeedMotorcycle Honda is created.

PowerMotorcycle Suzuki is created.

PowerMotorcycle Yamaha is created.

Race Locket is created.

Rider Michael received motorcycle Honda.

Rider Peter received motorcycle Suzuki.

Rider Michael received motorcycle Yamaha.

Race Locket cannot start with less than 3 participants.

Rider Michael added in Locket race.

Rider Peter added in Locket race.

Race Locket cannot start with less than 3 participants.

Rider Brian is created.

Rider Brian could not participate in race.

SpeedMotorcycle KTM-SX is created.

Rider Brian received motorcycle KTM-SX.

Rider Brian added in Locket race.

Rider Michael wins Locket race.

Rider Peter is second in Locket race.

Rider Brian is third in Locket race.

### Input

CreateRider Kevin

CreateRider Kevin

```
CreateRider Jose
CreateMotorcycle Speed KTM-SX-F 100
CreateMotorcycle Power KTM 100
CreateMotorcycle Power KTM-SX-F 100
CreateMotorcycle Power KTM-SX-F 100
StartRace Imola
CreateRace Imola 4
AddRiderToRace Lommel Kevin
AddRiderToRace Imola Jose
AddRiderToRace Imola Kevin
AddMotorcycleToRider Kevin KTM-SX-F
AddRiderToRace Imola Kevin
CreateMotorcycle Speed Honda 60
CreateMotorcycle Power Suzuki 80
CreateMotorcycle Power Yamaha 70
CreateRace Loket 2
CreateRider Michael
CreateRider Peter
AddMotorcycleToRider Michael Honda
AddMotorcycleToRider Peter Suzuki
AddRiderToRace Imola Michael
AddRiderToRace Imola Peter
StartRace Imola
End
```

#### Output

```
Rider Kevin is created.
Rider Kevin is already created.
Name Jose cannot be less than 5 symbols.
Invalid horse power: 100.
Model KTM cannot be less than 4 symbols.
PowerMotorcycle KTM-SX-F is created.
Motorcycle KTM-SX-F is already created.
Race Imola could not be found.
Race Imola is created.
Race Lommel could not be found.
Rider Jose could not be found.
Rider Kevin could not participate in race.
Rider Kevin received motorcycle KTM-SX-F.
```



Rider Kevin added in Imola race.  
SpeedMotorcycle Honda is created.  
PowerMotorcycle Suzuki is created.  
PowerMotorcycle Yamaha is created.  
Race Locket is created.  
Rider Michael is created.  
Rider Peter is created.  
Rider Michael received motorcycle Honda.  
Rider Peter received motorcycle Suzuki.  
Rider Michael added in Imola race.  
Rider Peter added in Imola race.  
Rider Peter wins Imola race.  
Rider Kevin is second in Imola race.  
Rider Michael is third in Imola race.

## 5.Task 3: Unit Tests (100 points)

You will receive a skeleton with **RaceEntry**, **UnitMotorcycle** and **UnitRider** classes inside. The class will have some methods, fields and one constructor, which are working properly. You are **NOT ALLOWED** to change any class. Cover the whole class with unit tests to make sure that the class is working as intended.

You are provided with a **unit test project** in the **project skeleton**.

Note: The **RaceEntry** is the class you need to test.

Do **NOT** use **Mocking** in your unit tests!