# Java OOP Exam - Easter Races

## Overview

It’s that time of the year again and the annual Easter races are about to begin. You love to race with your car and you are the biggest fan of the Easter races and for that reason, the Easter races federation hired you to create platform for storing information about drivers, cars and races.

## Setup

* Upload **only the** easterRacespackage 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

## Task 1: Structure (50 points)

You are given **8** interfaces, and you have to implement their functionality in the **correct classes**.

It is not required to implement your structure with Engine, ConsoleReader, ConsoleWriter and enc. It's good practice but it's not required.

There are **3** types of entities and 3 repositories in the application: **Car, Driver, Race and a Repository for each of them**:

### Car

BaseCar is a **base class** for any **type of Car** 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 car has 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 car 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 **BaseCar** should take the following values upon initialization:

String model, int horsePower, double cubicCentimeters

#### Child Classes

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

##### MuscleCar

The **cubic centimeters** for this type of car are 5000. Minimum **horsepower** is 400 and maximum **horsepower** is 600.

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

##### SportsCar

The **cubic centimeters** for this type of car are 3000. Minimum **horsepower** is 250 and maximum **horsepower** is 450.

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

### Driver

**DriverImpl** class

#### 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
* **car** - Car
* **numberOfWins** - int
* canParticipate **-** boolean
  + Default behaviour is false
  + **Driver** can participate in race, **ONLY** if he has a **Car** (**Car** is not null)

#### Behavior

##### void addCar(Car car)

This method adds a **Car** to the **Driver**. If the car is null, throw **IllegalArgumentException** with message **"****Car cannot be null."**.

If the given **Car** is not **null**, set the current **Car** as the given one and after that **Driver** can participate to race.

##### void winRace()

When the **Driver** wins a **Race**, the number of wins should be increased.

#### Constructor

A Drivershould take the following values upon initialization:

(String name)

### Race

**RaceImpl** class

#### 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 the laps are less than **1**.
* drivers - A **c**ollection of **Drivers**

#### Behavior

##### void addDriver(Driver driver)

This method adds a **Driver** to the **Race** if the **Driver** is valid. If the **Driver** is not valid, throw an **Exception** with the appropriate message.

Exceptions are:

* If a **Driver** is **null** throw an **IllegalArgumentException** with message **"****Driver cannot be null."**
* If a **Driver cannot** participate in the **Race** (the **Driver** doesn't own a **Car**) throw an IllegalArgumentException with message **"****Driver {driver name} could not participate in race."**
* If the **Driver** already **exists** in the **Race** throw an **IllegalArgumentException** with message:  
  **"****Driver {driver name} is already added in {race name} race."**

#### Constructor

Racershould take the following values upon initialization:

(String name, int laps)

### Repository

The repository holds information about the entity.

#### Data

* Models - A **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 **CarRepository**, **DriverRepository** and **RaceRepository** repositories.

## 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 implement a ControllerImplclass, 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.

#### CreateDriver Command

##### Parameters

* **driverName** - **String**

##### Functionality

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

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

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

**If a driver with the given name already exists in the driver repository, throw** an IllegalArgumentException **with message**"Driver {name} is already created."

#### CreateCar Command

##### Parameters

* type - String
* model - String
* horsePower - int

##### Functionality

Create a **Car** with the provided **model** and **horsepower** and add it to the repository. There are two types of **Car**: **"MuscleCar"** and **"SportsCar".**

If the **Car** already exists in the appropriate repository throw an IllegalArgumentException with following message:

"Car {model} is already created."

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

"{"MuscleCar"/ "SportsCar"} {model} is created."

#### AddCarToDriver Command

##### Parameters

* driverName - String
* carModel - String

##### Functionality

Gives the **Car** with given name to the **Driver** with given **name** (if exists).

If the **Driver does not exist** in the **DriverRepository**, throw **IllegalArgumentException** with message

* "Driver {name} could not be found."

If the **Car does not exist** in the **CarRepository**, throw **IllegalArgumentException** with message

* "Car {name} could not be found."

If everything is successful you should add the **Car** to the **Driver** and return the following message:

* "Driver {driver name} received car {car name}."

#### AddDriverToRace Command

##### Parameters

* raceName - string
* driverName - string

##### Functionality

Adds a **Driver** to the **Race**.

If the **Race does not exist** in the **RaceRepository**, throw an **IllegalArgumentException** with message:

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

If the **Driver does not exist** in the **DriverRepository**, throw an **IllegalArgumentException** with message:

* "Driver {name} could not be found."

If everything is successful, you should add the **Driver** to the **Race** and return the following message:

* "Driver {driver name} added in {race name} race."

#### CreateRace Command

##### Parameters

* name - string
* laps - int

##### Functionality

Creates a **Race** with the given **name** and **laps** and adds it to the **RaceRepository**.

If the **Race** with the given **name** already **exists**, throw an **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 big deal. If everything is valid, you should **arrange** all **Drivers** and then return the three fastest **Drivers**. To do this you should sort all **Drivers** in **descending** order by the result of CalculateRacePoints method in the **Car** object. At the end, if everything is valid **remove** this **Race** from the race repository.

If the **Race does not exist** in **RaceRepository**, throw an **IllegalArgumentException** with message:

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

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

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

If everything is successful, you should return the following message:

* "Driver {first driver name} wins {race name} race."  
  "Driver {second driver name} is second in {race name} race."  
  "Driver {third driver 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:

* **CreateDriver** **{name}**
* **CreateCar** **{car type} {model} {horsepower}**
* **AddCarToDriver {driver name} {car name}**
* **AddDriverToRace {race name} {driver 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** |
| CreateDriver Michael  CreateDriver Peter  CreateCar Sports Porsche 380  CreateCar Muscle Mustang 580  CreateCar Muscle Corvette 440  CreateRace Daytona 2  AddCarToDriver Michael Porsche  AddCarToDriver Peter Mustang  AddCarToDriver Michael Corvette  StartRace Daytona  AddDriverToRace Daytona Michael  AddDriverToRace Daytona Peter  StartRace Daytona  CreateDriver Brian  AddDriverToRace Daytona Brian  CreateCar Sports Mazda 350  AddCarToDriver Brian Mazda  AddDriverToRace Daytona Brian  StartRace Daytona  End |
| **Output** |
| Driver Michael is created.  Driver Peter is created.  SportsCar Porsche is created.  MuscleCar Mustang is created.  MuscleCar Corvette is created.  Race Daytona is created.  Driver Michael received car Porsche.  Driver Peter received car Mustang.  Driver Michael received car Corvette.  Race Daytona cannot start with less than 3 participants.  Driver Michael added in Daytona race.  Driver Peter added in Daytona race.  Race Daytona cannot start with less than 3 participants.  Driver Brian is created.  Driver Brian could not participate in race.  SportsCar Mazda is created.  Driver Brian received car Mazda.  Driver Brian added in Daytona race.  Driver Michael wins Daytona race.  Driver Peter is second in Daytona race.  Driver Brian is third in Daytona race. |

|  |
| --- |
| **Input** |
| CreateDriver Kevin  CreateDriver Kevin  CreateDriver Jose  CreateCar Sports Ford 500  CreateCar Sports Kia 300  CreateCar Muscle Ford 550  CreateCar Muscle Ford 550  StartRace LeMans  CreateRace LeMans 4  AddDriverToRace Dakar Kevin  AddDriverToRace LeMans Jose  AddDriverToRace LeMans Kevin  AddCarToDriver Kevin Ford  AddDriverToRace LeMans Kevin  CreateCar Sports Porsche 380  CreateCar Muscle Mustang 490  CreateCar Muscle Dodge 500  CreateRace Daytona 2  CreateDriver Michael  CreateDriver Peter  AddCarToDriver Michael Porsche  AddCarToDriver Peter Mustang  AddDriverToRace LeMans Michael  AddDriverToRace LeMans Peter  StartRace LeMans  End |
| **Output** |
| Driver Kevin is created.  Driver Kevin is already created.  Name Jose cannot be less than 5 symbols.  Invalid horse power: 500.  Model Kia cannot be less than 4 symbols.  MuscleCar Ford is created.  Car Ford is already created.  Race LeMans could not be found.  Race LeMans is created.  Race Dakar could not be found.  Driver Jose could not be found.  Driver Kevin could not participate in race.  Driver Kevin received car Ford.  Driver Kevin added in LeMans race.  SportsCar Porsche is created.  MuscleCar Mustang is created.  MuscleCar Dodge is created.  Race Daytona is created.  Driver Michael is created.  Driver Peter is created.  Driver Michael received car Porsche.  Driver Peter received car Mustang.  Driver Michael added in LeMans race.  Driver Peter added in LeMans race.  Driver Peter wins LeMans race.  Driver Kevin is second in LeMans race.  Driver Michael is third in LeMans race. |

## Task 3: Unit Tests (100 points)

You will receive a skeleton with one class inside. The class will have some methods, fields and constructors. Cover the whole class with unit test to make sure that the class is working as intended.