# C# OOP Exam – 9 April 2022

## Formula 1

## 1. Overview

You have to create a **`Formula1`** project, which stores information about pilots, cars, and races. There will be different types of cars.

## 2. Setup

- Upload **only the `Formula1`** project in every problem **except Unit Tests.**
- **Do not modify the interfaces or their namespaces.**
- 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** or **properties** in the concrete class than the interface has defined.
- Make sure you have **no public fields** anywhere.
- **Exception messages** and **output messages** can be found in the **`"Utilities"`** folder.
- For solving this problem use **Visual Studio 2019,** and **netcoreapp 3.1.**

## 3. 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: **`FormulaOneCar`**, **`Pilot`**, and **`Race`**. There should also be **`FormulaOneCarRepository`**, **`PilotRepository`**, and **`RaceRepository`**.

### FormulaOneCar

The **`FormulaOneCar`** is a **base class** of any **type of car** and it **should not be able to be instantiated**.

#### Data

- **`Model`** - **`string`**
  - If the model **is null, white space,** or the length is less than **3 symbols,** throw an **`ArgumentException`** with a message: `"Invalid car model: { model }."`
  - All names are unique
- **`Horsepower`** - **`int`**
  - If the horsepower **is less than 900, or more than 1050,** throw an **`ArgumentException`** with a message: `"Invalid car horsepower: { horsepower }."`
- **`EngineDisplacement`** - **`double`**
  - If the engine displacement **is less than 1.6, or more than 2.00**, throw an **`ArgumentException`** with a message: `"Invalid car engine displacement: { engine displacement }."`

### Behavior

#### `double RaceScoreCalculator(int laps)`

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

**engine displacement / horsepower * laps**

### Constructor

The constructor of the **`FormulaOneCar`** class should accept the following parameters:

```
string model, int horsepower, double engineDisplacement
```

## Child Classes

There are two concrete types of **`FormulaOneCar`**:

### Ferrari

The constructor should take the following values upon initialization:

```
string model, int horsepower, double engineDisplacement
```

### Williams

The constructor should take the following values upon initialization:

```
string model, int horsepower, double engineDisplacement
```

## Pilot

### Data

- **`FullName`** - **`string`**
  - If the pilot's full name **is null, white space** or the length is less than **5 symbols,** throw an **`ArgumentException`** with a message: `"Invalid pilot name: { fullName }."`
  - All names are unique
- **`CanRace`** - **`bool`**
  - Should be set to **`false`** as default
- **`Car`** - **`IFormulaOneCar`**
  - If the car is null throw a **`NullReferenceException`** with a message: `"Pilot car can not be null."`
- **`NumberOfWins`** - **`int`**

### Behavior

#### `void AddCar(IFormulaOneCar car)`

Sets a **car** to the **pilot**, and set **`CanRace`** to true.

#### `void WinRace()`

The **`WinRace`** method increases the **`NumberOfWins`** by one (1) every time a pilot wins a race.

#### `string ToString()`

Returns a **`string`** with information about **the number of wins for the pilot**. The returned string must be in the following format:

```
"Pilot { full name } has { number of wins } wins."
```

### Constructor

The constructor of the **Pilot** class should accept the following parameters:

```
string fullName
```

## Race

### Data

- **RaceName - string**
  - If the **race** name **is null, white space** or the length is less than **5 symbols,** throw an **ArgumentException** with a message: `"Invalid race name: { race name }."`
  - All race names are unique
- **NumberOfLaps - int**
  - If the number of laps **is less than 1,** throw an **ArgumentException** with a message: `"Invalid lap numbers: { number of laps }."`
- **TookPlace - bool**
  - Should be set to **false** as default
- **Pilots - ICollection<IPilot>**

### Behavior

#### void AddPilot(IPilot pilot)

**Adds** a **pilot** to the race.

#### string RaceInfo()

**Returns** a **string** with **information** about the **race** in the format below:

```
"The { race name } race has:
Participants: { number of participants }
Number of laps: { number of laps }
Took place: { Yes/No }"
```

**Note: Do not use `"\n\r"` for a new line.**

### Constructor

The constructor of the **Race** class should accept the following parameters:

```
string raceName, int numberOfLaps
```

## FormulaOneCarRepository

The **FormulaOneCarRepository** is a **repository** for the **cars**.

### Data

- **Models - a collection of formula one cars (unmodifiable)**

### Behavior

#### void Add(IFormulaOneCar car)

- **Adds a formula one car** to the **collection**

#### bool Remove(IFormulaOneCar car)

- **Removes a** formula one car from the **collection**. **Returns true** if the deletion was **successful**, **otherwise - false**.

#### IFormulaOneCar FindByName(string model)

- **Returns** the **first car of a given model**. **Otherwise**, returns **null**.

### PilotRepository

The **PilotRepository** is a **repository** for the **pilots**.

#### Data

- **`Models` - a collection of pilots (unmodifiable)**

#### Behavior

**`void Add(IPilot pilot)`**

- **Adds a pilot** to the **collection**.

**`bool Remove(IPilot pilot)`**

- **Removes** a pilot from the **collection**. **Returns true** if the deletion was **successful**, **otherwise** - **false**.

**`IPilot FindByName(string fullName)`**

- **Returns** the **first pilot with the given fullName**. **Otherwise**, returns **null**.

### RaceRepository

The **RaceRepository** is a **repository** for the **races**.

#### Data

- **`Models` - a collection of races (unmodifiable)**

#### Behavior

**`void Add(IRace race)`**

- **Adds a race** to the **collection**.

**`bool Remove(IRace race)`**

- **Removes** a race from the **collection**. **Returns true** if the deletion was **successful**, **otherwise** - **false**.

**`IRace FindByName(string raceName)`**

- **Returns** the **first race of a given model**. **Otherwise**, returns **null**.

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

The first interface is **`IController`**. Your task is to create a **`Controller`** class, which implements the interface and implements all of its methods. The constructor of **`Controller`** does not take any arguments. The given methods should have the logic described for each in the Commands section.

**NOTE:** When you create the **`Controller`** class, go into the **`Engine`** class constructor and uncomment the "`this.controller = new Controller();`" line.

## Data

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

- **`pilotRepository`** - **`PilotRepository`**
- **`raceRepository`** - **`RaceRepository`**
- **`carRepository`** - **`FormulaOneCarRepository`**

# Commands

There are several **commands**, which control the **business logic** of the **application**. They are **stated below**. The **`Formula1` name** passed to the methods will **always** be **valid**!

## CreatePilot Command

### Parameters

- **`fullName`** - **`string`**

### Functionality

**Adds** a **`Pilot`** to the **`PilotRepository`**.

- If a **pilot with the given full name exists**, throw a **`InvalidOperationException`** with **the following message:** `"Pilot { full name } is already created."`
- If the **`Pilot`** is **added successfully to the repository**, **return** the following **message**: `"Pilot { full name } is created."`

## CreateCar Command

### Parameters

- **`type`** - **`string`**
- **`model`** - **`string`**
- **`horsepower`** - **`int`**
- **`engineDisplacement`** - **`double`**

### Functionality

Creates a **formula one car** with the given parameters and **adds** it to the **`FormulaOneCarRepository`**. **Valid** types are: "**`Ferrari`**" and "**`Williams`**":

- If a car with the given **model** exists, throw an **`InvalidOperationException`** with a message: `"Formula one car { model } is already created."`
- If the car **type** is **invalid**, throw an **`InvalidOperationException`** with a message: `"Formula one car type { type } is not valid."`
- If **no errors** are **thrown**, **return** a string with the following **message**: `"Car { type }, model { model } is created."`

## CreateRace Command

### Parameters

- **`raceName`** - **`string`**
- **`numberOfLaps`** - **`int`**

### Functionality

**Creates** a race with the given **name**, **number of laps** and adds it to the **`RaceRepository`:**

- If a race with the given **race name exists**, **throw a `InvalidOperationException`** with **the following message**: `"Race { race name } is already created."`
- If **no errors** are **thrown**, **return** a string with the following **message**: `"Race { race name } is created."`

## AddCarToPilot Command

### Parameters

- **`pilotName` - `string`**
- **`carModel` - `string`**

### Functionality

**Adds** a **car** with the given **car model** to a **pilot** with the **given name**. After successfully adding a car to a pilot, remove the car from the **`FormulaOneCarRepository`:**

- If the pilot **does not exist,** or the pilot already **has a car**, throw a **`InvalidOperationException`** with **the following message:** `"Pilot { pilot name } does not exist or has a car."`
- If the **car model does not exist**, throw a **`NullReferenceException`** with **the following message:** `"Car { model } does not exist."`
- If **no errors** are **thrown**, **return** a string with the following **message**: `"Pilot { pilot name } will drive a {type of car} { model } car."`

## AddPilotToRace Command

### Parameters

- **`raceName` - `string`**
- **`pilotFullName` - `string`**

### Functionality

Adds a **pilot** with the **given name**, to the **race** with the **given race name**.

- If the **race does not exist**, throw a **`NullReferenceException`** with **the following message:** `"Race { race name } does not exist."`
- If the **pilot does not exist,** or **the pilot can not race,** or **the pilot is already in the race**, throw a **`InvalidOperationException`** with **the following message:** `"Can not add pilot { pilot full name } to the race."`
- If **no errors** are **thrown**, **return** a string with the following **message**: `"Pilot { pilot full name } is added to the { race name } race."`

## StartRace Command

### Parameters

- **`raceName` - `string`**

### Functionality

If everything is valid, you should **arrange** for all pilots in the given race to start racing. As a result, this method returns **the three fastest pilots**. To execute the race you should sort all riders in **descending** order by the result of the **`RaceScoreCalculator`** method in **`FormulaOneCar`** object. In the end, if everything is valid set the race's **`TookPlace`** property to **`true`**, **increase** the winner's score, and **return** the corresponding message.

- If the **race does not exist**, throw a **`NullReferenceException`** with **the following message:** `"Race { race name } does not exist."`
- If the **race has less than 3 pilots**, throw an **`InvalidOperationException`** with **the following message:** `"Race { race name } cannot start with less than three participants."`

- If the **race has been already executed**, throw an **InvalidOperationException** with **the following message:** `"Can not execute race { race name }."`

- If **no errors** are **thrown**, **return** a string with the following **message**:

```
"Pilot { pilot full name } wins the { race name } race.
Pilot { pilot full name } is second in the { race name } race.
Pilot { pilot full name } is third in the { race name } race."
```

**Note: Do not use `"\n\r"` for a new line.**

## RaceReport Command

### Functionality

Returns information about each **race** that **has been executed**. You can use the **RaceInfo** method in the **Race** class.

```
"The { race name } race has:
Participants: { number of participants }
Number of laps: { number of laps }
Took place: Yes

The { race name } race has:
Participants: { number of participants }
Number of laps: { number of laps }
Took place: Yes

(…)"
```

**Note: Do not use `"\n\r"` for a new line. There is not an empty row between different races.**

## PilotReport Command

### Functionality

Returns information about each pilot, ordered by the number of wins descending. You can use the override **ToString** method in the **Pilot** class.

```
"Pilot {FullName} has {NumberOfWins} wins.
Pilot {FullName} has {NumberOfWins} wins.
(…)"
```

**Note: Do not use `"\n\r"` for a new line. There is not an empty row between different reports.**

## 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 **IEngine** 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 **Engine** class with written logic in it. For the code to be **compiled**, some parts are **commented on**, **don't forget to uncomment them**.

## Input

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

- `CreatePilot { fullName }`
- `CreateCar { type } { model } { horsepower } { engineDisplacement }`
- `CreateRace { raceName } { numberOfLaps }`
- `AddCarToPilot { pilotName } { carModel }`
- `AddPilotToRace { raceName } { pilotFullName }`
- `StartRace { raceName }`
- `RaceReport`
- `PilotReport`
- `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 |
| --- |
| `CreatePilot Charles_Leclerc`<br>`CreateCar Ferrari SF71H 980 1.6`<br>`AddCarToPilot Charles_Leclerc SF71H`<br>`CreateCar Ferrari SF1000 990 1.7`<br>`CreatePilot Carlos_Sainz`<br>`AddCarToPilot Fernando_Alonso SF1000`<br>`AddCarToPilot Carlos_Sainz SF1000`<br>`CreateRace Monaco_GP 78`<br>`StartRace Monaco_GP`<br>`CreateRace Miami_GP 57`<br>`AddPilotToRace Monaco_GP Charles_Leclerc`<br>`AddPilotToRace Monaco_GP Carlos_Sainz`<br>`AddPilotToRace Monaco_GP Fernando_Alonso`<br>`CreateCar Williams FW43B 1025 1.6`<br>`CreatePilot Nicholas_Latifi`<br>`CreateCar Pagani MCL35M 990 1.8`<br>`StartRace Monaco_GP`<br>`AddPilotToRace Monaco_GP Nicholas_Latifi`<br>`CreatePilot Alexander_Albon`<br>`CreateCar Williams FW43 1050 1.9`<br>`AddCarToPilot Alexander_Albon FW43`<br>`AddPilotToRace Monaco_GP Alexander_Albon`<br>`AddCarToPilot Nicholas_Latifi FW43B`<br>`AddPilotToRace Monaco_GP Nicholas_Latifi`<br>`StartRace Monaco_GP`<br>`RaceReport`<br>`Exit` |
| **Output** |
| `Pilot Charles_Leclerc is created.`<br>`Car Ferrari, model SF71H is created.`<br>`Pilot Charles_Leclerc will drive a Ferrari SF71H car.`<br>`Car Ferrari, model SF1000 is created.`<br>`Pilot Carlos_Sainz is created.`<br>`Pilot Fernando_Alonso does not exist or has a car.`<br>`Pilot Carlos_Sainz will drive a Ferrari SF1000 car.`<br>`Race Monaco_GP is created.`<br>`Race Monaco_GP cannot start with less than three participants.`<br>`Race Miami_GP is created.`<br>`Pilot Charles_Leclerc is added to the Monaco_GP race.`<br>`Pilot Carlos_Sainz is added to the Monaco_GP race.` |

```
Can not add pilot Fernando_Alonso to the race.
Car Williams, model FW43B is created.
Pilot Nicholas_Latifi is created.
Formula one car type Pagani is not valid.
Race Monaco_GP cannot start with less than three participants.
Can not add pilot Nicholas_Latifi to the race.
Pilot Alexander_Albon is created.
Car Williams, model FW43 is created.
Pilot Alexander_Albon will drive a Williams FW43 car.
Pilot Alexander_Albon is added to the Monaco_GP race.
Pilot Nicholas_Latifi will drive a Williams FW43B car.
Pilot Nicholas_Latifi is added to the Monaco_GP race.
Pilot Alexander_Albon wins the Monaco_GP race.
Pilot Carlos_Sainz is second in the Monaco_GP race.
Pilot Charles_Leclerc is third in the Monaco_GP race.
The Monaco_GP race has:
Participants: 4
Number of laps: 78
Took place: Yes
```

**Input**

```
CreatePilot Charles_Leclerc
CreateCar Ferrari SF71H 980 1.6
AddCarToPilot Charles_Leclerc SF71H
CreateCar Ferrari SF1000 990 1.7
CreatePilot Carlos_Sainz
AddCarToPilot Carlos_Sainz SF1000
CreateRace Portuguese_GP 50
AddPilotToRace Portuguese_GP Charles_Leclerc
AddPilotToRace Portuguese_GP Carlos_Sainz
StartRace Spanish_GP
CreatePilot Alexander_Albon
CreateCar Williams FW43 1050 1.9
AddCarToPilot Alexander_Albon FW43
CreateRace Miami_GP 57
CreatePilot Nicholas_Latifi
CreateCar Williams FW43B 1025 1.6
AddCarToPilot Nicholas_Latifi FW43B
AddPilotToRace Miami_GP Charles_Leclerc
AddPilotToRace Miami_GP Carlos_Sainz
AddPilotToRace Miami_GP Alexander_Albon
AddPilotToRace Miami_GP Nicholas_Latifi
AddPilotToRace Portuguese_GP Alexander_Albon
AddPilotToRace Portuguese_GP Nicholas_Latifi
AddPilotToRace Portuguese_GP Nicholas_Latifi
CreateRace Monaco_GP 51
StartRace Miami_GP
PilotReport
StartRace Portuguese_GP
RaceReport
Exit
```

**Output**

```
Pilot Charles_Leclerc is created.
Car Ferrari, model SF71H is created.
Pilot Charles_Leclerc will drive a Ferrari SF71H car.
Car Ferrari, model SF1000 is created.
Pilot Carlos_Sainz is created.
Pilot Carlos_Sainz will drive a Ferrari SF1000 car.
```

```
Race Portuguese_GP is created.
Pilot Charles_Leclerc is added to the Portuguese_GP race.
Pilot Carlos_Sainz is added to the Portuguese_GP race.
Race Spanish_GP does not exist.
Pilot Alexander_Albon is created.
Car Williams, model FW43 is created.
Pilot Alexander_Albon will drive a Williams FW43 car.
Race Miami_GP is created.
Pilot Nicholas_Latifi is created.
Car Williams, model FW43B is created.
Pilot Nicholas_Latifi will drive a Williams FW43B car.
Pilot Charles_Leclerc is added to the Miami_GP race.
Pilot Carlos_Sainz is added to the Miami_GP race.
Pilot Alexander_Albon is added to the Miami_GP race.
Pilot Nicholas_Latifi is added to the Miami_GP race.
Pilot Alexander_Albon is added to the Portuguese_GP race.
Pilot Nicholas_Latifi is added to the Portuguese_GP race.
Can not add pilot Nicholas_Latifi to the race.
Race Monaco_GP is created.
Pilot Alexander_Albon wins the Miami_GP race.
Pilot Carlos_Sainz is second in the Miami_GP race.
Pilot Charles_Leclerc is third in the Miami_GP race.
Pilot Alexander_Albon has 1 wins.
Pilot Charles_Leclerc has 0 wins.
Pilot Carlos_Sainz has 0 wins.
Pilot Nicholas_Latifi has 0 wins.
Pilot Alexander_Albon wins the Portuguese_GP race.
Pilot Carlos_Sainz is second in the Portuguese_GP race.
Pilot Charles_Leclerc is third in the Portuguese_GP race.
The Portuguese_GP race has:
Participants: 4
Number of laps: 50
Took place: Yes
The Miami_GP race has:
Participants: 4
Number of laps: 57
Took place: Yes
```

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

You will receive a skeleton with **Shop** and **Smartphone** classes inside. The **Shop** class has some methods, fields, and one constructor, which are working properly. The **Smartphone** class has three properties and a constructor. You are **NOT ALLOWED** to change any class. Cover the whole **Shop** 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**.

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