TO - KTR

T-MSC-000

KTR

SS1

1.0

Table of contents

Administrative details

Hello, and welcome to your first exercise in this brand new adventure you've embarked on! The purpose of this test is to prove that your experience corresponds to the level expected at the beginning of MSc in order to avoid unpleasant surprises once the training has started.

This subject is agnostic language, that is, it has been thought of, and you can do it with your favorite language.

If you choose a compiled language or stack, please commit the binaries in addition to your source code.

Read the entire subject before beginning.

Follow the steps in order.

Bonuses are to be made in order, once the mandatory parts are completed.

Be meticulous about following instructions (that’s also being a developer).

Carry the tests alone

Commits your work as soon as possible 4 hours after receiving it.

Do not commit temporary or files unrelated to your project.

Introduction

In this series of exercises, we will discuss the fundamentals of object-oriented programming.

Unless specified otherwise, all messages must be followed by a newline.

Unless specified otherwise, the getter and setter name will always add “get” or “set” in front of the name of the attribute, in CamelCase.

Try to comply as much as possible with the instructions, taking into account the constraints of the language you have chosen.

Step 00 - Create the repository

1. Create your github account if you don't have one yet
2. Create a private repository named `ktr-msc-ss1`
3. Give the rights to this deposit to the login you were given when you made contact

Step 01

Files to hand in: ./Character.*

Create an abstract `Character` class that is composed of the following protected attributes: "name", "life", "agility", "strength", "wit", and a constant "RPGClass" string attribute, with the corresponding getters. These attributes must have the following values by default:

- name: first argument passed to constructor
- RPGClass: second argument passed to constructor
- life: 50
- agility: 2
- strength: 2
- wit: 2

Add an **attack** method that takes a string as argument, and prints the following (whatever the argument):

```
[name]: Rrrrrrrrr....
```

Of course, [name] must be replaced by your character's name.

Here is an example, in java, in which `TestCharacter` is an implementation of our abstract class which doesn't change any attributes:

```java
public class Example {
    public static void main(String[] args) {
        Character perso = new TestCharacter("Jean-Luc");

        System.out.println(perso.getName());
        System.out.println(perso.getLife());
        System.out.println(perso.getAgility());
        System.out.println(perso.getStrength());
        System.out.println(perso.getWit());
        System.out.println(perso.getRPGClass());
        perso.attack("my weapon");
    }
}
```

```
~/T-MSC-000> java Example
Jean-Luc
50
2
2
2
Character
Jean-Luc: Rrrrrrrrr....
```

STEP 02

Files to hand in:

- ./Character.*
- ./Warrior.*
- ./Mage.*

Create the `Warrior` class as well as a `Mage` class, which *extends* the `Character` class.
Modify each class's attributes as follows:

Warrior
- RPGClass: "Warrior"
- life: 100
- strength: 10
- agility: 8
- wit: 3

Mage
- RPGClass: "Mage"
- life: 70
- strength: 3
- agility: 10
- wit: 10

These two classes must each implement the **attack** method.
Its parameter defines the weapon used to attack.

The Warrior can attack with a "hammer" or a "sword".
If anything else is passed as parameter, he doesn't attack.
The "Warrior" class's **attack** method must display:

```
[name]: Rrrrrrrrr....
[name]: I'll crush you with my [weapon]!
```

The Mage can attack with "magic" or with a "wand".
If anything else is passed as parameter, he doesn't attack.
The "Mage" class's **attack** method must display:

```
[name]: Rrrrrrrrr....
[name]: Feel the power of my [weapon]!
```

Of course, [name] must be replaced by your character's name, and [weapon] by your character's weapon.

Our characters are proud and they like to announce themselves on the battlefield.
You will make sure that when creating a "Warrior" or "Mage" object, a message is written in the following format:

```
[name]: My name will go down in history!
```

for a Warrior, and

```
[name]: May the gods be with me.
```

for a Mage.
Here is an example in java:

```java
public class Example {
    public static void main(String[] args) {
        Character warrior = new Warrior("Jean-Luc");
        Character mage    = new Mage("Robert");

        warrior.attack("hammer");
        mage.attack("magic");
    }
}
```

Terminal

```
~/T-MSC-000> java Example
Jean-Luc: My name will go down in history!
Robert: May the gods be with me.
Jean-Luc: Rrrrrrrrr....
Jean-Luc: I'll crush you with my hammer!
Robert: Rrrrrrrrr....
Robert: Feel the power of my magic!
```

Step 03

Files to hand in:

- ./Character.*
- ./Warrior.*
- ./Mage.*
- ./Movable.*

We now have characters who can be Mages or Warriors.
They can attack, fair enough, but they still cannot move! This is bothersome...
In order to add this behavior to our classes, we are going to create an interface called `Movable` that contains the following methods: **moveRight**, **moveLeft**, **moveForward**, and **moveBack**.

This interface will obviously be implemented by the `Character` classes.

These methods must display the following messages, respectively:

```
[name]: moves right
[name]: moves left
[name]: moves back
[name]: moves forward
```

Step 04

Files to hand in:

- ./Character.*
- ./Warrior.*
- ./Mage.*
- ./Movable.*

Paralysis is over!
Our characters can now move, but, being so proud, they want more!
Our friend Warrior refuses to be compared to a small and skinny Mage.
While the Warrior moves in a bold and virile manner, the Mage moves delicately!

To satisfy our boorish Warrior, implement overloads for the **Movable** methods inherited by `Character`.
Your methods must display the following messages that correspond to the class that overloads them:
for a warrior:

```
[name]: moves right like a bad boy.
[name]: moves left like a bad boy.
[name]: moves back like a bad boy.
[name]: moves forward like a bad boy.
```

for a mage:

```
[name]: moves right furtively.
[name]: moves left furtively.
[name]: moves back furtively.
[name]: moves forward furtively.
```

Here is an example in java:

```java
public class Example {
    public static void main(String[] args) {
        Warrior warrior = new Warrior("Jean-Luc");
        Mage mage       = new Mage("Robert");
```

```
            warrior.moveRight();
            warrior.moveLeft();
            warrior.moveBack();
            warrior.moveForward();
            mage.moveRight();
            mage.moveLeft();
            mage.moveBack();
            mage.moveForward();
        }
    }
```

```
Terminal
~/T-MSC-000> java Example
Jean-Luc: My name will go down in history!
Robert: May the gods be with me.
Jean-Luc: moves right like a bad boy.
Jean-Luc: moves left like a bad boy.
Jean-Luc: moves back like a bad boy.
Jean-Luc: moves forward like a bad boy.
Robert: moves right furtively.
Robert: moves left furtively.
Robert: moves back furtively.
Robert: moves forward furtively.
```

STEP 05

Files to hand in:

- ./Character.*
- ./Warrior.*
- ./Mage.*
- ./Movable.*

Our characters are now customized to talk, walk and attack.
Yet, they still can't unsheathe their weapons!
Being able to attack is nice, but attacking while the weapon is still in its sheath is going to be difficult…

You will agree that, whether Warrior or Mage, the character will draw his weapon the same way.
This is why, you will make sure that the `Character` class implements the **unsheathe** method so that both "Warrior" and "Mage" inherit from it.
However, you will also make sure (if possible, depending on the language you have chosen) that the **unsheathe** method cannot be overloaded by "Warrior"and "Mage".

This method must display the following text when called:

```
[name]: unsheathes his weapon.
```

STEP 06

Files to hand in:

- ./exceptions/Character.*
- ./exceptions/Warrior.*
- ./exceptions/Mage.*
- ./exceptions/Movable.*
- ./exceptions/WeaponException.*

Copy you previous classes in a directory called `exceptions`.
Let's create a **WeaponException** class dedicated to weapons error management.

This class must inherit from the *Exception* class, in which at least two different messages must be declared: when the weapon is not defined, and when it does not fit the character.

Use this new class, to print the following message:

```
[name]: I refuse to fight with my bare hands.
```

when the *attack* method is called with an empty string, and

```
[name]: A [weapon]?? What should I do with this?!
```

for Warrior or

```
[name]: I don't need this stupid [weapon]! Don't misjudge my powers!
```

for Mage, if the weapon does not fit the character.
The `attack` method must *throw* a WeaponException (if possible, depending on the language you have chosen) with the appropriated message in case of errors.
You must also implement a new method tryToAttack that call the `attack` methods, catch the exception and print the message.
Here is an example in java:

```java
public class Example {
    public static void main(String[] args) {
        Character warrior = new Warrior("Jean-Luc");
        Character mage    = new Mage("Robert");

        warrior.tryToAttack("screwdriver");
        mage.tryToAttack("hammer");
        warrior.tryToAttack("hammer");
        try {
                    mage.attack("");
        } catch (WeaponException e) {
            System.out.println(e.getMessage());
            }
        }
    }
```

```
Terminal
~/T-MSC-000> java Example
Jean-Luc: My name will go down in history!
Robert: May the gods be with me.
Jean-Luc: A screwdriver?? What should I do with this?!
Robert: I don't need this stupid hammer! Don't misjudge my powers!
Jean-Luc: Rrrrrrrrr....
Jean-Luc: I'll crush you with my hammer!
Robert: I refuse to fight with my bare hands.
```