Programming

week 5 - exercises

New Austrian Coding School

Exercise 1

Derek goes to a modern art museum and gets fascinated by the shapes and colors of the paintings.
Follow the instructions below:

- Create the *Shape* interface. It has the *getName* method that returns a String.
- Create the *Circle* class that implements the *Shape* interface. Implement the *getName* method with the help of Intellij. Return the word *circle* in that method.

Create the *CircleApplication* class and write the following instructions in its *main* method:

- Create a *Circle* and display its name.
- In a new line, assign the *circle* variable to a *Shape* variable called *shape*. Just use the = symbol.
- Display the *shape*'s name.
- Create another *Circle*, but this time assign it directly to a *Shape* with the name *anotherShape*.
- Display *anotherShape*'s name.
- Create the *CircleTest* class that tests that the *Circle* is working correctly as both *Circle* and *Shape*.

```
Class: circle
Shape: circle
Another shape: circle
```

Exercise 2

Derek moves into another room and sees another painting with many more shapes.
Follow the instructions below:

- Create the *Square* class that implements the *Shape* interface. Implement the *getName* method with the help of Intellij. Return the word *square* in that method. Reuse the *Shape* interface from exercise 1.
- Create the *Triangle* class that implements the *Shape* interface. Implement the *getName* method with the help of Intellij. Return the word *triangle* in that method. Reuse the *Shape* interface from exercise 1.

Create the *BasicShapesApplication* class and write the following instructions in its *main* method:

- Create a *Circle* as a *Shape,* name it *circle* and display its name.
- Create a *Square* as a *Shape,* name it *square* and display its name.
- Create a *Triangle* as a *Shape,* name it *triangle* and display its name.
- Create a list of *Circle* named *circles* and try to add *circle, square* and *triangle* to it. It doesn't work because *circle* is not a *Circle* now, it is a *Shape*, and it's not the same. Also, *square* and *triangle* don't work either because they are not *Circle*s.
- Create a list of *Shape* named *shapes* and add *circle* to it. Try to add *square* and *triangle* as well. It works because the three of them are *shape*s.
- Display the list. It shows memory addresses. Implement the *toString* methods in each one of the three *Shape*s returning the name of the shape itself by reusing the interface method. Display it now.
- Use a *foreach* loop to go through all the *shapes* and one by one display its names.
- Create the *SquareTest* class that tests that the *Square* is working correctly as both *Square* and *Shape*.
- Create the *TriangleTest* class that tests that the *Triangle* is working correctly as both *Triangle* and *Shape*.

```
Shape: circle
Shape: square
Shape: triangle
Shapes: [circle, square, triangle]
Shape name: circle
Shape name: square
Shape name: triangle
```

Exercise 3

Derek gets shocked by all the colors of the paintings in the next room. He takes a picture and sends it to Matilda.
Follow the instructions below:

- Create the *Colored* interface. It has the *getColor* method that returns a String.
- Create the *Rectangle* class that implements the *Colored* interface. Implement the *getColor* method with the help of Intellij. Return the color of your choice in that method.

Create the *BasicColorApplication* class and write the following instructions in its *main* method:

- Create a *Rectangle* as a normal *Rectangle,* name it *rectangle* and display its color.
- Add the *Shape* interface to the *Rectangle* class. Don't replace *Colored*. Just place a comma after *Colored* and add *Shape* right afterwards. Implement the *getName* method with the help of Intellij. Return the word *rectangle* in that method. Reuse the *Shape* interface from exercise 1.
- Display the *rectangle*'s name.
- In a new line, write the following: *Colored colored = (Colored) rectangle;* This is called *casting*. Now the *rectangle* is no longer a *Rectangle*, but it is a *Colored* object. Display its *color*. Note that you cannot see any more the *getName* method because the *Colored* interface does not have it.
- In a new line, write the following: *Shape shape = (Shape) rectangle;* This is called *casting*. Now the *rectangle* is no longer a *Rectangle*, but it is a *Shape* object. Display its *name*. Note that you cannot see any more the *getColor* method because the *Shape* interface does not have it.
- Create a list of *Rectangle* called *rectangles* and add the *rectangle*. Note that you cannot add either *colored* or *shape* because they are not *Rectangle*s.
- Create a list of *Colored* called *coloreds* and add *rectangle* and *colored* to it. Note that you cannot add *shape* because it's not a *Colored* object.
- Create a list of *Shape* called *shapes* and add *rectangle* and *shape* to it. Note that you cannot add *colored* because it's not a *Shape* object.
- Add the *toString* method to the *Rectangle*. Make sure you use its interface methods to return its *color* and *name* as a *String*. Display the three lists
- Create the *RectangleTest* class that tests that the *Rectangle* is working correctly as *Rectangle, Colored* and *Shape*.

```
Color: violet
Name: rectangle
Colored: violet
Shape: rectangle
[violet rectangle]
[violet rectangle, violet rectangle]
[violet rectangle, violet rectangle]
```

Exercise 4

Matilda stops a superhero film she's watching at home and checks Derek's message. She finds it interesting, but she still likes more the film she's enjoying.
Follow the instructions below:

- Create the *Superhero* interface. It has the *getName* method that returns a String.
- Create the *Batman, Superman* and *Spiderman* classes that implement the *Superhero* interface. Implement the *getName* methods with the help of Intellij in each one of them.
- Create the *SuperheroCaller* class that has a list with the three *superheroes*. It has the *call* method that returns a random *Superhero* every time you use it.

Create the *SuperheroApplication* class and write the following instructions in its *main* method:

- Create a *SuperheroCaller* and use it to *call* one *superhero*. Display its name.
- Create a list of *Superhero* and add six superheroes by calling them with the *SuperheroCaller*. Use a lop for this, please.
- Display the *superheroe* names one by one with a loop by also showing the number of appearance.
- Display the number of *superheroes* by using the list's size.
- Create the *SuperheroCallerTest* class to test the *call* method. Make sure it never returns *null* and that the superhero names are either *Batman, Superman* or *Spiderman*.

```
Superhero: Superman
Superhero 1: Batman
Superhero 2: Batman
Superhero 3: Batman
Superhero 4: Spiderman
Superhero 5: Batman
Superhero 6: Spiderman
Number of superheroes: 6
```

Exercise 5

Matilda continues watching the superhero film and the villain causes a big plot twist that the heroes must solve working as a team as soon as possible. They must fight him ten times and weaken him each one until he is finally defeated.

Follow the instructions below:

- Create the *Villain* interface. It has the *weaken* method that receives a *name*.
- Add to the *Superhero* interface from the exercise 4 the *fight* method. It receives a *villain.*
- Update the *Batman, Superman* and *Spiderman* classes from exercise 4 with the *fight* method. Inside it, it *weakens* the *villain*.
- Create the *Joker* class that implements *Villain.* It has the *lives* attribute that starts with the value ten. In its *weaken* method, display the message *"Damn you, <Superhero's name>! You managed to weaken me!"* and subtract one from his *lives* counter. If his *lives* reaches zero, he displays the message *"You all defeated me! But I will be back!"*.

Create the *AdvancedSuperheroApplication* class and write the following instructions in its *main* method:

- Create a *SuperheroCaller*. Reuse the one from exercise 4.
- Create one *Joker*.
- Use the *SuperheroCaller* ten times to *call* one *superhero* and make him *fight* the *joker*.

```
Damn you, Spiderman! You managed to weaken me!
Damn you, Batman! You managed to weaken me!
Damn you, Spiderman! You managed to weaken me!
Damn you, Spiderman! You managed to weaken me!
Damn you, Batman! You managed to weaken me!
Damn you, Batman! You managed to weaken me!
Damn you, Batman! You managed to weaken me!
Damn you, Spiderman! You managed to weaken me!
Damn you, Batman! You managed to weaken me!
Damn you, Spiderman! You managed to weaken me!
You all defeated me! But I will be back!
```

Exercise 6

Hansel finally arrives home after spending the whole morning at Ikea choosing the right wardrobe. He opens the box and reads the instructions. He doesn't want to complicate himself too much. He will just follow the steps in the right order and hope that it works out.

Follow the instructions below:

- Create the *Furniture* interface. It has the *add* method that receives a String representing a *part*.
- Create the *Wardrobe* class that has a list of String representing its different *parts*. It implements *Furniture*. Its *add* method adds the *part* to its *parts.* Implement its *toString* method to display its *parts*.
- Create the *Step* interface. It has the *perform* method that receives a *furniture*.
- Create the *AddSide, AddTop, AddBottom, AddBack, AddShelf* and *AddDoor* classes. They implement *Step*. Each one of them adds only one *part* of the following respectively to the *furniture*: *side, top, bottom, back, shelf* and *door*.
- Create the *WardrobeBuilder* class that has a list of all the *Step*s it has to follow to *build* a *wardrobe*. It has the *build* method. It creates an empty *Wardrobe* and applies all the *steps* one by one to it until it's finished and then it returns it. A *Wardrobe* needs the following to be completed: one bottom, two sides, one back, one top, eight shelves and two doors.

Create the *IkeaApplication* class and write the following instructions in its *main* method:

- Create a *WardrobeBuilder* and *build* one *Wardrobe*.
- Display the *wardrobe*. It should have all the right *parts*.
- Create the *WardrobeBuilderTest* class to make sure the *build* method returns a proper *wardrobe*. For this you will need the *getParts* method in the *Wardrobe* class.

Hints:

- ➢ *In order to create a meaningful test, create a list of string manually with exactly the right parts in the right order. Then you can test that both lists have the same size and that they are exactly the same.*
- ➢ *A more dynamic way to do it would be to place the parts of the wardrobe in a map, counting how many of each are there, and then you check that it is indeed the right amount for each part.*

```
Wardrobe{parts=[bottom, side, side, back, top, shelf, shelf, shelf, shelf, shelf, shelf, shelf, shelf, door, door]}
```

Exercise 7 [Problem solving] (Optional)

Derek, Matilda and Hansel meet in the weekend and they are talking about what to do. Derek wants to go to a museum, Matilda wants to go to the cinema and Hansel wants to go shopping. They decide to play rock, scissors, paper and the winner gets to choose.

Game rules:

- Each player chooses a move. The possible options are *rock*, *scissors* and *paper*.
- Rock defeats scissors. Scissors defeats paper. Paper defeats rock.

Develop the *RockScissorsPaperApplication* following the rules below:

- One human *player* (player 1) plays against the computer *player* (player 2).
- Both players choose a *move*, either rock, scissors or paper.
- The human *player* types in the *move* until it is typed correctly.
- A *decision maker* compares both *moves* and returns the result a message.
- Both *players* answer if they want to play again.
- The *game* repeats itself until either player chooses not to continue playing.
- Make sure all combinations work by creating a test that covers them.

Hints:

- ➢ *Use the strategies to understand the problem and break it down into smaller pieces.*
- ➢ *The purpose of the exercise is not only to develop a game that works. It has to work applying the concept of interfaces.*
- ➢ *Try to abstract interfaces to simplify the solution. There are two types of players: the human and the computer. There are three types of moves: rock, scissors and paper. Player and move seem enough.*
- ➢ *Identify responsibilities and create classes accordingly. There should be at least one class for the main application, one for the game, one for the decision maker that compares the moves. Then the two different players and the three different moves.*
- ➢ *The three moves are only allowed to be created in one class. Every other class interacts with this one to get the moves. You are allowed to use static methods in that class to make class communication easier.*

```
Let’s play rock, paper, scissors!
These are your options: rock, paper, scissors
Choose one:
rok
These are your options: rock, paper, scissors
Choose one:
rock
Player 1 chose: rock
Player 2 chose: scissors
You win!
Do you want to play again? (yes/no)
no
See you next time!
```