Programming

New Austrian Coding School

week 6 - exercises

Exercise 1

Derek goes with his friends on a weekend trip to a summer camp and they find a hot air balloon service that they can use to have a high altitude overview of the long fields.
Follow the instructions below:

- Create the *HotAirBalloon* abstract class. It has the *liftUp* method that displays that the balloon lifts up. It has the *land* method that just displays that the balloon lands.
- Create the *SmallHotAirBalloon* class that extends the *HotAirBalloon* abstract class.
- Create the *BigHotAirBalloon* class that extends the *HotAirBalloon* abstract class.

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

- Create a *SmallHotAirBalloon,* name it *smallBalloon* and use its *liftUp* and *land* methods. Notice that the class had no code, but it's using the code from its parent class *HotAirBalloon.*
- Create a *BigHotAirBalloon,* name it *bigBalloon* and use its *liftUp* and *land* methods. Notice that the class had no code, but it's using the code from its parent class *HotAirBalloon.*
- In a new line, assign the variables *smallBalloon* and *bigBalloon* to *HotAirBalloon* variables called *small* and *big* respectively. Just use the = symbol.
- Use both methods from both *small* and *big* variables. Notice that now we are using them as *HotAirBalloon*s, exactly the same as if it were an interface. The difference is that the *liftUp* and *land* methods are implemented in the abstract and the code is shared among the different classes that extend from it.

```
smallBalloon --
The balloon lifts up
The balloon lands
bigBalloon --
The balloon lifts up
The balloon lands
small --
The balloon lifts up
The balloon lands
big --
The balloon lifts up
The balloon lands
```

Exercise 2

After spending the morning exploring the long fields using a hot air balloon, they decide to sit down and have an ice-cream.
Follow the instructions below:

- Create the *IceCream* abstract class. It has two private String attributes called *flavor* and *topping*. It receives the values through the constructor. It has the abstract method *eat* that returns a String. The method cannot have any implementation because the way we eat it will depend on the specific classes that extend from it.
- Create the *ConeIceCream* class that extends the *IceCream* abstract class. Implement the *eat* method with the help of Intellij. Return a sentence that fulfills the following format: "The <flavor> ice-cream with <topping> is licked". Note that you have no way to access the *flavor* and *topping* attributes. This is because they are private. Not even classes that extend from it can have access to them. Implement the constructor with the help of Intellij. The reason why you're compelled to implement it is because the *IceCream* abstract class requires those values and because they are requested via the constructor, then all its children classes have to provide a constructor to be able to facilitate them.
- Add to the *IceCream* abstract class the protected *getFlavor* and *getTopping* methods that return the attribute values.
- You can complete now the *eat* method from the *ConeIceCream* class. You can use the *getFlavor* and *getTopping* methods inside the *ConeIceCream* class because it extends from *IceCream* class. Classes that

are in the same package can use those methods as well, even if they don't extend from it. Other classes that don't extend from it and are organized in a different package have no access to them.

- Create the *CupIceCream* class that extends the *IceCream* abstract class. Implement the *eat* method with the help of Intellij. Return a sentence that fulfills the following format: "The <flavor> ice-cream with <topping> is eaten with a spoon". Implement the constructor with the help of Intellij. The reason why you're compelled to implement it is because the *IceCream* abstract class requires those values and because they are requested via the constructor, then all its children classes have to provide a constructor to be able to facilitate them.

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

- Create a *ConeIceCream* as an *IceCream,* name it *cone* and display the *eat* message.
- Create a *CupIceCream* as an *IceCream,* name it *cone* and display the *eat* message.

Create the *IceCreamTest* class that tests that the *ConeIceCream* and the *CupIceCream* return the right messages.
Create the *competitor* package and inside of it the *CompetitorIceCreamApplication* class and write the following instructions in its *main* method:

- Create both a *ConeIceCream* and a *CupIceCream* and check whether you can use the *getFlavor* or *getTopping* methods. It doesn't work because the *CompetitorIceCreamApplication* is in a different package and it also does not extend from *IceCream*, therefore the protected methods are not accessible.

```
The Chocolate ice-cream with Oreo is licked
The Vanilla ice-cream with Cookies is eaten with a spoon
```

Exercise 3

After finishing their ice-cream the waiter asks them if they want to have a coffee, and they have so many different coffee machines! Each one of them has to be used differently.
Follow the instructions below:

- Create the *Coffee* class. It has the *madeBy* String attribute and the *brewedTime* Integer attribute. They are provided via constructor. It has the *getMadeBy* and *getBrewedTime* methods.
- Create the *CoffeeMaker* abstract class.
    - It has the *getName* abstract method that returns a String.
    - It has the *getBrewingTime* abstract method that returns an Integer.
    - It has the *brew* method that returns a *Coffee*. It uses the *getName* and *getBrewingTime* methods to create a *Coffee* and returns it.
- Create the *AutomaticCoffeeMachine* class that extends *CoffeeMaker*. Implement its methods with the help of Intellij. Return "Automatic coffee machine" as name and 5 as brewing time. Notice that you don't need to implement the *brew* method because it's not abstract. All *CoffeeMaker*s will share that code without the need of implementing it. However, they need to specify their names and brewing times as customization.
- Create the *FrenchPress* class that extends *CoffeeMaker*. Implement its methods with the help of Intellij. Return "French press" as name and 10 as brewing time.
- Create the *AutoDrip* class that extends *CoffeeMaker*. Implement its methods with the help of Intellij. Return "Auto drip" as name and 6 as brewing time.
- Create the *CoffeeMakers* class. It has the *asList* static method that returns the three *CoffeeMakers* as a list of *CoffeeMaker*. It has the *get* static method that receives a String representing the name of the coffee maker you want to receive, and it returns an *Optional* with the *CoffeeMaker* you wanted if it found it within its list of available ones. Otherwise an empty one.

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

- Use the *CoffeeMakers* class to receive all available coffee makers as a list. For each one of them, *brew* a *Coffee* and display the *madeBy* and *brewedTime* from it.
- Use the *CoffeeMakers* class to *get* one by one each *CoffeeMaker*. Check with the *Optional* if they are present, and if so, use them to *brew* a *Coffee* and display the *madeBy* and *brewedTime* from it.
- Use the *CoffeeMakers* class to *get* one maker that does not exist. Check with the *Optional* that it is not present and display a message saying that this particular coffee maker is not available.

Create the *CoffeeMakersTest* class that tests that uses the *CoffeeMakers* class to *get* each one of them and test that it returns the right ones by checking their names and their brewing times. Test also the case when the wrong name is given, expecting that the received *Optional* will be empty.

```
Automatic coffee machine brewed a coffee and it took 5 minutes
French press brewed a coffee and it took 10 minutes
Auto drip brewed a coffee and it took 6 minutes
Automatic coffee machine brewed a coffee and it took 5 minutes
French press brewed a coffee and it took 10 minutes
Auto drip brewed a coffee and it took 6 minutes
The coffee maker with the name Moka pot is not available
```

Exercise 4 [Problem solving]

The waiter arrives and gets asked for the check. Hansel decides to invite them and pays with a money ticket. The waiter calculates the change of the cents and gives it back to him, making sure he returns the least amount of coins.
Develop the code that the waiter uses in order to return the least amount of cent coins.
Follow the instructions below:

- Create the *Money* abstract class that will represent one single cent coin. Design this class with the help of attributes, methods or abstract methods, so that at least it can tell its amount as Integer and whether it is applicable for an Integer amount.
- Create the *Cent50, Cent20, Cent10, Cent5, Cent2 and Cent1* classes that extend from *Money* and represent each particular coin. Assume that only these coins can be given back as change. Forget about the 1 and 2 coins and every ticket. You want to use Integer numbers. If you use Double, the program might misbehave with decimal approximations.
- Create the *GreedyAlgorithm* class. It has the *change* method that receives an Integer amount of the cents that the waiter has to give back, and it returns a list of *Money* with the exact coins that he has to give back.
- Create the *GreedyAlgorithmTest* class that checks at least the following cases:
    - The amount given is zero, then the change list should be empty.
    - The amount given is 88, then the change list should contain 6 coins.
    - The amount given is 149, then the change list should contain 7 coins.

Exercise 5 [Problem solving]

Matilda interrupts the waiter as he is calculating the change and says that she wants to invite them because she already owed them from the previous trip. She decides to pay with a credit card. The waiter takes the credit card and swipes it in the machine.
Develop the code that the charging machine uses to determine whether the given credit card number is an American Express, a MasterCard, a Visa or is invalid according to Luhn's algorithm.

Credit card rules:

- American Express uses 15-digit numbers.
- American Express numbers all start with 34 or 37.
- MasterCard uses 16-digit numbers.
- MasterCard numbers start with 51, 52, 53, 54, or 55.
- Visa uses 13- and 16-digit numbers.
- Visa numbers all start with 4.
- Every credit card number must be syntactically valid, according to the Luhn's algorithm.

Luhn's algorithm explanation:

- Multiply every other digit by 2, starting with the number's second-to-last digit, and then add those products' digits together.
- Add the sum to the sum of the digits that weren't multiplied by 2.
- If the total's last digit is 0 (or the total modulo 10 equals 0), the number is valid!

Example of the Luhn's algorithm sequence on an American Express with the number 378282246310005:

- First we underline every other digit, starting with the number's second-to-last digit (which means we start counting number from right to left):
  3<u>7</u>8<u>2</u>82<u>2</u>46<u>3</u>10<u>0</u>0<u>5</u>
- Then we multiply each of the underlined digits by 2:
  7•2 + 2•2 + 2•2 + 4•2 + 3•2 + 0•2 + 0•2

That gives us:
14 + 4 + 4 + 8 + 6 + 0 + 0

- Then we add those products' digits together:
  1 + 4 + 4 + 4 + 8 + 6 + 0 + 0 = 27
- Then we add that sum (27) to the sum of the digits that weren't multiplied by 2:
  27 + 3 + 8 + 8 + 2 + 6 + 1 + 0 + 5 = 60
- Then we check if the total (60) is divisible by 10 with the modulo operator. In this case it is, so this card is legit.

Follow the instructions below:

- Create the *Credit* class that receives an Integer number that represents the credit card number to check, and returns a String with only one of the following values: "INVALID", "AMERICAN EXPRESS", "MASTERCARD", "VISA".
- Create the *CreditTest* class to check all the possible four cases.
- Create additional tests accordingly for every other class that contains logic, to make sure that it does exactly what we expect individually.

Hints:

- *Use the **strategies** to break down the problem into smaller problems.*
- *Try to identify **responsibilities** and create classes for them. If several classes have the same responsibilities you can create abstract classes.*
- *Maybe you would want to abstract the different credit cards logic into a CreditCard abstract class.*
- *Maybe each CreditCard could have a list of Validators to run on a given number, like their length or the numbers they start with.*
- *Take your time with this exercise. It is something that will require thinking a lot before starting to code, but it will prepare you for an incredible future as developer. Don't give up and try your best.*
- *Dealing with integer numbers to do the validations is very complicated. Make sure you transform them to a String at the beginning and you transform them back to numbers only when you need it. Most of the time you will be working with a list of Strings that represent the digits.*
- *Remember that there are two different kind of questions here, whether a card number is valid or not according with the Luhn's algorithm and if it is valid, then which particular card type it is.*
- *It could be that the credit card number is valid, but it's none of the three mentioned credit card types. In that case just return invalid.*
- *To test your algorithm you can use the test credit card numbers that you can find in this website.*
- *This is the first time that you solve a problem like this with abstraction. Don't panic, calm down and just remember everything that we have learnt already. You have all the tools that you need already.*

Exercise 7 [Problem solving]

Derek, Matilda and Hansel want to take the rock, scissors and paper game to the next level. They decide to add the lizard and spock moves.

Game rules:

- Rock crushes scissors and lizard.
- Scissors cuts paper and decapitates lizard.
- Paper covers rock and disproves spock.
- Lizard poisons spock and eats paper.
- Spock smashes scissors and vaporizes rock.

Develop the following code:

- Develop the rock, scissors, paper game if you didn't develop it before.
- Update the rock, scissors, paper *Game* using interfaces with the lizard and spock moves. If your previous implementation was right, you only need to create the new moves, update the logic of the old ones and add the new moves into the *Moves* class. If the one who determines the winner is the *Judge* instead of the moves themselves, then you also have to update it.
- Create an alternative version of the whole game code, replacing the *Move* interface by the *Move* abstract class. It has the *getDefeats* abstract method that returns a list of String with the names of the moves that it defeats. It has the *getName* abstract method. It has the *defeats* method, which is not abstract, that receives

a *Move* and it returns a boolean that says if the name of that move is contained in the list of names of *getDefeats*.

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