

May 9th, 2022

Sheet 05 – Java-API and Recap of Learned Contents

In this exercise sheet we will discuss the role of the class `Object`, mechanics behind wrapper classes, the class `String` and an exercise to recap the topics *associations*, *inheritance*, and *unit testing*. Start by copying the given files from OLAT into your project.

Exercise 1 (Object)

[2 Points]

Given are the files `MyCustomClass1.java` and `Main.java`. `MyCustomClass1` has four attributes, all being of different type and the values are supplied via the constructor. In the main method in `Main.java`, four instances of `MyCustomClass1` are created. This is followed by printing the string representation of the four instances, three comparisons and printing of the hash code of the four instances. Execute the code in `Main.java` and work through the following tasks :

- Why is the string representation of the instances of type `MyCustomClass1` rather cryptic? Explain why that is the case and where this string representation comes from. Put your answers in a file called `e01.txt`.
- Assume that two instances of type `MyCustomClass1` are equal if the values of their attributes are equal. Why do some of the comparisons with `equals()` not show the expected outcome? Put your answers in a file called `e01.txt`.
- Create the file `MyCustomClass2.java` in the same package, use the code of `MyCustomClass1` as a starting point, and uncomment the two code blocks in `Main.java` where `MyCustomClass2` is used. Override the necessary methods from `Object`, such that instances of type `MyCustomClass2` are equal if the corresponding values of the four attributes are equal. Additionally, make sure to create a formatted string representation for `MyCustomClass2`.

Submit



```
at/ac/uibk/pm/gXX/zidUsername/s05/e01/e01.txt
at/ac/uibk/pm/gXX/zidUsername/s05/e01/Main.java
at/ac/uibk/pm/gXX/zidUsername/s05/e01/MyCustomClass1.java
at/ac/uibk/pm/gXX/zidUsername/s05/e01/MyCustomClass2.java
```

Exercise 2 (Wrapper)

[1 Point]

Take a look at the code in `MyCustomClass1.java`^{OLAT} and `Main.java`^{OLAT} ¹. Take note of the datatypes of the attributes of `MyCustomClass1`, the parameters of the constructor, and the values that are supplied when the instances are created. Answer the following questions and write your answers in a file named `e02.txt`

- a) In the main method, where `mcc1a` is created, an integer and a float literal are passed where the constructor expects something of type `Integer` and `Double`, respectively. At the instantiation of `mcc1b`, the `int` variable `int17` is passed where the constructor expects something of type `Integer`. Why does this work? Which mechanism in Java is used to handle such cases?
- b) At the definition of `mcc1c` and `mcc1d` in the main method, a value of type `Integer` is passed where the constructor expects an value of data type `int`. Explain how Java handles this case and which mechanism is used to make this work.

Submit



at/ac/uibk/pm/gXX/zidUsername/s05/e02/e02.txt

Exercise 3 (String)

[2 Points]

The initial situation is that we have some data about persons as comma-separated values². The data is stored in the `String` array named `csv` in `Main.java`^{OLAT}, where each entry in the array holds the data of one individual person. When comparing the structure of the data and the attributes of the classes `Person` and `Address`, we can notice that simply splitting the strings using a comma as separator will not be enough. Solve the following tasks by utilizing existing methods of the datatype `String`³, without editing or changing the data in the array.

- a) Add code to split the data, such that the instances of type `Person` can be created. Use the first loop in `Main.java`^{OLAT} as a starting point.
- b) Notice the strange spelling in the hobbies column and make sure to save them in lowercase letters.
- c) Override the `toString()` method in `Address` and `Person` to have a readable string representation for both classes.

Hint



To successfully extract the hobbies, you will need to extract them as a substring before splitting the individual hobbies.

Submit



at/ac/uibk/pm/gXX/zidUsername/s05/e03/Person.java

at/ac/uibk/pm/gXX/zidUsername/s05/e03/Address.java

at/ac/uibk/pm/gXX/zidUsername/s05/e03/Main.java

¹Code is the same as given for exercise 1.

²https://en.wikipedia.org/wiki/Comma-separated_values

³<https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/lang/String.html>

Exercise 4 (Associations and Inheritance (Recap))

[3 Points]


In this exercise, you will implement a very simplified “Barsimulator” using the principles of Object Oriented Programming.

- a) Create the classes `Bar`, `Person`, `Barkeeper`, `Guest`, `Drink`, `Beer`, and `OrangeJuice`.
- b) A `Bar` has a name, one fixed `Barkeeper`, an array of `Guests`, a maximum number of guests and the current amount of guests. A `Bar` provides methods for `Guests` to enter and leave and for guests to order a drink.
- c) `Drink` is the superclass of `Beer` and `OrangeJuice`, and holds the information about the price (in cent), and an upper- and lower-limit that can be ordered of that drink. `Drink` offers a method to check, if a given order amount is within the upper- and lower-bound, a getter for the price, and a string representation stating the cost in Euro.
- d) `Beer` and `OrangeJuice` have their respective price, upper- and lower-limit, stored as constants and supplied to the constructor of the super class. Both drinks also offer a string representation, returning the name of the drink and the information about how much it costs.
- e) `Person` has a name. `Barkeeper` and `Guest` are both subclasses of `Person`.
- f) `Barkeeper` stores a reference of the bar, the `Barkeeper` works in. `Barkeeper` provides a method to serve a specific amount of a drink to a specific guest. The drink is given by a string, either “beer” or “orangejuice”. The class also offers a string representation that returns “The barkeeper is called ”, followed by the name of the barkeeper.
- g) `Guest` holds a reference to the `Bar` currently visited and also the drink and the current amount of the drink. As simplification, a `Guest` can only have one type of drink at a time. A `Guest` can enter and leave a `Bar` and place an order using either “beer” or “orangejuice” and the desired amount. A `Guest` can also consume a drink if one is available. Instances of type `Guest` are also able to test whether they are equal to another instance of type `Guest`, which is the case if the names of the `Guests` are equal. Lastly, `Guest` also provides a string representation, stating that the instance is a guest and the name of the guest.
- h) Add at least five custom exceptions and throw them where appropriate, e.g., the bar is full and a new guest wants to enter, or a guest wants to consume a drink, but does not hold one.

Submit



- Any .java file of the classes or exceptions you implemented for this exercise. All files submitted for this exercise must be placed in the package

 `at/ac/uibk/pm/gXX/zidUsername/s05/e04/`
or one of its sub-packages.

Exercise 5 (Testing (Recap))


[2 Points]

In this exercise, you will test the functionality of the bar simulator you developed in the previous exercise. Write unit tests using JUnit 5⁴ to test the behavior of your application in the following scenarios:

- Guest enters Bar
- Guest enters Bar twice
- Guest enters a full Bar
- Guest leaves Bar
- Guest, not in Bar, leaves
- Guest orders 1 beer
- Guest orders 999999 beer
- Guest orders -1 beer
- Guest orders a "lizard"
- Guest orders an "ueicbksjdhd"
- Add at least two more unit tests for cases not described above

Submit



 at/ac/uibk/pm/gXX/zidUsername/s05/e05/BarTest.java

Important: Submit your solution to OLAT and mark your solved exercises with the provided checkboxes. The deadline ends at 6:00 pm (18:00) on the day before the discussion.

⁴<https://junit.org/junit5/>