Prof. Dr.-Ing. Ralf Schenkel
Tobias Zeimetz
Trier University

Wintersemester 22/23

Exercises for the Class

Elements of Computer Science: Programming

Assignment 11

Submission of solutions until 3:30 p.m. at 30.01.2023
at `moodle.uni-trier.de`

- Every task needs to be edited in a meaningful way in order to get a point!
- Please comment your solutions, so that we can easy understand your ideas!
- If you have questions about programming or the homeworks, just ask you teachers!
- **Submission that can't be compiled are rated with 0 points!**

Exercise 1 (Abstract classes and exceptions)

25 a) Write an abstract class `Pet` that stores the weight and name of a pet. The class should not have a getter and setter method, but its variables should be declared as **`public`**. Additionally, the following methods should be provided:

- **`void`** `feed(`**`double`** `g)`: The weight of the pet is increased by `g`
- **`void`** `dropWeight(`**`double`** `g)`: The weight of the pet is reduced by `g` due to the daily basal metabolic rate.
- `String toString()`: To be implemented as an abstract method.
- `String makeNoise`: An abstract method that returns a typical sound for the respective animal. See down below for more details about the noises.
- **`int`** `compareTo(Pet pet)`: Is to be implemented as an abstract method to compare two pets by weight. The method shall return a -1 if the weight of **`this`** is less than that of `pet`, 0 if both objects have the same weight and otherwise a 1.

b) Write two more classes `Cat` and `Dog` which extend the class `Pet`

- The name of the animal and a "Meow" or "Woof". Example:

```
Garfield: Meow
Lucky: Woof
```

- The class `Dog` shall include a method for walking a dog. The method has the following signature

  ```
  public void walkTheDog()
  ```

  and has two tasks: (1) the dog should lose 0.2 kg of weight while walking and (2) the method should provide an output of the following form:

  !Lucky goes for a walk and loses weight!

- Familiarize yourself with exception handling by deriving the two classes

  ```
  NotComparableException
  TooHeavyException
  ```

  from the `Exception` class already contained in Java.

  - The `TooHeavyException` is designed to prevent dogs weighing more than 15 kilos from being walked. In this example, they want to eat and sleep all day long! The `TooHeavyException` class has a constructor of the following form:

    ```
    TooHeavyException(double weight)
    ```

    When "throwing" this exception, the overweight of the dog should be calculated and stored in a variable. The class also provides a method, denoted as `String getErrMsg()`, which returns a string of the following form:

    ```
    Exception:  Dogs with overweight don't go
    for walks
    ```

    Extend the method `walkTheDog` so that too heavy dogs "throw" an exception.

  - The `NotComparableException` is intended to prevent that animals of different species are compared with each other (e.g. dog with cat). To do this, implement this class so that it has a constructor of the following form:

    ```
    NotComparableException(Pet pet)
    ```

    When "throwing" this exception a string is to be stored in a variable. This string depends on whether dogs are compared with cats or other different animal species. The string to be stored has accordingly one of the following forms:

    ```
    Exception: You can not compare cats and dogs
    Exception: No comparison possible
    ```

    The class provides, like the exception before, a method with the signature `String getErrMsg()` This method returns the previously saved string.

    Now extend the methods `compareTo(Pet pet)` of the classes `Cat` and `Dog` so that an exception is "thrown" if cats and dogs are compared.

- The method `toString()` shall be overwritten so that a string of the following form is returned

```
Cat: Garfield weighs 4.2 kg
Dog: Lucky weighs 9.8 kg
```

Exercise 2 (Abstract Classes and Interfaces)

25 Model different road users in a hierarchical structure. To do this, implement the following model:

a) An abstract class `Traffic`, from which all other classes are derived:

- Road users have several values like `name` (string), `wheels` (int), `drivenKilometers` (double) and `maxSpeed` (int).
- Write a constructor that receives the expected parameters and writes them into the variables provided.
- A vehicle of class `Traffic` also has an abstract method with the signature

  ```
  abstract boolean licenseNeeded()
  ```

  This method should return whether a driving licence is required depending on the road user (car, motorcycle or bicycle).
- Implement a method with the signature

  ```
  public void addKilometer(int km).
  ```

  This method should be used to increase the number of kilometres travelled.
- Implement another method with the signature `String toString()`.
  This should return a string of the following form:

  ```
  name, drivenKilometers, maxSpeed, wheels
  ```

b) Derive from the class `Traffic` the classes `Car`, `Motorbike` and `Bicycle`

- A **bicycle** has only two wheels and a maximum speed of 30 km/h. Furthermore, a bicycle can be used by a courier. Therefore this class should store a value `carrier` (boolean). Design the constructors of the class so that you do not need to specify a number of tires or a maximum speed.
- Like a bicycle, a **motorbike** has only two wheels, but can travel at any speed (depending on the model, etc.). When implementing the constructor, note that here too, no information about the number of wheels is necessary.
- A **car** has four wheels and can have any maximum speed like a motorcycle.

c) Derive from the class `Car` a class `Oldtimer`. This class stores a value `year` (**`int`**) with the year of manufacture of the car. Design an appropriate constructor.

d) Extend all classes derived from `Traffic` (also oldtimer) with a method denoted as `String toString()` This method should call the `toString()` method of the superclass, but first specify the type of the car. This results in a string of the following form:

```
Car:  name, km, maxSpeed, wheels
```

An oldtimer should call the constructor of the superclass (`Car`) and add the following addition:

```
Car:  name, km, maxSpeed, wheels (Oldtimer)
```

e) Implement two interfaces `Collectible` and `ProtectiveClothing`

- Since vehicles can also always be collectibles, the classes `Car`, `Motorbike` and `Oldtimer` should have the interface `Collectible` The interface has only one method with the signature **`int`** `calcValue()` The value of a collector's item is, for the sake of simplicity, calculated by multiplying the driven kilometers by the number of tires.
- The second interface should contain a method with the signature

```
boolean needsProtectiveClothing()
```

  All classes derived from `Traffic` should have this interface and implement the method `needsProtectiveClothing()`. Cars and oldtimers do not need special protective clothing.

Attention: The tasks are intentionally underspecified. Find meaningful and suitable solutions yourself and familiarize yourself with the `main` method!