Technical University of Denmark
Written examination date: 15 December 2023

Course title: Introductory Programming

Course number: 02314

Aids allowed: All aids allowed

Exam duration: 4 hours

Weighting: 7-point grading scale

Problems:

Problem 1 Water Temperatures . . . 2
  Task 1.1 Printing the Average Temperature . . . 2
  Task 1.2 Printing the Number of Temperature Drops . . . 2
  Task 1.3 Printing Complete Temperature Information . . . 2
Problem 2 Boulder Game . . . 3
  Task 2.1 Creating an Empty Playfield . . . 3
  Task 2.2 Moving the Player in an Empty Playfield . . . 3
  Task 2.3 Pushing Boulders Around . . . 4
Problem 3 Alien Party . . . 5
  Task 3.1 Implementing the Class `Alien` . . . 5
  Task 3.2 Implementing the Class `Gorn` . . . 6
  Task 3.3 Implementing the Class `Cardassian` . . . 6
Problem 4 Movie Collection . . . 7
  Task 4.1 Validating the Movie Title . . . 7
  Task 4.2 Validating the Opinion . . . 7
  Task 4.3 Comparing Two Movies . . . 7

Instructions

1. Each problem in this exam paper has a corresponding **handout (ZIP file) that you must download from DTU Digital Eksamen**. Each problem handout includes:
   - a `./grade` command that you can execute to test your solution, and
   - sometimes, various `.java` files containing test programs and utilities. You should read those files and try to run them, but you must not modify them.
2. You must **submit your solutions on DTU Digital Eksamen**. You can do it **only once**, so submit only when you have completed your work.
3. You must submit **one separate solution file per problem** with **the exact file name required by each problem**. Submit the solution file of one of the problems as "main document" and each solution file for the remaining problems as a separate "attachment."
4. Additional secret tests may be executed on your submissions after the exam.
5. Feel free to add comments to your code.
6. **Suggestions.** Read all problems and tasks before starting your work. Begin by solving the tasks that look easier: feel free to pick tasks that belong to different problems, or proceed out-of-order — *unless a task explicitly says that you must solve another task first.*

Problem 1: Water Temperatures

The lifeguards in Skovshoved have hired you for writing a program that can perform computations on water temperatures. When the program runs, the user can write on the console one or two comma-separated operations (as `String`s) followed by one or more water temperatures (of type `double` and written in the English format, e.g. `1.23`). The program must read inputs until the input stream is closed; then, the program must display on the console one or two results about such water temperatures depending on the specified operations (as explained in the tasks below), and end.

To solve the tasks below, modify the file `WaterTemps.java` provided in the handout.

Submission instructions. When you finish, submit the file `WaterTemps.java`.

Hints. These hints apply to Tasks 1.1, 1.2, and 1.3 below.

- To see examples of the expected console inputs and outputs of each task, run the `./grade` command provided in the handout, and read the reports of the tests of each task.
- Remember that, after creating an object `s` of type `java.util.Scanner`, you can configure its localisation by writing e.g.:

```
s.useLocale(java.util.Locale.ENGLISH);
```

- When you run your program, you can close the standard input stream by pressing Ctrl + D. If that does not work on Windows, try: Ctrl + Z and then ↩.

Task 1.1: Printing the Average Temperature

If the user specifies the operation `average`, your program must print the average of all the temperatures written by the user.

Task 1.2: Printing the Number of Temperature Drops

If the user specifies the operation `drops`, your program must print the number of times the user writes a temperature that is followed by a lower temperature.

Task 1.3: Printing Complete Temperature Information

If the user specifies both operations described in the previous tasks separated by a comma `","`, your program must print the corresponding results in the corresponding order.

Problem 2: Boulder Game

You are implementing a video game that takes place in a playfield of size $n \times m$ cells (with $n, m \geq 1$), where the player can move between cells and push boulders around.

The playfield is represented as an array of arrays of `char` values where each array element represents a cell, and:

- `'.'` represents an empty cell;
- `'X'` represents the unique cell containing the player;
- `'#'` represents a cell containing a boulder.

```
...#..#.
...#....
#..X#.#.
##..#...
```

A playfield with 4 rows and 8 columns is depicted above; the cell at row 0 and column 0 is in the top-left corner.

To solve the tasks below, modify the class `Game` in the file `Game.java` in the handout.

Submission instructions. When you finish, submit the file `Game.java`.

Task 2.1: Creating an Empty Playfield

Implement the method `createplayfield(...)` outlined in the class `Game` in the handout. The method must return an empty playfield with the given number of `rows` and `columns`.

Task 2.2: Moving the Player in an Empty Playfield

Important. Before attempting this task, you must solve Task 2.1.

Implement the method `move(...)` outlined in the class `Game` provided in the handout. The method attempts to move the player in the given `playfield` — which **you can assume to be empty (except for one cell that contains the player)**. The method takes a `direction` as argument, to move the player `up`, `down`, `left`, or `right`. If the player can move in the given `direction`, the method must modify the `playfield` accordingly and return `true`; otherwise, the method must return `false` leaving the `playfield` unchanged.

Hints.
- This method performs checks that are quite similar to Task 2.3 below. With some planning, you could avoid code duplication.
- To see examples of the intended behaviour of the method `move(...)`, run the `./grade` command in the handout and read the reports of the tests for this task.

Task 2.3: Pushing Boulders Around

Important. Before attempting this task, you must solve Task 2.1. Moreover, it may be very helpful to solve Task 2.2 before this task.

This task improves the method `move(...)` described in Task 2.2: the difference is that **now the `playfield` might contain boulders**, and the player can move by pushing a boulder in the given `direction` if the boulder has an empty cell behind.

For instance, consider the following `playfield`:

```
...#..#.
...#....
#..X#.#.
##..#...
```

Calling the method `move(...)` to move the player `up` must return `false` without changing the `playfield`. This is because the boulder in that direction does not have an empty cell behind (along the same direction): therefore, the player cannot push the boulder, hence cannot move `up`.

Instead, calling the method `move(...)` on the `playfield` above to move the player `right` must return `true` and change the `playfield` as follows:

```
...#..#.
...#....
#...X##.
##..#...
```

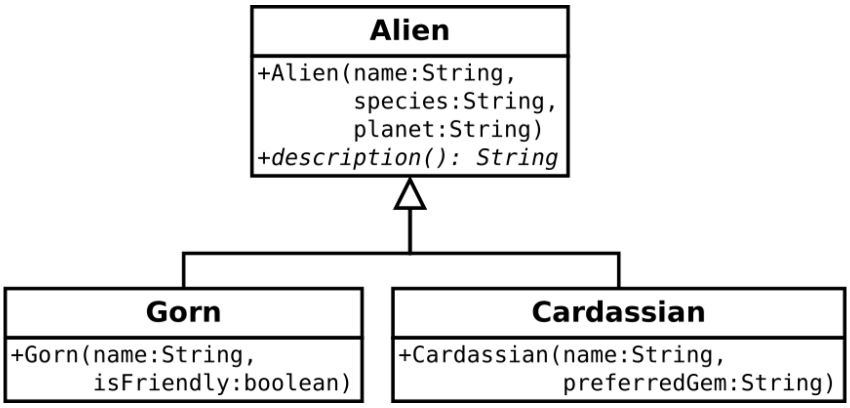
i.e. the player pushes the boulder to the `right` onto the empty cell behind it (along the same direction); also, the player moves onto the cell that was occupied by that boulder.

Now, the player cannot move `right` again: this is because the boulder to the right of the player cannot be pushed, since the cell behind it is not empty. Also, the player cannot move `down`, because there is a boulder that cannot be pushed (since it does not have an empty cell behind). Instead, the player can move `up` or `left`.

Hints.
- This method performs checks that are quite similar to Task 2.2. With some planning, you could avoid code duplication.
- To see more examples of the intended behaviour of the method `move(...)`, run the `./grade` command in the handout and read the reports of the tests for this task.

Problem 3: Alien Party

The United Federation of Planets has asked you to develop a program to manage aliens coming to a party from various planets. Your goal is to implement the classes shown in the following UML diagram (and detailed in the tasks below).


Alien
+Alien(name:String, species:String, planet:String)
+description(): String
Gorn
+Gorn(name:String, isFriendly:boolean)
Cardassian
+Cardassian(name:String, preferredGem:String)

To solve the tasks below, modify the file `Alien.java` in the handout.

Submission instructions. When you finish, submit the file `Alien.java`.

Task 3.1: Implementing the Class `Alien`

Your task is to implement a class called `Alien` in the file `Alien.java`. Each object of the class `Alien` represents an alien individual with a name, species, and planet of origin. The requirements are the following.

- All fields of the class `Alien` must be **`private`** (and you can define all the fields you like). **Note:** the UML diagram above only mentions **`public`** constructors and methods.
- The class `Alien` must provide the constructor outlined in the UML diagram above.

  Hints. To see how the constructor is used, see the file `Task01Test01.java` in the handout.

- The class must provide a method `description()` (outlined in the UML diagram above and in `Alien.java` in the handout) which returns a description of **`this`** alien.

  Hints. To see examples of the description, implement the constructor above, run `./grade` in the handout, and see the reports for the tests of this task.

Task 3.2: Implementing the Class `Gorn`

Important. Before attempting this task, you must solve Task 3.1.

Your task is to implement the class `Gorn`. `Gorn`s come from planet `Arcturus`. The class `Gorn` has the following requirements.

- All fields of the class `Gorn` must be `private` (you can define all the fields you like).
- The class `Gorn` must provide the constructor outlined in the UML diagram above. The argument `isFriendly` tells whether `this` Gorn is friendly or not (because some Gorns are quite aggressive).

  Hints. To see how the constructor is used, see the file `Task02Test01.java` in the handout.

- The method `description()` of the class `Gorn` must tell whether `this` Gorn is friendly or aggressive.

  Hints. To see examples of the description, implement the constructor above, run `./grade` in the handout, and see the reports for the tests of this task.

Task 3.3: Implementing the Class `Cardassian`

Important. Before attempting this task, you must solve Task 3.1.

Your task is to implement the class `Cardassian`. `Cardassian`s come from planet `Bajor`. The class `Cardassian` has the following requirements.

- All fields of the class `Cardassian` must be `private` (you can define all fields you like).
- The class `Cardassian` must provide the constructor outlined in the UML diagram above. The argument `preferredGem` tells which precious gem `this` Cardassian likes the most (because the Cardassians like precious gems).

  Hints. To see how the constructor is used, see the file `Task03Test01.java` in the handout.

- The method `description()` of the class `Cardassian` must tell which gem `this` Cardassian prefers.

  Hints. To see examples of the description, implement the constructor above, run `./grade` in the handout, and see the reports for the tests of this task.

Problem 4: Movie Collection

Your friend Bartolomeus is writing a program for managing his collection of movies and his opinions about them. He has sketched a class `Movie` in the file `Movie.java` and various test programs (provided in the handout), and he needs your help. He wishes to throw an exception if a movie title or opinion is not valid — because right now, the movie title and opinion are not checked, neither by the constructor of a new `Movie` object, nor by its setter methods. Bartolomeus also wants to compare two movies based on his opinion about them.

To solve the tasks below, modify the class `Movie` in the file `Movie.java` in the handout.

Submission instructions. When you finish, submit the file `Movie.java`.

Task 4.1: Validating the Movie Title

Improve the constructor and the method `setTitle(...)` of the class `Movie` to check whether a movie title given as argument is valid.

Hints. To see the requirements for a valid movie title, you need to:
- read the comments that Bartolomeus left in the file `Movie.java` in the handout;
- run `./grade` in the handout and read the reports of the tests for this task.

Task 4.2: Validating the Opinion

Improve the constructor and the method `setOpinion(...)` of the class `Movie` to check whether a movie opinion given as argument is valid.

Hints. To see the requirements for a valid movie opinion, you need to:
- read the comments that Bartolomeus left in the file `Movie.java` in the handout;
- run `./grade` in the handout and read the reports of the tests for this task.

Task 4.3: Comparing Two Movies

The class `Movie` contains a method that must return `true` if the opinion for `this` movie is better than the opinion for the `other` movie. Your task is to implement that method.

Hints. To see which movie opinion is better than the other, you need to:
- read the comments that Bartolomeus left in the file `Movie.java` in the handout;
- run `./grade` in the handout and read the reports of the tests for this task.