Exercise Sheet #9

If you haven't already, read the relevant parts of the course materials and try to answer the questions at the end of the chapter.

book up until page 403 (excluding the chapter Nebenläufigkeit)

Game

Aufgabe 1: Playing field

Write the class `PlayingField`, that represents a two-dimensional landscape. It consists of 40x40 fields (arranged in a square), each of which contains one `char`. Each of these lower-case chars represents an entity:

1. m for a ladybird.
2. l for lice.
3. * for the player (also a louse).
4. s for a sunray.
5. b for a leaf.
6. z for the goal (also a leaf).

Implement the method `String toString()`, which returns a string representation of the whole `PlayingField`. Empty fields should be represented as blanks. When constructing a new `PlayingField` with its constructor, it should contain only empty fields.

Implement the method `void add(int, int, char)` for the `PlayingField` class, that places the given `char` on an empty field with the given position (x- and y-coordinates). Also implement the `equals` method for that class, that checks whether two `PlayingFields` contain the same `chars` on each field.

Test your implementation by changing several field contents on your `PlayingFields` and comparing the `PlayingFields` with `equals`. Don't forget to write comments, conditions and constraints for everything you implement.

Aufgabe 2: Saving

Implement the method `void toFile(String filename)`, that writes the contents of a `PlayingField` to the file with the given filename. Use the same format as `toString()`.

Test your implementation by saving `PlayingFields` to files. Make sure that your code is well documented including conditions and constraints.

Aufgabe 3: Loading

Implement the method `void fromFile(String filename)`, which reads a previously saved `PlayingField` from the file with the given filename.

Test your implementation by loading the previously loaded `PlayingFields` into your program and checking with `equals` that their content hasn't changed during the saving and loading procedure. Make sure that your code is well documented including conditions and constraints.

Aufgabe 4: Moving

Implement the method `void move(int, int, Direction)`, that moves an entity (selected by the coordinates) to the adjoining field (selected by the direction) on the `PlayingField`. `Direction` should be an `enum`. It represents the possible directions in which an entity may move. It is not possible to leave the `PlayingField`.

In case the target field is not empty, use the following rules to determine the result of the move:

i.e, the field that the entity ends up in

1. m (ladybug) eats l and * (lice)
2. l (louse) eats b (leaf)
3. * (special, user-controlled louse) eats b (normal leaves) and z (target leaf)
4. b and z (leaves) absorbs s (sunray)
5. s (sunray) burns m (ladybug), l and * (lice)

If a move isn't covered by these rules, it is impossible. In this case, the `PlayingField` remains unchanged.

Test your implementation by moving entities around on your `PlayingField`. Make sure that your code is well documented including conditions and constraints.

Aufgabe 5: User input

Write the class `Game`, which reads user input from the console in an infinite loop. Each command is sent to the program by pressing return.

Your program should accept commands for:

1. moving the * louse up, down, left and right.

2. saving and loading the current `PlayingField`, where the filename is supplied as an additional command argument. Ensure that * and z exist when loading a `PlayingField`.

3. quiting the game.

Before the game starts a specific number of each entity is placed on the playing field. Note that exactly one player and target leaf must be placed.

After every move of *, every other entity should move around the `PlayingField` randomly. You lose the game if you get eaten by an enemy.

including the target leaf z, excluding * itself

Make sure to place both the user-controlled louse * and the target leaf zon the `PlayingField`. You win the game if your louse * reaches the target leaf z. As always, make sure that your code is well documented including conditions and constraints.

Theory questions

Answer the following questions:

Aufgabe 6: Memory

1. Which two memory areas does every Java program have, and what data is stored in each of them?

2. What is garbage collection used for?

3. How does a garbage collector work?

4. Which pitfalls may we encounter when using garbage collection?

5. How can we influence garbage collection when writing our program?

Aufgabe 7: Files

1. Which errors can happen easily when dealing with files?

2. What happens when we try to read from/write to the same file multiple times?

3. What is character encoding?

4. What are lock files?

Aufgabe 8: Response time

1. What is response time?
2. When does response time increase by an unexpectedly large amount?
3. What is busy waiting?
4. What are the disadvantages of busy waiting?

Aufgabe 9: Integers

1. Why is it usually a bad idea to use floating-point numbers (like `double`) instead of integers (like `int` or `long`) when the integers' range of values is insufficient?
2. What is an over-/underflow?
3. When do we have to use `BigInteger` instead of `int` or `long`?
4. When dealing with integers of arbitrary size, which problems may still arise when using `BigInteger`?

Aufgabe 10: Common errors

1. Which mistakes have you made when writing programs?
2. What are off-by-one errors, and why do they happen?
3. How can we prevent off-by-one errors?
4. Which pitfalls may we encounter when dealing with `null`?