American University of Armenia, CSE
CS120 Intro to OOP A, B, C
Spring 2022

Homework Assignment 9

Due Date: Saturday, April 23 by 23:59 electronically on moodle

This homework assignment builds upon the previous two. It comes with two versions of the code: the HW8 model solution and the HW9 basis. The latter is an extension of the former with some added functionality. Make sure you understand both the HW8 model solution and the changes that are introduced in the HW9 basis.

You are welcome to use your solution to HW8 as a basis for HW9 (but make sure to introduce the added functionality of the HW9 basis) or, alternatively, the HW9 basis accompanying this assignment on Moodle.

You are allowed to write additional methods if you need them during development, but you must not change the headings of any of the specified or existing methods.

Your code should follow good OOP practices and directly apply the principles of abstraction, encapsulation, inheritance and polymorphism.

1. **(3 points)** While all chess pieces are a kind of `Piece`, there should be no objects of the type `Piece` on the actual board. Making the `Piece` class `abstract`, prevents the creation of `Piece` objects. Does it contain any methods that should become abstract too? If so, make them abstract.

   On the other hand, the method `getPieceColor` in `Piece` should never be allowed to be overridden. Add a special keyword to its heading to enforce this rule.

2. **(2 points)** Properly override the `equals` method in the class `Position`.

3. **(15 points)** There are aspects to the program that would clearly benefit from the usage of exceptions. For instance, one of the `Chess` constructors expects a string parameter describing the arrangement of pieces on the board, but the contents of the string may be invalid.

   Introduce an *exception* class `IllegalArgumentException` with two properly defined constructors. Furthermore, derive class `InvalidNumberOfKingsException` from it and, again, include two constructors.

   Inside `Chess`, introduce a method with heading
   `public static void verifyArrangement(String s) throws IllegalArrangementException`
   and implement it to verify that:

   (a) the length of `s` is actually 64,

   (b) there is exactly one white and exactly one black king on the board.

   If the first requirement is violated, the method should throw an `IllegalArgumentException` object. If the second requirement is violated, an `InvalidNumberOfKingsException` object should be thrown.

   Modify the `Chess` constructors to rely on this newly introduced method for arrangement verification instead of performing checks locally.

4. **(10 points)** Reconsider `Chess`'s `getBoard` method. It contains *privacy leaks* since mutable `Piece` objects are referred to in both matrices (original and copy). This problem requires a fix! You need to put *copies* of those pieces in the copy matrix, not the original pieces.

   Doing this with copy constructors would be a tedious job (and very bad OOP style!). You would have to check the actual object type of each `Piece` to know the copy constructor of which class to call—`Rook`, `King`, or anything else.

   Instead, you should rely on the `cloning mechanism` for the `Piece` class hierarchy. Make the `Piece` class `Cloneable` and properly override the `clone` method for it. Some of the derived classes of `Piece`, e.g. `Rook`, have additional instance variables. You need to make those derived classes `Cloneable` and properly override the `clone` method for them too.

   Now that `getBoard` doesn't contain any privacy leaks, you should also introduce the cloning mechanism for the class `Chess`.

5. **(25 points)** We want to change the logic of the interactions between the program and the users. From now on, the program should allow the users to not only play chess, but also to maintain a database of chess puzzles and play those.

   In order to achieve that, we need to introduce a new package `am.aua.chess.puzzles`. The basis of this package is the class `Puzzle`. Every puzzle should have:

   - a 64-character string arrangement of pieces on the board;
   - a turn expressed as a `Chess.PieceColor`;
   - a difficulty level, expressed as an enumerated type for easy, medium, hard puzzles and puzzles with unspecified difficulty;
   - a string description that doesn't contain any line breaks.

   Below is a textual representation of a sample puzzle.

   ```
   rs---------npp-pp--p--pln-pP------p-PP---PS-N---P---BP-P-----S-L,WHITE,MEDIUM
   Additional puzzle that should be added.
   ```

   Here, the first line combines the arrangement, the turn and the difficulty, while the second line represents the description.

   Introduce the `final` class `Puzzle` to contain the members described above. In addition, it should include:

   - a constructor that given the two lines of the textual representation, correctly initialises the puzzle object;
   - a copy constructor;
   - an accessor for each instance variable;
   - `toString` method that generates the textual representation of a puzzle using the format in the sample above;
   - properly overridden `equals` method that relies checks for arrangement, turn and difficulty equivalence, but doesn't include description;

   The first constructor should verify the arrangement of pieces on the board. If the verification fails, the constructor should generate and throw a `MalformedPuzzleException` object. This is yet another exception type that your program should define and use.

   In addition, we want a *natural ordering* to be specified for puzzles. This will enable the sorting of puzzles according to some logic. To this end, make the `Puzzle` class implement the `Comparable<Puzzle>` interface and, hence, include a `compareTo` method.

   The logic of the `compareTo` method should be as follows:

(a) if the difficulties of two puzzles are different, then the easier one comes first;

(b) otherwise, if the turns of the two puzzles are different, then the one with white's turn comes first;

(c) otherwise, if the arrangements of the two puzzles are different, then the one with *lexicographically smaller* arrangement comes first.

This implies that the `compareTo` method should produce the value `0` *iff* the `equals` method produces `true`.

6. (25 points) You may have noticed that the HW9 basis includes a `database.txt` file with textual representations of three puzzles. That file is part of the program now. It stores the database of the puzzles collected by the users so far. Every time the program starts execution, it should load the contents of the file to make those puzzles available to the users for playing. Similarly, at the end of every execution, the set of the puzzles should be saved into the database file. You will soon introduce logic into the program to enable users to list the contents of the puzzle database, select a puzzle to play, or add new puzzles into the database.

In order to provide the functional support for all of this, introduce a class `PuzzleDatabase`. As a static constant, it should maintain the relative path to the `database.txt` file. The only instance variable should be an array of puzzles *without any duplicates*.

The API of the class should consist of:

- a no-arg constructor that loads the puzzles from the database file into the puzzle array;
- a `load` method that loads the puzzles from the database file into the puzzle array;
- a `save` method that saves the puzzles from the array into the database file;
- an `addPuzzlesFromFile` method;
- accessor `getSize` to return the number of puzzles in the database;
- accessor `getPuzzle(int)` to return the puzzle from the database (array) at specified index.

The puzzles may be listed in any order in the database file. However, the loaded set of puzzles in the puzzle array should be ordered according to the *natural ordering* of puzzles. Check out the `sort(Object[]` method from the class `java.util.Arrays` for an easy way to achieve this.

When loading the database from file, two types of problems may arise: difficulties finding/opening the file and malformed puzzle representations. You may want to abort the program if any such problem is detected. Similarly, difficulties creating the file may be detected when saving the database. Handle them in a similar fashion.

The `addPuzzlesFromFile` method, given a filename, reads all the puzzle textual descriptions from that file and, if not already present, adds them to the end of the array of puzzles. You may want to introduce utility methods to help with the implementation. Make sure potential exceptions are handled in `addPuzzlesFromFile` properly. *Note:* unlike `database.txt`, the files used in `addPuzzlesFromFile` don't start with a counter indicating the number of puzzles. So, you need to read puzzle representations until you detect the end of file.

7. (5 points) Produce a UML diagram of the classes inside the `am.aua.chess.puzzles` package. You can do this with a pen on paper and include a photo/scan with your submission.

8. **(5 points)** As a final step, you need to modify the `ChessConsole` class to reflect the introduction of puzzles into the program.

To this end, introduce a second `database` instance variable to represent the puzzle database. Add a no-arg constructor that properly initialises `database`.

The following methods should be included too. Make sure you understand exactly what they do.

```java
public void run() {
    Scanner sc = new Scanner(System.in);
    String inputLine;

    System.out.println("Welcome to Chess Console!");
    printInstructions();
    inputLine = sc.nextLine();
    while (!inputLine.equals("q")) {
        try {
            if (inputLine.equals("p")) {
                game = new Chess();
                play();
            }
            else if (inputLine.equals("l")) {
                int databaseSize = database.getSize();
                for (int i = 0; i < databaseSize; i++)
                    System.out.println(i + ": " + database.getPuzzle(i));
            }
            else if (inputLine.startsWith("a "))
                database.addPuzzlesFromFile(inputLine.substring(2));
            else if (inputLine.startsWith("p ")) {
                int puzzleNumber = Integer.parseInt(inputLine.substring(2));
                Puzzle puzzle = database.getPuzzle(puzzleNumber);
                game = new Chess(puzzle.getArrangement(), puzzle.getTurn());
                play();
            }
            else
                System.out.println("Unknown instruction. Please try again.");
        }
        catch(Exception e) {
            e.printStackTrace();
        }
        printInstructions();
        inputLine = sc.nextLine();
    }
    database.save();
}
private void printInstructions() {
    System.out.println("Input 'p' to play chess.");
    System.out.println("Input 'l' to list the puzzles in the database.");
    System.out.println("Input 'a <filename>' to add new puzzles into the"
            + " database.");
    System.out.println("Input 'p <number>' to play a puzzle.");
    System.out.println("If you want to end the program, input 'q'.");
}
```

Note that `game` creation should now be removed from the `play` method since it is handled in the `run` method.

Finally, modify the `main` method of your program to rely on the `run` method of `chessConsole` instead of the `play` method.