UNIVERSIDADE DA CORUÑA

Software Design

Assignment 1 (2022-2023)

INSTRUCTIONS COMMON TO ALL PRACTICES:

- **Practice groups**
  - The exercises will preferably be done in pairs (they can be done alone but we do not recommend it) and both members of the group will be responsible and should know everything that is delivered on their behalf. We recommend to carry out the practice with techniques of "pair programming".
  - The name of the practice team shall be the name of the practice group to which the members belong (with a "`DS-`"prefix) followed by their corresponding UDC *logins* separated by underscores, e.g.: `DS-12_jose.perez_francisco.garcia`.
  - In the case of belonging to different groups of practices, put the two groups at the beginning, the first group corresponding to the first login, as in the following example: `DS-12-32_jose.perez_francisco.garcia`.
- **Submission**
  - The exercises will be developed using the IntelliJ IDEA tool (Community version) that runs on Java.
  - Exercises will be delivered using the Git version control system using the GitHub Classroom service.
  - We will have an practice class dedicated to explain Git, its use in IntelliJ, GitHub Classroom and how to deliver assignments using this system. Until then you can develop your assignment locally.
  - For the evaluation of the practice we will only take into account those contributions made up to the deadline in the corresponding GitHub Classroom repository, later submissions will not be taken into account.
- **Evaluation**
  - **Important: Plagiarism in an exercise means a grade of zero in the entire practice, both for the original and for the copy.**

INSTRUCTIONS FOR ASSIGNMENT 1:

Deadline: October 07, 2022 (until 23:59).

- **Realization of the assignment**
  - You must push to your repository a specific IntelliJ IDEA project for this assignment. The name of the project will be the team's ID followed by the -A1 suffix. (e.g., `DS-12_jose.perez_francisco.garcia-A1`).
  - You will create a package for each exercise in the assignment, with the following names: `e1`, `e2`, and so on.
  - You must follow the instructions for all exercises closely, as they aim to test your knowledge of particular aspects of Java and object orientation.
- **Testing the correctness of the exercises using JUnit**
  - In this course we use the JUnit 5 framework to check the correctness of the assignments via unit tests.
  - For this first assignment, we will provide the JUnit tests that your exercises must pass in order to be considered valid.
  - **IMPORTANT: You must not modify the tests we have given you**. What you can do is add new tests if you deem them necessary for testing your code (e.g., for increasing the percentage of code coverage in essential parts of the code).
  - We will give a class in which we will explain JUnit and teach you how to run tests and calculate the percentage of code coverage and in the Git seminar we will discuss its integration with GitHub Classroom.
- **Evaluation**
  - This assignment is 1/3 of the final practicum grade.
  - **General evaluation criteria**: the code compiles correctly; there are no runtime errors; specifications have been adhered to; the good programming practices that have been taught in class have been followed; and so forth.
  - **Successfully passing our tests is an important requirement in the evaluation of this assignment**.
  - In addition to the essential evaluation criteria outlined above, each exercise has specific assessment criteria that are stated in the exercise itself.
  - Failing to follow the rules we have set will result in a penalization in the grade.

1. **Utilities for dates**

   Create a class DateUtilities with the following static methods:

```
public class DateUtilities {
    /**
     * Indicates whether a year is a leap year. A leap year is divisible by 4,
     * unless it is divisible by 100, in which case it must be divisible by 400
     * in order to be considered a leap year (e.g., 1900 is not a leap year,
     * but 2000 is) => See the JUnit seminar for an example.
     * @param year the given year
     * @return True if the given year is a leap year, false otherwise.
     */
    public static boolean isLeap(int year) { /* ... */ }

    /**
     * Indicates the number of days of a given month. As the number of days in
     * the month of February depends on the year, it is also necessary to pass
     * the year as an argument.
     * @param month The given month
     * @param year The given year
     * @return The number of days of that month in that year.
     * @throws IllegalArgumentException if the month is not valid.
     */
    public static int numberOfDays(int month, int year) { /* ... */ }

    /**
     * The ISO date format is a standard format that displays the dates
     * starting with the year, followed by the month and the day, i.e.,
     * "YYYY-MM-DD". For example, in that format the text "July 28, 2006"
     * would be represented as "2006-07-28".
     * The "convertToISO" method converts a date in the "Month DD, AAAA"
     * format to its ISO representation. For simplicity, let us assume that
     * the values are correctly formatted even if the date is invalid
     * (e.g., "February 31, 2006" is correctly formatted but it is not a valid date)
     *
     * @param dateText Date in textual format (USA style).
     * @return A string with the given date in ISO format.
     */
    public static String convertToISODate(String dateText) { /* ... */ }

    /**
     * Given a String representing an ISO-formatted date, the methods checks
     * its validity. This includes checking for non-valid characters, erroneous
     * string lengths, and the validity of the date itself (i.e., checking the
     * number of days of the month).
     * @param ISODate A date in ISO format
     * @return True if the ISO-formatted date is a valid date, False otherwise.
     */
    public static boolean checkISODate(String ISODate) { /* ... */ }
```

   Criteria:

   - Managing typical control structures from Java.
   - Managing the class String and its methods.

2. **Social distance in the University**

   Your university has decided to arrange the classrooms' layout with the goal of maintaining the necessary social distance between students. Therefore, each seat has been marked either with an "A", to indicate that it is available, or a ".", to indicate that it must not be used.

   For example, here is the layout of a small classroom, but one not particularly well marked.

```
A . A A .
A A A A A
A . A . A
A A A A .
A . A A .
```

   The students may now sit down freely at the places marked with an "A". However, students prefer to be prudent and follow this rule when they sit down:

   - "Sit down here only if no one is sat down at any of the 8 adjacent seats".

   The students choose where to sit down by looking at the layout of the classroom, and they all sit down at the same time. Thus, in the first iteration all the seats are initially empty and all of them will be occupied, since no one is adjacent to them. The classroom will end up as shown below, where a "#" indicates that the seat is occupied by a student.

```
# . # # .
# # # # #
# . # . #
# # # # .
# . # # .
```

   Once sat, students are reluctant to leave the seat, but if they see too many neighbors at the adjacent seats, they will get up and leave following this rule:

   - "Leave the seat if four or more of its adjacent seats are occupied".

   Therefore, the classroom's layout will end up as follows:

```
# . A A .
# A A A #
A . A . #
# A A A .
# . A # .
```

   The students will continue to sit down and get up following the two rules mentioned above (they sit down if there are no neighbors; they get up if there are four or more neighbors). Keep in mind that all students get up or sit down at the same time. Therefore, our next iteration will be as follows:

```
# . # A .
# A # A #
A . # . #
# A A A .
# . A # .
```

At this point, we see that no student needs to get up — because no one has four or more neighbors — and no one is going to sit down, either — because there are no more empty seats with no neighbors. So this will be the final arrangement for our classroom.

Therefore, this exercise consists in the implementation of the following function:

```java
public class SocialDistance {
    /**
     * Given the layout of a class with available sites marked with an 'A' and
     * invalid sites marked with a '.', returns the resulting layout with the
     * sites occupied by the students marked with a '#' following two rules:
     * - Students occupy an empty seat if there are no other adjacent students.
     * - A student leaves a seat empty if he/she has 4 or more adjacent students.
     * @param layout The initial layout.
     * @return The resulting layout.
     * @throws IllegalArgumentException if the initial layout is invalid (is null,
     * is ragged, includes characters other than '.' or 'A')).
     */
    public static char[][] seatingPeople(char[][] layout) { /* ... */ }
}
```

Criteria:

- Managing *arrays* in Java.
- Managing loops.
- Throwing exceptions.

3. Triangles

Create a **Java record** called `Triangle` with the following methods:

```java
/**
 * Constructs a Triangle object given its three internal angles
 * It is the canonical constructor.
 * @param angle0 Angle 0
 * @param angle1 Angle 1
 * @param angle2 Angle 2
 * @throws IllegalArgumentException if the angles do not sum 180 degrees
 */
public Triangle { /* ... */ }

/**
 * Copy constructor. Constructs a Triangle using another Triangle.
 * @param t The Triangle object to copy.
 */
public Triangle(Triangle t) { /* ... */ }

/**
 * Tests if a triangle is right.
 * A right triangle has one of its angles measuring 90 degrees.
 * @return True if it is right, false otherwise
 */
public boolean isRight() { /* ... */ }

/**
 * Tests if a triangle is acute.
 * A triangle is acute if all angles measure less than 90 degrees.
 * @return True if it is acute, false otherwise
 */
public boolean isAcute() { /* ... */ }

/**
 * Tests if a triangle is obtuse.
 * A triangle is obtuse if it has one angle measuring more than 90 degrees.
 * @return True if it is obtuse, false otherwise
 */
public boolean isObtuse() { /* ... */ }

/**
 * Tests if a triangle is equilateral.
 * A triangle is equilateral if all the angles are the same.
 * @return True if it is equilateral, false otherwise
 */
public boolean isEquilateral() { /* ... */ }

/**
 * Tests if a triangle is isosceles.
 * A triangle is isosceles if it has two angles of the same measure.
 * @return True if it is isosceles, false otherwise
 */
public boolean isIsosceles() { /* ... */ }

/**
 * Tests if a triangle is scalene.
 * A triangle is scalene if it has all angles of different measure.
 * @return True if it is scalene, false otherwise
 */
public boolean isScalene() { /* ... */ }

/**
 * Tests if two triangles are equal.
 * Two triangles are equal if their angles are the same,
 * regardless of the order.
 * @param o The reference object with which to compare.
 * @return True if they are equal, false otherwise.
 */
@Override
public boolean equals(Object o) { /* ... */ }

/**
 * Hashcode function whose functioning is consistent with equals.
```

```
     * Two triangles have the same hashcode if their angles are the same,
     * regardless of the order.
     * @return A value that represents the hashcode of the triangle.
     */
    @Override
    public int hashCode() {/* ... */ }
}
```

Criteria:

- Using Java records.
- Object instantiation.
- Abstraction and encapsulation.
- Exception handling.
- `equals` and `hashCode` contracts.

4. **Movie ratings**

Create an enumeration called `MovieRating` representing the scores assigned by the users of a web site to a film. The specific details of the enumeration are up to you, but must include:

- The following values with their corresponding numerical values: `NOT_RATED`, `AWFUL` (0), `BAD` (2), `MEDIOCRE` (4), `GOOD` (6), `EXCELLENT` (8), `MASTERPIECE` (10).
- The correct constructor for said elements.
- A method called `int getNumericRating()` that returns the numerical value of the rating.
- A public method called `boolean isBetterThan(MovieRating)` that returns `true` if and only if the rating is better than the one that is passed as an argument. Assume that any rating is better than `NOT_RATED`.

N.B.: Your `MovieRating` enumeration should work perfectly with the tests we have given you, without changing them.

Create a class called `Movie` in order to represent films and associated lists of ratings. The specification of the class is as follows:

```java
public class Movie {
    /**
     * Creates a new movie with the list of ratings empty.
     * @param title Movie title.
     */
    public Movie(String title) { /* ... */ }

    /**
     * Returns the movie title
     * @return the movie title.
     */
    public String getTitle() { /* ... */ }

    /**
     * Inserts a new movieRating.
     * It is allowed to insert NOT_RATED.
     * @param movieRating MovieRating to be inserted.
     */
    public void insertRating(MovieRating movieRating) { /* ... */ }

    /**
     * Check if this movie has any rating.
     * @return true if and only if there is a valuation other than NOT_RATED.
     */
    private boolean isRated() { /* ... */ }

    /**
     * Gets the highest rating for this movie.
     * @return  maximum rating; or NOT_RATED if there are no ratings.
     */
    public MovieRating maximumRating() { /* ... */ }

    /**
     * Calculate the numerical average rating of this movie.
     * NOT_RATED values are not considered.
     * @return  Numerical average rating of this movie.
     * @throws java.util.NoSuchElementException if there are no valid ratings.
     */
    public double averageRating() { /* ... */ }
```

Criteria:

- Complex enumerations (constructors, internal state, method overriding, etc.) and associated methods (if necessary).
- The type and design of the enumeration must be decided by the student.