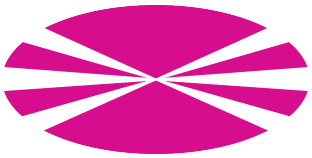

UNIVERSIDADE DA CORUÑA

Software Design

Assignment 1 (2021-2022)

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" that we will explain to you.
  - Each group must register in the wiki available on the virtual campus for this purpose where it will be associated with a certain identifier (e.g. `DS-11-01`).
  - No change of practice group will be made during the course. The group can be undone but its members will continue to carry out the practices alone.
- **Submission**
  - The exercises will be developed using the IntelliJ IDEA tool (Community version) that runs on Java.
  - The exercises will be submitted using the Git version control system on the GitHub service (`https://github.com/`).
  - Each group of students must create a **private repository** for all the assignments in the course, using their group name as an identifier (e.g., `DS-61-01`) and adding its practice professor (or professors) as collaborator. A seminar at the beginning of the course will explain the basic functioning of Git on GitHub and its integration with IntelliJ IDEA.
  - There is a strict deadline for each assignment. Past due submissions will be rejected.
- **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 08, 2021 (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-61-01-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.
- **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. **Word and character count**

   Create a class called `StringCount` with the following specifications:

```java
public class StringCount {
    /**
    * Counts the number of words in a given String.
    * Words are groups of characters separated by one or more spaces.
    *
    * @param text String with the words
    * @return Number of words in the String or zero if it is null
    */
    public static int countWords(String text) { /* ... */ }

    /**
    * Counts the number of times the given character appears in the String.
    * Accented characters are considered different characters.
    * @param text String with the characters
    * @param c the character to be found
    * @return Number of times the character appears in the String or zero if null
    */
    public static int countChar(String text, char c) { /* ... */ }

    /**
    * Counts the number of times the given character appears in the String.
    * The case is ignored so an 'a' is equal to an 'A'.
    * Accented characters are considered different characters.
    * @param text String with the characters
    * @param c the character to be found
    * @return Number of times the character appears in the String or zero if null
    */
    public static int countCharIgnoringCase(String text, char c) { /* ... */ }

    /**
    * Checks if a password is safe according to the following rules:
    * - Has at least 8 characters
    * - Has an uppercase character
    * - Has a lowercase character
    * - Has a digit
    * - Has a special character among these: '?',  '@', '#', '$', '.' and ','
    * @param password The password, we assume it is not null.
    * @return true if the password is safe, false otherwise
    */
    public static boolean isPasswordSafe(String password) { /* ... */ }
}
```

   Criteria:

   - Managing typical control structures of Java.
   - Managing the class `String` and its methods.
   - Managing utility classes such as `Character`.

2. **Skiing down slopes**

In a ski resort, the slopes that are outside the piste are full of trees. We have mapped one of the slopes and the result is shown below (a "." dot represents a clear space and a "#" hash mark represents a tree).

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

We have a skier willing to go down one of these slopes. The skier is good but a bit limited, in the sense that, after choosing a descension strategy, the skier follows the strategy no matter how many trees are encountered along the way.

When it comes to executing the descension strategies, we must take into account the following:

- The descension strategies are defined by two numbers (`right` and `down`) that define the movement of the skier in the map (rightwards and downwards).
- The skier always starts at the top-left corner and finishes when the last row is reached.
- The skier always executes the `right` movement first, and the `down` movement after that.
- When the skier reaches the rightmost limit of the map, the skier continues by going to the leftmost limit of the same row (as if the map were infinite and constantly repeated the same pattern).

The skier decides to try a strategy (`right=3` and `down=1`) to go down the slope. The path for that strategy is shown below. The initial/final points of the movement are marked by "`[]`" square brackets and the intermediate points are marked by "`()`" round brackets. As can be seen, the rightwards move is always executed first and, when the last column is reached, the skier continues from the fist column of the same row. The result is as follows:

```
[.]  (.)  (#)  (#)   .    .    .    .    .    .    .
 #    .    .   [.]  (#)  (.)  (.)   .    #    .    .
 .    #    .    .    .    .   [#]  (.)  (.)  (#)   .
(.)  (.)   #    .    #    .    .    .    #   [.]  (#)
 .   [#]  (.)  (.)  (.)   #    #    .    .    #    .
 .    .    #    .   [#]  (#)  (.)  (.)   .    .    .
 .    #    .    #    .    #    .   [.]  (.)  (.)  (#)
(.)  (#)  (.)   .    .    .    .    .    .    .   [#]
 #    .   [#]  (#)  (.)  (.)   .    #    .    .    .
 #    .    .    .    #   [#]  (.)  (.)  (.)   .    #
(.)   #    .    .    #    .    .    .   [#]  (.)  (#)
```

The skier has bumped into 17 trees along the way. The skier decides to change the strategy and follow one based on *jumps*. That is, the movement is as before (right and down) except that it is done in jumps. Therefore, when it comes to crashing into a tree, we will only consider the trees that are at the initial and final points of the movement. Doing it like this, the skier will only encounter 7 trees, as shown below:

```
[.]   .    #    #    .    .    .    .    .    .    .
 #    .    .   [.]   #    .    .    .    #    .    .
 .    #    .    .    .    .   [#]   .    .    #    .
 .    .    #    .    #    .    .    .    #   [.]   #
 .   [#]   .    .    .    #    #    .    .    #    .
 .    .    #    .   [#]   #    .    .    .    .    .
 .    #    .    #    .    #    .   [.]   .    .    #
 .    #    .    .    .    .    .    .    .    .   [#]
 #    .   [#]   #    .    .    .    #    .    .    .
 #    .    .    .    #   [#]   .    .    .    .    #
 .    #    .    .    #    .    .    .   [#]   .    #
```

Write the code for the functions `downTheSlope()` and `jumpTheSlope()`:

```java
/**
* Traverses the slope map making the right and down movements and
* returns the number of trees found along the way.
* @param slopeMap A square matrix representing the slope with spaces
*                 represented as "." and trees represented as "#".
* @param right Movement to the right
* @param down Downward movement
* @return Number of trees
* @throws IllegalArgumentException if the matrix is incorrect because:
*     - It is not square.
*     - It has characters other than "." and "#"
*     - right >= number of columns or right < 1
*     - down >= number of rows of the matrix or down < 1
*/
public static int downTheSlope(char[][] slopeMap, int right, int down) { /* */}

/**
* Traverses the slope map making the right and down movements and
* returns the number of trees found along the way.
* Since it "jumps" from the initial position to the final position,
* only takes into account the trees on those initial/final positions.
*
* Params, return value and thrown expections as in downTheSlope...
*/
public static int jumpTheSlope(char[][] slopeMap, int right, int down) { /* */}
```

Criteria: Managing *arrays* in Java.

3. Melodies and notes

A melody is a sequence of notes with a duration for each one of them. A **note** consists of its name, that is, ***do***, ***re***, ***mi***, ***fa***, ***sol***, ***la*** and ***si*** (often represented in English-language regions as *C*, *D*, *E*, *F*, *G*, *A* and *B*, respectively) and an **accidental**, which can be **natural** (no alteration nor symbol), **sharp** (♯) or **flat** (♭).

A melody is constructed starting from nothing and adding notes (with their alterations and durations) sequentially.

Bearing this in mind, write the code for the class `Melody` following the specifications stated below:

```
public class Melody {

    /**
    * Creates an empty Melody instance.
    */
    public Melody() { /* ... */ }

    /**
    * Add a note at the end of this melody.
    * @param note The note to add
    * @param accidental The accidental of the note
    * @param time The duration of the note in milliseconds
    * @throws IllegalArgumentException if the note, the accidental
    * or the time are not valid values.
    */
    public void addNote(Notes note, Accidentals accidental, float time) {/* ... */}

    /**
    * Returns the note on the given position
    * @param index The position of the note to get.
    * @return The note on index.
    * @throws IllegalArgumentException if the index is not a valid position.
    */
    public Notes getNote(int index) { /* ... */ }

    /**
    * Returns the accidental of the note on the given position
    * @param index The position of the accidental to get.
    * @return The accidental on index.
    * @throws IllegalArgumentException if the index is not a valid position.
    */
    public Accidentals getAccidental(int index) { /* ... */ }

    /**
    * Returns the duration of the note on the given position
    * @param index The position of the time to get.
    * @return The time on index.
    * @throws IllegalArgumentException if the index is not a valid position.
    */
    public float getTime(int index) { /* ... */ }

    /**
    *  Returns the number of notes in this melody.
    * @return The number of notes in this melody.
    */
    public int size() { /* ... */ }

    /**
    * Returns the duration of this melody.
    * @return The duration of this melody in milliseconds.
    */
    public float getDuration() { /* ... */ }

    /**
    * Performs the equality tests of the current melody with another melody
    * passed as a parameter. Two melodies are equal if they represent the same
    * music fragment regardless of the name of its notes.
    * @param o The melody to be compared with the current melody.
```

```
    * @return true if the melodies are equals, false otherwise.
    */
    @Override
    public boolean equals(Object o) { /* ... */ }

    /**
    * Returns an integer that is a hash code representation of the melody.
    * Two melodies m1, m2 that are equals  (m1.equals(m2) == true) must
    * have the same hash code.
    * @return The hash code of this melody.
    */
    @Override
    public int hashCode() { /* ... */ }

    /**
    * The string representation of this melody.
    * @return The String representantion of this melody.
    */
    @Override
    public String toString() { /* ... */ }
}
```

You must consider the fact that two notes with different names and alterations can be actually the same note. The equivalences are as follows:

- Do ♯ = Re ♭
- Re ♯ = Mi ♭
- Mi = Fa ♭
- Mi ♯ = Fa
- Fa ♯ = Sol ♭
- Sol ♯ = La ♭
- La ♯ = Si ♭
- Si = Do ♭
- Si ♯ = Do

This way, the melodies represented by the notes DO♯ MI SOL and RE♭ FA♭ SOL are actually equal (assuming that the corresponding durations are also the same).

The textual representation of a melody must use, for each note, the following format: {`Name`}{`Accidental`}{(`Duration`)}. The notes in the melody are separated by a single space. For example, the melody DO♯ MI SOL♭ with durations 2.0, 1.0, and 3.5 (respectively) will have the following representation: `Melody: DO#(2.0) MI(1.0) SOLb(3.5)`.

Optionally, and if deemed necessary, you may add other classes.

Criteria:

- Object instantiation.
- Abstraction and encapsulation.
- Exception handling.
- Collections of elements.
- Simple enumerations.
- `equals` and `hashCode` contracts.

4. **Batch calculator**

   You must implement a calculator that works in batches. That is, you...

   - Add the desired operations.
   - Execute them and get the results.

   The calculator works by accumulating results and does not have a memory. Therefore, the first operation will only have two operands. For the next operations, the first operand will be the accumulated value from the previous operation, which means that you only need a single additional operand.

   You must write the code for the class `Calculator`, with the following **public** methods (you may add other private methods considered useful for your implementation.):

```java
public class Calculator {
    /**
     * Public constructor of the calculator.
     */
    public Calculator() { /* ... */ }

    /**
     * Clean the internal state of the calculator
     */
    public void cleanOperations() { /* ... */ }

    /**
     * Add a new operation to the internal state of the calculator.
     * It is worth mentioning that the calculator behaves in an accumulative way,
     * thus only first operation has two operands.
     * The rest of computations work with the accumulated value and only an extra
     * new operand. Second input value must be ignored if the operation does not
     * correspond to the first one.
     *
     * @param operation operation to add, as string, "+", "-", "*", "/".
     * @param values    Operands of the new operation (one or two operands).
     *                  Uses the varargs feature.
     * https://docs.oracle.com/javase/8/docs/technotes/guides/language/varargs.html
     * @throws IllegalArgumentException If the operation does not exist.
     */
    public void addOperation(String operation, float... values) { /* ... */ }

    /**
     * Execute the set of operations of the internal state of the calculator.
     * Once execution is finished, internal state (operands and operations)
     * is restored (EVEN if exception occurs).
     * This calculator works with "Batches" of operations.

     * @return result of the execution
     * @throws ArithmeticException If the operation returns an invalid value
     *                                  (division by zero)
     */
    public float executeOperations() { /* ... */ }

    /**
     * Current internal state of calculator is printed
     * FORMAT:
     * "[{+/-/"/"/*}]value1_value2[{+/-/"/"/*}]value1[{+/-/"/"/*}]value1{...}"
     * EXAMPLES: JUnit tests

     * @return String of the internal state of the calculator
     */
    @Override
    public String toString() { /* ... */ }
}
```

Criteria:

- Complex enum types (constructors, internal state, overridden methods, and so on) and associated methods (if necessary).
- The specific types and design of the enum types are up to you.