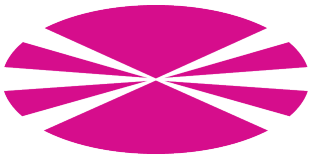

UNIVERSIDADE DA CORUÑA

Diseño Software

Boletín de Ejercicios 1 (2021-2022)

INSTRUCCIONES COMUNES A TODAS LAS PRÁCTICAS:

- **Grupos de prácticas**
  - Los ejercicios se realizarán preferentemente por parejas (pueden hacerse en solitario pero no lo recomendamos) y ambos miembros del grupo serán responsables y deben conocer todo lo que se entregue en su nombre.
  - Recomendamos realizar la práctica con técnicas de "programación en pareja" (*pair programming*) que os explicaremos.
  - Cada grupo deberá inscribirse en el wiki disponible en el campus virtual en donde se asociará cada grupo a un determinado identificador (p.ej. `DS-11-01`).
  - No se pondrá cambiar de grupo de prácticas durante la asignatura, todo lo más se podrá deshacer el grupo y sus miembros continuarán realizando las prácticas en solitario.
- **Entrega**
  - Los ejercicios serán desarrollados mediante la herramienta IntelliJ IDEA (versión *Community*) que se ejecuta sobre Java.
  - Los ejercicios se entregarán usando el sistema de control de versiones Git en el servicio GitHub (`https://github.com/`).
  - Cada grupo deberá crear un **repositorio privado** para todas las prácticas de la asignatura usando como nombre su identificador (p.ej. `DS-11-01`) y añadiendo como colaborador a su profesor (o profesores) de prácticas. Os explicaremos en un seminario el manejo básico de Git en GitHub y su uso en IntelliJ IDEA.
  - Para la evaluación de la práctica sólo tendremos en cuenta aquellas contribuciones hechas hasta la fecha de entrega de la misma, los envíos posteriores no serán tenidos en cuenta.
- **Evaluación**
  - **Importante: Si se detecta algún ejercicio copiado en una práctica, ésta será anulada en su totalidad (calificación cero), tanto el original como la copia**.

INSTRUCCIONES BOLETÍN 1:

Fecha límite de entrega: 08 de octubre de 2021 (hasta las 23:59).

- **Realización del boletín**
  - Se deberá subir al repositorio un único proyecto IntelliJ IDEA para el boletín con el nombre del grupo de prácticas más el sufijo -B1 (por ejemplo `DS-11-01-B1`).
  - Se creará un paquete por cada ejercicio del boletín usando los siguientes nombres: `e1`, `e2`, etc.
  - Es importante que sigáis detalladamente las instrucciones del ejercicio, ya que persigue el objetivo de probar un aspecto determinado de Java y la orientación a objetos.
- **Comprobación de la ejecución correcta de los ejercicios con JUnit**
  - En la asignatura usaremos el framework JUnit 5 para comprobar, a través de pruebas, que el funcionamiento de las prácticas es el correcto.
  - En este primer boletín os adjuntaremos los tests JUnit que deben pasar los ejercicios para ser considerados válidos.
  - **IMPORTANTE: No debéis modificar los tests que os pasemos**. Sí podéis añadir nuevos tests si consideráis que quedan aspectos importantes por probar en vuestro código (por ejemplo, que su cobertura sea baja en partes fundamentales).
  - En el seminario de JUnit os daremos información detallada de como ejecutar los tests y calcular la cobertura de los mismos.
- **Evaluación**
  - Este boletín corresponde a 1/3 de la nota final de prácticas.
  - **Criterios generales**: que el código compile correctamente, que no de errores de ejecución, que se hayan seguido correctamente las especificaciones, que se hayan seguido las buenas prácticas de la orientación a objetos explicadas en teoría, etc.
  - **Pasar correctamente nuestros tests es un requisito importante en la evaluación de este boletín**.
  - Aparte de criterios fundamentales habrá criterios de corrección específicos que detallaremos en cada ejercicio.
  - No seguir las normas aquí indicadas significará una penalización en la nota.

1. **Conteo de palabras y caracteres**

   Crea una clase `StringCount` que cumpla las siguientes especificaciones:

```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) { /* ... */ }
}
```

   Criterios:

   - Manejo de estructuras típicas de control de Java.
   - Manejo de la clase `String` y sus métodos.
   - Manejo de clases de utilidades como `Character`.

2. **Descenso de pendientes**

En una estación de esquí las pendientes que están fuera de pista están plagadas de árboles. Hemos hecho el mapeo de una de estas pendientes y el resultado es el siguiente (donde un punto "`.`" representa un espacio libre y una almohadilla "`#`" representa un árbol).

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

Tenemos un esquiador dispuesto a bajar una de estas pendientes. Nuestro esquiador es habilidoso pero un poco limitado, ya que una vez que decide una estrategia de descenso sigue con ella sin importar los árboles que se encuentre por el camino.

A la hora de ejecutar las estrategias de descenso tenemos que tener en cuenta que:

- Las estrategias de descenso se definen por dos números (`right` y `down`) que definen el movimiento que hace el esquiador hacia la derecha y hacia abajo en el mapa.
- El esquiador siempre empieza en la esquina superior izquierda y acaba en el momento en que abandona la última fila.
- El esquiador siempre hace primero el movimiento `right` y luego el movimiento `down`.
- Cuando el esquiador llega al límite derecho del mapa sigue por el izquierdo en la misma fila (como si el mapa fuese un mapa infinito que repite el mismo patrón constantemente).

Nuestro esquiador decide probar con una estrategia (`right=3` y `down=1`) para bajar la pendiente. A continuación mostramos el recorrido que hace, marcando los puntos iniciales/finales del movimiento con corchetes "`[]`" y los puntos intermedios con paréntesis "`()`". Como vemos siempre hace primero el movimiento a la derecha y, en caso de llegar a la última columna, continúa por la primera columna dentro de la misma fila. El resultado es el siguiente:

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

Nuestro esquiador se ha encontrado con 17 árboles en el camino. Decide cambiar la estrategia y seguir una de *saltos*, es decir, el movimiento será igual que antes (primero a la derecha y luego abajo) pero se moverá ejecutando un salto, por lo que solo tendremos en cuenta, a la hora de chocar con los árboles, aquellos que se encuentren en los puntos iniciales y finales del recorrido. De esta forma solo se encontrará con 7 árboles, como se muestra a continuación:

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

Implementa las funciones `downTheSlope()` y `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) { /* */}
```

Criterios: Manejo de *arrays* en Java.

3. Melodías y notas

Una melodía es una sucesión de notas con un tiempo de duración para cada una de ellas. Una **nota** se compone de su nombre principal (*do*, *re*, *mi*, *fa*, *sol*, *la* y *si*) y de una **alteración** (*accidental*, en inglés), que puede ser **natural** (no hay alteración ni símbolo), **sostenido** (♯ o *sharp*, en inglés) o **bemol** (♭ o *flat*, en inglés).

Una melodía se construye partiendo de cero y añadiendo notas (con una alteración y una duración) de manera secuencial.

Teniendo en cuenta esto, crea una clase `Melody` que cumpla con las especificaciones indicadas a continuación:

```
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() { /* ... */ }
}
```

Es necesario tener en cuenta que algunas notas con nombre y alteraciones diferentes son en realidad la misma nota. Estas son las equivalencias:

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

De esta manera, y por ejemplo, las melodías formadas por las notas DO♯ MI SOL y RE♭ FA♭ SOL son iguales (suponiendo que las duraciones de las notas equivalentes también sean iguales).

La representación textual de una melodía debe seguir, para cada una de las notas, el formato siguiente: {`Nombre`}{`Alteración`}{(`tiempo`)}. Cada nota de la melodía irá separada por un espacio de la siguiente. Por ejemplo, la melodía DO♯ MI SOL♭ con duraciones 2.0, 1.0, y 3.5 respectivamente tendrá la siguiente representación: `Melody: DO#(2.0) MI(1.0) SOLb(3.5)`.

Opcionalmente, y en caso de considerarlo necesario, se pueden añadir otras clases.

Criterios:

- Instanciación de objetos.
- Abstracción y encapsulamiento.
- Manejo de excepciones.
- Colecciones de elementos.
- Tipos enumerados sencillos.
- Contratos del `equals` y el `hashCode`.

4. **Calculadora por lotes**

   Se quiere implementar una calculadora que funcione por lotes, esto es:

   - Se añaden las operaciones deseadas.
   - Se ejecutan y obtienen los resultados.

   La calculadora funciona acumulando resultados y no tiene memoria. Por lo tanto, únicamente se tendrán 2 operandos para la primera operación. Para las siguientes, se utilizará el valor acumulado de la operación previa como primer operando, por lo que necesitaremos un único operando adicional.

   Se pide implementar la clase `Calculator`, con los siguientes métodos **públicos** (se pueden añadir otros métodos privados si los consideráis útiles para vuestra implementación):

```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() { /* ... */ }
}
```

Criterios:

- Tipos enumerados complejos (constructores, estado interno, sobrescritura de métodos, etc.) y métodos asociados (si fueran necesarios).
- Los tipos y el diseño de los enumerados dependen del alumno.