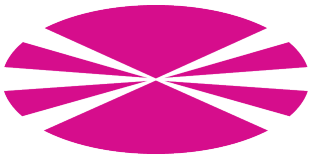

UNIVERSIDADE DA CORUÑA

Diseño Software

Boletín de Ejercicios 1 (2020-2021)

INSTRUCCIONES COMUNES A TODAS LAS PRÁCTICAS:

- **Estructura de los ejercicios**
  - Los ejercicios se realizarán por parejas de alumnos. Si algún alumno tiene algún problema para establecer una pareja que se lo comunique a su profesor de prácticas e intentará buscarle un compañero. Mantened el grupo de prácticas para todas las prácticas de la asignatura.
  - Cada pareja deberá inscribirse en el wiki disponible en el campus virtual en donde se asociará a un determinado identificador (p.ej. `DS-11-01`).
  - Los ejercicios serán desarrollados mediante la herramienta IntelliJ IDEA (versión *Community*) que se ejecuta sobre Java.
- **Entrega de los ejercicios**
  - Los ejercicios se entregarán usando el sistema de control de versiones Git en el servicio GitHub (`https://github.com/`).
  - Cada pareja deberá crear un **repositorio privado** para todas las prácticas de la asignatura usando como nombre su identificador (`DS-11-01`) y añadiendo como colaboradores a los profesores de la asignatura. 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: Todo ejercicio en el que haya sospechas fundadas de copias entre grupos implicará la anulación de su nota (tanto para el grupo que lo haya desarrollado originalmente como para el que lo haya copiado). Si la copia es flagrante y/o repetida puede llevar a la anulación de todo el boletín o práctica**.
  - **Criterios generales a evaluar son**: 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, que se hayan seguido las instrucciones para su realización, etc.

INSTRUCCIONES BOLETÍN 1:

Fecha límite de entrega: 23 de octubre de 2020 (hasta las 23:59).

- **Realización del boletín**
  - Se deberá crear 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.
  - En el Campus Virtual existe un ejemplo de proyecto con varios ejercicios resueltos que os puede servir de plantilla para crear vuestro proyecto
- **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 cómo ejecutar los tests y calcular la cobertura de los mismos.
- **Evaluación**
  - Este boletín corresponde a 1/3 de la nota final de prácticas.
  - **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. **Mezclas de Strings**

   Dados dos *Strings* `A` y `B`, consideramos que un *String* `C` es una mezcla válida de `A` y `B` si cumple que:

   - `C` contiene todos los caracteres que están en `A` y en `B`.
   - El orden en el que los caracteres aparecen en `A` y en `B` se preserva en `C`.

   Por ejemplo, si `A` es "`Bye`" y `B` es "`World`" los siguientes ejemplos serían considerados **mezclas válidas**: `ByeWorld`, `WorldBye`, `BWyoerld`, `ByWorled`.

   Y los siguientes ejemplos serían considerados **mezclas NO válidas**: `eyBdlroW`, `ByeWorlds`, `ByeWordl`, `HelloWorld`, `ByWorl`, `BByyeeWWoorrlldd`.

   Dicho esto, implementa las funciones que se detallan a continuación. Para simplificar las mismas podremos suponer lo siguiente:

   - Los Strings `a` y `b` que se pasan por parámetro no tienen ningún carácter en común.
   - Todos los *Strings* pasados a las funciones (`a`, `b` y `c`) son *Strings* no nulos que contienen al menos un carácter imprimible.
   - La comparación es *case-sensitive*, es decir, el carácter "`a`" es distinto al carácter "`A`".

```java
public class StringUtilities {
    /**
     * Checks if a given string is a valid c of two others. That is, it contains
     * all characters of the other two and order of all characters in the individual
     * strings is preserved.
     * @param a First String to be mixed
     * @param b Second String to be mixed
     * @param c Mix of the two other Strings
     * @return true if the c is valid, false otherwise
     */
    public static boolean isValidMix(String a, String b, String c) { /*...*/ }

    /**
     * Generates a random valid mix for two Strings
     * @param a First String to be mixed
     * @param b Second String to be mixed
     * @return A String that is a random valid mix of the other two.
     */
    public static String generateRandomValidMix(String a, String b) { /*...*/ }
}
```

   Criterios:

   - Manejo de estructuras típicas de control de Java.
   - Manejo de la clase `String`, la clase `StringBuilder` y los caracteres en Java.

2. **Código en un teclado alfanumérico**

Estamos diseñando un sistema de seguridad que consiste en teclear una serie de numeros en un teclado alfanumérico. El teclado tiene que cumplir los siguientes requisitos:

- Tener forma rectangular.
- El número en la esquina superior izquierda es el `1`.
- Los números siguen una secuencia bien por filas y luego columnas, o por columnas y luego filas.
- La secuencia que siguen es: primero los números (`1, 2, 3, .., 0`) y luego las letras mayúsculas usando el alfabeto inglés sin `Ñ`: (`A, B, .., Z`).

A continuación se muestran varios teclados válidos:

1	2	3
4	5	6
7	8	9

1	4	7	0	C
2	5	8	A	D
3	6	9	B	E

Para saber qué número hay que introducir en el teclado, en vez de decírselo directamente a la persona interesada, le diremos una secuencia de movimientos que tiene que hacer sobre el teclado partiendo de la esquina superior izquierda para llegar al número de la clave.

Los movimientos pueden ser arriba (`U`), abajo (`D`), izquierda (`L`) y derecha (`R`). Por ejemplo, si nuestra clave tiene cuatro dígitos le pasaremos los siguientes movimientos para cada uno de los números: `RD, DRUU, LLD, D`.

Estos movimientos sobre el primer teclado dan la clave `5347` de la siguiente forma:

- Empezamos en `1`, nos movemos a la derecha (`R`) y abajo (`D`) y llegamos al `5`.
- Partiendo del `5`, nos movemos abajo (`D`), a la derecha (`R`), arriba (`U`) y arriba (`U`) y llegamos al `3`.
- Partiendo del `3`, nos movemos a la izquierda (`L`), a la izquierda (`L`) y abajo (`D`) y llegamos al `4`.
- Partiendo del `4`, nos movemos abajo (`D`) y llegamos al `7`.

Si algún movimiento nos lleva fuera del teclado ignoraremos ese movimiento. De esta forma los movimientos `ULL, RRDDD, LURDL` sobre el primer teclado dan la clave de tres dígitos `198` de la siguiente forma:

- Empezamos en `1`, movimiento `U` ignorado, movimiento `L` ignorado, movimiento `L` ignorado, nos quedamos en `1`.
- Partiendo de `1`, nos movemos a la derecha (`R`), a la derecha (`R`), abajo (`D`), abajo (`D`) e ignoramos el último `D` llegando a `9`.
- Partiendo de `9`, nos movemos a la izquierda (`L`), arriba (`U`), a la derecha (`R`), abajo (`D`) y a la izquierda (`L`) llegando a `8`.

Dada esta definición del problema implementa los siguientes métodos:

```
public class Code {

    /**
     * Determines whether a keypad is valid. To do this, it must be a rectangle and
     * the numbers must follow the alphanumeric sequence indicated. If the array
     * (or any of its subarrays) is null it will be returned false.
     * @param keypad The keypad to be analyzed.
     * @return true if it is valid, false otherwise.
     */
    public static boolean isKeypadValid(char[][] keypad) { /* ... */  }

    /**
     * Checks if an array of movements is valid when obtaining a key on a keypad.
     * An array of movements is valid if it is formed by Strings that only have the
     * four capital letters U, D, L and R. If the array of Strings or any of the
     * Strings is null it will be returned false.

     * @param movements Array of Strings representing movements.
     * @return true if it is valid, false otherwise.
     */
    public static boolean areMovementsValid(String[] movements) { /* ... */ }

    /**
     * Given a keypad and an array of movements, it returns a String with the code
     * resulting from applying the movements on the keypad.
     * @param keypad alphanumeric keypad.
     * @param movements Array with the different movements to be made for each
                        number of the key.
     * @return Resulting code.
     * @throws IllegalArgumentException if the keypad of the movements are invalid.
     */
    public static String obtainCode(char[][] keypad, String[] movements) {/* ... */}
}
```

Criterios:

- Manejo de estructuras típicas de control de Java.
- Manejo de *arrays* y matrices en Java.

3. **Relojes**

Crea una clase `Clock` que cumpla las especificaciones que se detallan a continuación:

```
public class Clock {
    /**
     * Creates a Clock instance parsing a String.
     * @param s The string representing the hour in 24h format (17:25:15) or in
     *          12h format (05:25:15 PM).
     * @throws IllegalArgumentException if the string is not a valid hour.
     */
    public Clock(String s) { /*...*/ }

    /**
     * Creates a clock given the values in 24h format.
     * @param hours Hours in 24h format.
     * @param minutes Minutes.
     * @param seconds Seconds.
     * @throws IllegalArgumentException if the values do not represent a valid
     * hour.
     */
    public Clock(int hours, int minutes, int seconds) { /*...*/ }

    /**
     * Creates a clock given the values in 12h format. Period is a enumeration
     * located inside the Clock class with two values: AM and PM.
     * @param hours Hours in 12h format.
     * @param minutes Minutes.
     * @param seconds Seconds.
     * @param period Period used by the Clock (represented by an enum).
     * @throws IllegalArgumentException if the values do not represent a valid
     * hour.
     */
    public Clock(int hours, int minutes, int seconds, Period period) { /*...*/ }


    /**
     * Returns the hours of the clock in 24h format
     * @return the hours in 24h format
     */
    public int getHours24() { /*...*/ }

    /**
     * Returns the hours of the clock in 12h format
     * @return the hours in 12h format
     */
    public int getHours12() { /*...*/ }

    /**
     * Returns the minutes of the clock
     * @return the minutes
     */
    public int getMinutes() { /*...*/ }

    /**
     * Returns the seconds of the clock.
     * @return the seconds.
     */
    public int getSeconds() { /*...*/ }

    /**
     * Returns the period of the day (AM or PM) that the clock is representing
     * @return An instance of the Clock.Period enum depending if the time is
     * before noon (AM) or after noon (PM).
     */
    public Period getPeriod() { /*...*/ }

    /**
     * Prints a String representation of the clock in 24h format.
     * @return An string in 24h format
     * @see String.format function to format integers with leading zeroes
     */
    public String printHour24() { /*...*/ }
```

```
    /**
     * Prints a String representation of the clock in 12h format.
     * @return An string in 12h format
     * @see String.format function to format integers with leading zeroes
     */
    public String printHour12() { /*...*/ }

    /**
     * Performs the equality tests of the current clock with another clock
     * passed as a parameter. Two clock are equal if they represent the same
     * instant regardless of being in 12h or 24h format.
     * @param o The clock to be compared with the current clock.
     * @return true if the clocks are equals, false otherwise.
     */
    @Override
    public boolean equals(Object o) { /*...*/ }

    /**
     * Returns a integer that is a hash code representation of the clock using
     * the "hash 31" algorithm.
     * Clocks that are equals must have the same hash code.
     * @return the hash code
     */
    @Override
    public int hashCode() { /*...*/ }
}
```

Criterios:

- Instanciación de objetos.
- Abstracción y encapsulamiento.
- Manejo de excepciones.
- Contratos del `equals` y el `hashCode`.
- Conversiones de entero a `String` y viceversa.

4. **Semáforos y cruces**

Un semáforo tiene las siguientes características:

- Tiene tres colores: rojo, ambar y verde. Solo uno puede estar encendido a la vez.
- El color ambar puede parpadear, pero los otros no.
- La secuencia de cambio de colores es siempre la misma: de verde a ambar, luego a rojo y de vuelta de nuevo al verde.
- Los semáforos tienen un tiempo determinado para estar en verde y en amarillo, luego automáticamente pasan al siguiente color de la secuencia.
- Por simplificar supondremos que todos los semáforos tienen los mismos tiempos: `15` segundos en verde y `5` segundos en amarillo.
- El tiempo que pasan en rojo dependerá de cuántos semáforos hay en un cruce, pero como mínimo será de `20` segundos (`15`+`5`).
- El color ambar con parpadeo es un color especial que, en nuestro modelo, se usa para indicar que el semáforo está desactivado. No tiene un tiempo en concreto en el que estar en dicho estado (puede ser indefinidamente).
- Todo semáforo tendrá un nombre que lo identifique.

En base a eso crea una clase `TrafficJunction` que represente a un cruce controlado por semáforos que tenga los métodos que se indican a continuación:

```
public class TrafficJunction {
    /**
     * Creates a trafic junction with four traffic lights named north, south,
     * east and west. The north traffic light has just started its green cycle.
     */
    public TrafficJunction() { /* ... */ }
    /**
     * Indicates that a second of time has passed, so the traffic light with
     * the green or amber light should add 1 to its counter. If the counter
     * passes its maximum value the color of the traffic light must change.
     * If it changes to red then the following traffic light changes to green.
     * The order is: north, south, east, west and then again north.
     */
    public void timesGoesBy() { /* ... */ }
    /**
     * If active is true all the traffic lights of the junction must change to
     * blinking amber (meaning a non-controlled junction).
     * If active is false it resets the traffic lights cycle and started again
     * with north at green and the rest at red.
     * @param active true or false
     */
    public void amberJunction(boolean active) { /* ... */ }
    /**
     * Returns a String with the state of the traffic lights.
     * The format for each traffic light is the following:
     * - For red colors: "[name: RED]"
     * - For green colors: "[name: GREEN counter]"
     * - For yellow colors with blink at OFF: "[name: YELLOW OFF counter]
     * - For yellow colors with blink at ON: "[name: YELLOW ON]
     * Examples:
     * [NORTH: GREEN 2][SOUTH: RED][EAST: RED][WEST: RED]
     * [NORTH: AMBER OFF 5][SOUTH: RED][EAST: RED][WEST: RED]
     * [NORTH: AMBER ON][SOUTH: AMBER ON][EAST: AMBER ON][WEST: AMBER ON]
     * @return String that represents the state of the traffic lights
     */
    @Override
    public String toString() { /* ... */ }
}
```

Criterios:

- El problema debe resolverse haciendo uso de valores enumerados en Java. Qué tipos deben ser y con qué diseño es tarea vuestra.
- Se valorará especialmente que no se usen enumerados simples sino que sean enumerados complejos (con constructores, métodos, etc.).
- También se valorará que los enumerados se usen eficientemente, haciendo uso de sus propios métodos (como `values()`) o de clases diseñadas para manejarlos (como `EnumSet` o `EnumMap`).
- Los semáforos nunca pueden ser puestos en un estado inconsistente (en ningún color, en más de un color, en verde parpadeando, etc.)
- Se valorará también que `TrafficJunction` sea una clase sencilla que delegue la mayoría del funcionamiento en otras clases (cambio de colores en el semáforo, tiempo en un determinado color, etc.).