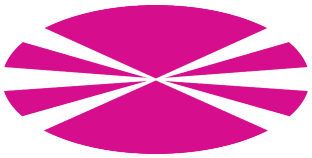

UNIVERSIDADE DA CORUÑA

Diseño Software

Boletín de Ejercicios 1 (2022-2023)

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".
  - El nombre del equipo de prácticas será el nombre del grupo de prácticas al que pertenecen los miembros (con un prefijo "`DS-`") seguido por sus correspondientes *logins* de la UDC separados por guiones bajos, por ejemplo: `DS-12_jose.perez_francisco.garcia`.
  - En caso de pertenecer a grupos de prácticas distintos anteponer los dos grupos al inicio, siendo el primer grupo el que corresponde al primer login, como en el siguiente ejemplo: `DS-12-32_jose.perez_francisco.garcia`.
- **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 utilizando el servicio *GitHub Classroom*.
  - Tendremos una clase de prácticas dedicada a explicar Git, su uso en IntelliJ, GitHub Classroom y a cómo entregar las prácticas usando este sistema. Hasta entonces podéis ir desarrollando las prácticas en local.
  - Para la evaluación de la práctica sólo tendremos en cuenta aquellas contribuciones hechas hasta la fecha de entrega en el correspondiente repositorio de GitHub Classroom, 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: 07 de octubre de 2022 (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-12_jose.perez_francisco.garcia-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 y en el seminario de Git os comentaremos su integración con GitHub Classroom.
- **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. **Utilidades para fechas**

   Crea una clase `DateUtilities` que tenga los siguientes métodos estáticos:

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

   Criterios:

   - Manejo de estructuras típicas de control de Java.
   - Manejo de la clase `String` y sus métodos.

2. **Distancia social en la universidad**

La universidad ha decidido marcar la disposición de las aulas para que se mantenga la distancia social entre los alumnos, de esta forma ha marcado cada una de las mesas con una "`A`" indicando que está disponible para los alumnos y con un "`.`" indicando que no debe usarse.

Por ejemplo, aquí vemos una disposición de un aula pequeña en la que el responsable del marcado no ha sido especialmente hábil.

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

Los alumnos pueden ahora sentarse libremente en los sitios marcados con una "`A`". Sin embargo, los alumnos prefieren ser prudentes y a la hora de sentarse siguen la siguiente regla:

- "Solo sentarse en un asiento si en los 8 asientos adyacentes al mismo no hay otra persona sentada".

Los alumnos toman la decisión de dónde sentarse viendo la disposición del aula, y se sientan todos a la vez, así en la primera iteración todos los sitios están libres inicialmente y todos serán ocupados al no tener nadie adyacente. El aula quedará entonces de esta forma, donde un "`#`" indica que el asiento está ocupado por el alumno.

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

Una vez sentados, los alumnos son reticentes a levantarse, pero si ven que hay mucha gente en los asientos adyacentes decidirán levantarse siguiendo esta regla:

- "Levantarse de un asiento en el que cuatro o más de sus asientos adyacentes estén ocupados".

La disposición del aula quedará por tanto de la siguiente forma:

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

Los alumnos continuaran sentándose y levantándose siguiendo las dos reglas antes mencionadas (nos sentamos si no hay alumnos adyacentes, nos levantamos si hay cuatro o más alumnos adyacentes), siempre teniendo en cuenta que los alumnos se levantan o se sientan todos a la vez. Por lo tanto, nuestra siguiente iteración será:

#	.	#	A	.
#	A	#	A	#
A	.	#	.	#
#	A	A	A	.
#	.	A	#	.

Llegados a este punto vemos ningún alumno tiene la necesidad de levantarse, porque no tiene 4 o más alumnos adyacentes, y ningún alumno puede sentarse, porque no hay sitios libres sin nadie adyacente, por lo que esta será la disposición final de nuestra aula.

Para este ejercicio se pide, por tanto, la implementación de la siguiente función:

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

Criterios:

- Manejo de *arrays* en Java.
- Manejo de bucles.
- Lanzamiento de excepciones.

3. **Triángulos**

Desarrolla un **registro** `Triangle` que represente el funcionamiento básico de los triángulos y que incluya los siguientes métodos:

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

Criterios:

- Uso de registros.
- Instanciación de objetos.
- Abstracción y encapsulamiento.
- Manejo de excepciones.
- Contratos del `equals` y el `hashCode`.

4. **Valoraciones de películas**

Escribe un enumerado `MovieRating` que represente las valoraciones asociadas a películas que los usuarios hacen usualmente en páginas web. Los detalles del enumerado corren a vuestro cargo pero este debe incluir:

- Los siguientes valores con sus valores numéricos asociados: `NOT_RATED`, `AWFUL` (0), `BAD` (2), `MEDIOCRE` (4), `GOOD` (6), `EXCELLENT` (8), `MASTERPIECE` (10).
- El constructor adecuado para dichos elementos.
- Un método `int getNumericRating()` que devuelva el valor numérico de la valoración.
- Un método público `boolean isBetterThan(MovieRating)` que devuelve `true` sí y sólo sí la valoración actual es mayor que otra que se le pasa por parámetro. Podemos suponer que cualquier valoración es mejor que la valoración `NOT_RATED`.

Nota: El enumerado `MovieRating` que creéis tiene que funcionar perfectamente con los tests suministrados sin necesidad de cambiar estos últimos.

Escribe una clase `Movie` destinada a representar películas y asociarles una lista de valoraciones. La especificación de la clase se muestra a continuación:

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

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.