UNIVERSIDADE DA CORUÑA

Diseño Software

Práctica 1 (2023-2024)

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`[1].
  - 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**.

[1] Si tenéis un *login* muy largo podéis tratar de acortarlo poniendo solo un apellido, siempre y cuando no exista confusión con algun compañero vuestro.

INSTRUCCIONES PRÁCTICA 1:

Fecha límite de entrega: 13 de octubre de 2023 (hasta las 23:59).

- **Realización de la práctica**
  - Los ejercicios se entregarán usando *GitHub Classroom*. En concreto en el *assignment* `2324-P1` del *classroom* `GEI-DS-614G010152324-OP1`.
  - Se deberá subir al repositorio un único proyecto IntelliJ IDEA para la práctica con el nombre del grupo de prácticas más el sufijo "`-P1`" (por ejemplo `DS-12_jose.perez_francisco.garcia-P1`).
  - Se creará un paquete por cada ejercicio de la práctica 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 esta primera práctica os adjuntaremos algunos de los tests JUnit que deben pasar los ejercicios para ser considerados válidos.
  - **IMPORTANTE: No debéis modificar los tests que os pasemos**, sí podréis añadir nuevos tests para esos ejercicios si lo consideráis necesario. **Si un ejercicio no tiene tests es responsabilidad vuestra desarrollar los tests para el mismo**. Valoraremos la calidad de las pruebas realizadas.
  - 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**
  - Esta práctica 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 esta práctica**.
  - 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. **Manipulación de Strings**

   Crea una clase `StringUtilities` que implemente los siguientes tres métodos de utilidad para manipulación de `strings`:

   - **isValidString**: Determina si un `String` que se pasa como parámetro es válido, devolviendo un valor `boolean`. Dicho `String` será válido o no en base a los siguientes criterios:
     a) Los caracteres que contiene únicamente pueden ser dígitos numéricos o estar entre los caracteres incluidos en un `String` de caracteres admisibles que también se pasará como parámetro del método.
     b) La longitud del `String` ha de ser mayor o igual que un valor numérico indicado en un parámetro del método.
     c) Un `String` nulo o vacío es NO válido.
   - **lowercaseFirst**: Reordena los caracteres de un `String` que se pasa como parámetro de modo que las letras en minúscula aparezcan primero. El método devolverá el nuevo string reordenado.
     - El `String` de entrada puede contener letras mayúsculas y minúsculas en cualquier orden.
     - Precondición: el `String` de entrada debe ser no nulo.
     - En la implementación debe realizarse un único bucle de recorrido sobre los caracteres del `String` de entrada.
   - **checkTextStats**: Valida si un texto que se pasa como un parámetro de tipo `String` cumple con un conjunto de requisitos estadísticos referentes al recuento de las palabras que contiene. El resultado del método será un valor `boolean`. Consideraciones:
     - El `String` que se pasa como parámetro contendrá palabras separadas por un espacio en blanco (ej. "Java es un lenguaje de programación").
     - El método devolverá `True` si el texto cumple:
       a) La longitud media de las palabras del texto está entre un valor mínimo y un valor máximo que también se pasan como parámetros.
       b) La longitud de la palabra más larga no supera al doble de la longitud media de las palabras.
     - El método lanzará la excepción `IllegalArgumentException` si el texto de entrada es un `String` nulo o vacío, y también si alguno de los valores mínimo y máximo que se pasan como parámetro son negativos o cero.

   Ayuda:

   - Al ser métodos de utilidad, todos deberían ser definidos como públicos y estáticos (ejemplo: `public static boolean miMetodo(...)`).
   - La clase `Character` de Java tiene métodos que pueden ser útiles como `Character.isDigit(...)` o `Character.isLowerCase(...)`.

   Criterios:

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

2. **Matrices inmutables**

   Crea una clase `ImmutableMatrix` que representa a una matriz de enteros inmutable (una vez creada no puede modificarse, de la misma forma que los `String` Java).

   Para instanciar el objeto dispondremos de los siguientes constructores:

   - **Un constructor que acepta un array bidimensional de valores enteros (`int[][] arr`)**. El constructor deberá copiar los valores de dicho array para inicializar la matriz inmutable. No podrá usar dicho array porque es un objeto externo mutable, y por lo tanto puede ser modificado. El constructor deberá comprobar que el array pasado por parámetro no es irregular (*ragged*). En caso de serlo deberá lanzar la excepción `IllegalArgumentException`.
   - **Un constructor que acepta dos valores enteros (numero de filas y número de columnas) y crea una matriz con esas dimensiones** y rellena con los números que empiezan en 1 (en la esquina superior izquierda) incrementando sucesivamente los valores (de izquierda a derecha y de arriba a abajo) hasta llegar al valor filas x columnas en la esquina inferior derecha. Los valores tendrán que ser valores enteros mayores que cero, sino se lanzará la excepción `IllegalArgumentException`. En la imagen vemos el resultado de llamar al constructor con el valor de `filas=3` y `columnas=4`.

1	2	3	4
5	6	7	8
9	10	11	12

   Además de los constructores la clase `ImmutableMatrix` deberá tener los siguientes métodos:

   - `toString`: Devuelve una representación en String del objeto.
   - `at`: Devuelve el elemento en la posición indicada (fila y columna) o lanza la excepción `IllegalArgumentException` si las coordenadas no son válidas.
   - `rowCount`: Devuelve el número de filas de la matriz.
   - `columnCount`: Devuelve el número de columnas de la matriz.
   - `toArray2D`: Devuelve una representación de la matriz como un array 2D. Es importante que lo que devuelva sea una copia, para no afectar a la inmutabilidad del objeto.
   - `reverse`: Devuelve otro objeto `ImmutableMatrix` que es similar al que se pasa por parámetro pero las filas están en orden inverso, tal y como muestra la imagen.

Example:

1	2	3	4
5	6	7	8
9	10	11	12

`.reverse()` returns

4	3	2	1
8	7	6	5
12	11	10	9

   - `transpose`: Devuelve otro objeto `ImmutableMatrix` que representa la matriz traspuesta (intercambiando filas y columnas) de la matriz pasada por parámetro, tal y como muestra la imagen.

Example:

1	2	3	4
5	6	7	8
9	10	11	12

`.transpose()` returns

1	5	9
2	6	10
3	7	11
4	8	12

- `reshape`: Dado un número de columnas que se pasa por parámetro, devuelve otro objeto del tipo `ImmutableMatrix` que es el resultado de reorganizar los datos de la matriz original en una nueva matriz que tenga las columnas indicadas. Como la matriz no puede ser irregular se tiene que cumplir que, si las dimensiones de la matriz original eran r (filas) y c (columnas), entonces $r \times c$ tiene que ser divisible por *newColumns*. En caso contrario se lanzará la excepción `IllegalArgumentException`. Ver ejemplo en la imagen siguiente.

Example:

1	2	3	4
5	6	7	8
9	10	11	12

`.reshape(6)` returns

1	2	3	4	5	6
7	8	9	10	11	12

Si tenéis dudas sobre el funcionamiento de alguno de los métodos descritos, os dejaremos en el campus virtual un test que deberá pasar vuestro código y que ejemplifica el funcionamiento de dichos métodos. El test es bastante sencillo y os aconsejamos completarlo con nuevos casos, pero los casos ya incluidos no deberéis modificarlos.

Se aconseja el uso de la clase `Arrays` y alguno de sus métodos de utilidad como `copyOf` o `toString`.

Criterios:

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

3. **Base de datos de grabaciones musicales**

   En este ejercicio manejaremos metadatos acerca de grabaciones musicales (recordings), lanzamientos (releases) y pistas (tracks):

   - Un **lanzamiento** es un disco de música que incluye atributos como un identificador, un título, un artista principal, un tracklist, etc.
   - Una **pista** incluye atributos como la posición de la canción en el disco, la duración, el identificador de la grabación correspondiente, etc.
   - Una **grabación**, por simplificar, se denotará simplemente por un identificador, sin más metadatos adicionales. El concepto de grabación existe simplemente para identificar una versión específica de una composición. Es habitual que un mismo artista lance diferentes versiones de un mismo tema, y las consideraríamos grabaciones diferentes.

   En este ejercicio deberéis crear una **clase** llamada `Release` para los lanzamientos y un **registro** llamado `Track` para las pistas. Deberéis crear los habituales métodos para leer y escribir los atributos mencionados anteriormente, así como los métodos típicos de los objetos Java como `toString` (mostrando la información en el orden en el que debería aparecer), etc. También deberéis lanzar **excepciones** para errores típicos y triviales.

   Un concepto muy importante en este ejercicio es la igualdad. Consideraremos que dos **pistas son iguales si y solo si** corresponden a las mismas grabaciones, ignorando los demás atributos, ya que estos pueden variar ligeramente al reeditar el disco o al remasterizarlo.

   De manera similar, dos **lanzamientos son iguales si y solo si** contienen exactamente las mismas grabaciones independientemente de su orden, ya que nuevamente es algo que puede variar entre ediciones. Por ejemplo, cuando una cara B de un single se convierte en éxito es habitual que se reedite como cara A.

   Os hemos entregado **tests** parciales con ejemplos de lanzamientos, pistas y grabaciones que deberán ser pasados. A estos tests debéis añadir los vuestros propios. Nuestros ejemplos están basados en MusicBrainz (`https://musicbrainz.org/`), una base de datos open source de metadatos que es utilizada a menudo para etiquetar ficheros de audio.

   En este ejercicio se recomienda no utilizar arrays sino las clases para listas o conjuntos del **Framework de Colecciones de Java**.

   Criterios:

   - Contratos del `equals` y el `hashCode`.
   - Instanciación de objetos.
   - Encapsulamiento.
   - Uso de *registros*.
   - Manejo de excepciones.
   - Método `toString`.

4. **Billar**

   En este ejercicio simularemos el funcionamiento del juego de billar. Para ello, crearemos una clase enumerada `BolaBillar` que tendrá los siguientes valores enumerados con las siguientes características (número, color y tipo):

   - BLANCA: 0 BLANCO LISA
   - BOLA1: 1 AMARILLO LISA
   - BOLA2: 2 AZUL LISA
   - BOLA3: 3 ROJO LISA
   - BOLA4: 4 VIOLETA LISA
   - BOLA5: 5 NARANJA LISA
   - BOLA6: 6 VERDE LISA
   - BOLA7: 7 GRANATE LISA
   - BOLA8: 8 NEGRO LISA
   - BOLA9: 9 AMARILLO RAYADA
   - BOLA10: 10 AZUL RAYADA
   - BOLA11: 11 ROJO RAYADA
   - BOLA12: 12 VIOLETA RAYADA
   - BOLA13: 13 NARANJA RAYADA
   - BOLA14: 14 VERDE RAYADA
   - BOLA15: 15 GRANATE RAYADA

   Crearemos también una clase `MesaBillar`. Incluirá un constructor que permita crear una mesa de billar con todas las bolas disponibles en el cajetín de la mesa. Además, contará con los siguientes métodos:

   - `iniciarPartida`: Inicializará el estado de la mesa indicando que hay una partida en marcha y que todas las bolas están encima de la misma.
   - `meterBola`: Si es una bola normal (de 1 a 7 o de 9 a 15) se sacará de encima de la mesa y se pondrá en el cajetín con las otras bolas ya metidas (en orden de introducción). En caso de que la bola sea la blanca, volverá a la mesa (a no ser que se haya acabado la partida). Finalmente, si la bola es la negra se considerará que la partida ha acabado.
   - `bolasMesa`: Devuelve las bolas que hay en juego en la mesa.
   - `bolasCajetin`: Devuelve las bolas que hay en el cajetín la mesa.
   - `esPartidaIniciada`: Comprueba si la partida está en juego o no.
   - `obtenerGanador`: Indica qué jugador va ganando (si el de las bolas lisas, bolas de la 1 a la 7) o el de las bolas rayadas (bolas de la 9 a la 15). Va ganando, o ha ganado si la partida ha terminado, aquel jugador que tenga menos bolas de su tipo encima de la mesa.

   Criterios:

   - Creación y manejo de objetos enumerados complejos.
   - Manejo de clases de colecciones de objetos como `ArrayList`.