PRÁCTICA 01

Estructuras de Datos Lineales en Java

Objetivos

- Estudiar, desde el punto de vista del almacenamiento de datos, consulta y manipulación, dos de las principales estructuras de datos lineales en Java: **ArrayList<T>** y **LinkedList<T>**.
- Utilización conjunta de ambas estructuras para la resolución de problemas.

Requisitos

Para superar esta práctica se debe realizar lo siguiente:

- Dominar las estructuras de datos lineales **ArrayList<T>** y **LinkedList<T>**, así como la combinación de ellas en la resolución de problemas.
- Conocer cómo realizar operaciones de consulta y manipulación eficientes sobre este tipo de estructuras de datos, así como la mejor forma de plantearlas en problemas sencillos.
- Dado un problema concreto, determinar con criterios objetivos cuál sería la estructura de datos idónea para su resolución.

Enunciado (Twitter intrusivo)

Durante los últimos años se ha generado un debate polémico sobre la necesidad de control de las redes sociales como, por ejemplo, Twitter. Desde el principio de la libertad de expresión, los usuarios envían mensajes (en nuestro caso, solo textos) a las redes haciendo uso de dispositivos de comunicación. Aunque, según ciertas fuentes, este ejercicio de libertad de expresión ejercidos por unos puede vulnerar derechos fundamentales de otros.

Así pues, a modo de experimento tecnológico y, desde un punto de vista muy intrusivo, vamos a desarrollar un programa que permita: (1) almacenar las palabras que los usuarios utilizan en los mensajes que envían a través de dispositivos; y (2) permitir operaciones de consulta y manipulación sobre este conjunto de palabras, permitiendo saber si un determinado usuario hace uso (o no) de una determinada palabra prohibida para poder sustituirla por otra o, directamente, eliminarla (según un criterio censor).

En nuestra versión simplificada de lo que podría ser una red social de este tipo, vamos a trabajar con dos entidades diferentes: Dispositivos (Device) y Usuarios (User).

El conjunto de requisitos básicos se puede resumir en:

- Un usuario envía mensajes haciendo uso de un determinado dispositivo de comunicación.
- Un usuario podrá utilizar diferentes dispositivos de comunicación.
- Un dispositivo podrá ser utilizado por diferentes usuarios.
- De los mensajes enviados solo nos interesa mantener el conjunto de palabras diferentes que ha utilizado. Para simplificar el problema, vamos a suponer que en una fase de preproceso se

han eliminado los signos de puntuación, tildes, el carácter ñ, así como cualquier otro carácter no existente en el abecedario inglés.
- Así pues, un mensaje se define como una secuencia de palabras separadas por espacios en blanco.
- Aunque el usuario puede hacer uso de mayúsculas y minúsculas según su criterio personal, las palabras las vamos a almacenar siempre en minúsculas.

Para determinar la funcionalidad de nuestra aplicación, se proporciona un conjunto de clases parcialmente implementadas junto con un test de prueba, que será quien guíe de forma estricta el proceso de implementación.

Para la resolución del problema, vamos a proponer la definición de las clases **Device** y **User** tal y como se muestra a continuación.

```
public class Device implements Iterable<String> {
        private static int numDevices=0; //contador de dispositivos...atributo estático
        private String name; //Nombre del dispositivo (clave)
        private int id;  //Identificador del dispositivo
        private LinkedList<String> words; //Conjunto de palabras enviadas a través de él

        public Device() {...}
        public Device(String name) {...}
        public static void inicializaNumDevices() {...}
        public int getId() {...}
        public void clear() {...}
        public void sendMessage(String msg) {...}
        public boolean contains(String word) {...}
        public boolean substitute(String word1, String word2) {...}
        @Override
         public String toString() {...}
        @Override
        public boolean equals(Object o) {...}
        @Override
        public Iterator<String> iterator() {...}
}
```

De esta propuesta, destacamos los siguientes aspectos en relación a los atributos:

- Un dispositivo va a estar identificado de forma unívoca por su nombre (el atributo name es la clave). Se trata de una restricción relativamente importante, ya que un usuario no va a poder tener, por ejemplo, dos "*iPad*", cuestión que no se da con demasiada frecuencia. Aunque sí que podrá tener un *iPad Air* y un *iPad Pro*. Habrá que tener en cuenta este detalle cuando implementemos el método equals().
- A cada dispositivo se le asocia un identificador autonumérico (id). La primera vez que se crea un dispositivo se le asocia el identificador id=1; al segundo dispositivo se le asigna id=2 y así sucesivamente. Para llevar el contador de dispositivos "creados" vamos a hacer uso de la variable estática numDevices que, como sabéis, se trata de una variable de clase compartida por todos y cada uno de los objetos de tipo **Device**.
- El conjunto de palabras que han sido enviadas a través de este dispositivo se almacena en el contenedor lineal words. En la solución que os proponemos hemos decidido implementarlo haciendo uso de la estructura **LinkedList<String>**. Os sugerimos que reflexionéis sobre su idoneidad en este caso concreto, es decir, ¿no sería mejor hacer uso de **ArrayList<String>**?

Del conjunto de métodos públicos (interfaz de la clase), destacamos los siguientes aspectos:

- El método inicializaNumDevices() se va a encargar únicamente de inicializar el atributo estático numDevices asignándole el valor 0. ¿Es necesario que este método sea estático? Si no lo fuera, ¿qué modificaciones habría que hacer en el test?
- La idea que subyace a la utilización del método iterator() (necesario cuando nuestra clase es **Iterable**) es la siguiente: Si desde una clase distinta a **Device** queremos iterar sobre el contenedor words (**ArrayList<T>**) tengo tres opciones:
  - Hago que el atributo words sea público (**incorrecto**)..
  - Implemento un método getWords() que devuelva el **ArrayList<T>** (**peligroso e incorrecto**, ya que desde cualquier clase externa se podría modificar los datos sin permiso ni control de la clase propietaria).
  - Permito que se **itere** sobre esta estructura (**perfecto**). Así pues, iterar sobre **Device** equivale a iterar sobre las palabras contenidas en el contenedor words. De ahí que nuestra clase implemente la interfaz **Iterable<String>** y, consecuentemente, tengamos que implementar el método iterator().

Por otra parte, la clase **User** que proponemos es la siguiente:

```
public class User {
        private String name;
        private ArrayList<Device> devices;

        public User(String name) {...}
        public void clear() {...}
        public boolean addDevices(Device... devs) {...}
        public int getNumDevices() {...}
        public boolean loadMessages(Device dev, String fileName) {...}
        public boolean sendMessage(Device dev, String msg) {...}
        public void substitute(String word1, String word2) {...}
        public boolean contains(String word) {...}
        public String getWords() {...}
        public ArrayList<String> getOrderedWords() {...}
        @Override
        public String toString() {...}
}
```

El conjunto de dispositivos utilizados por un usuario se va a almacenar en el contenedor devices. Una vez más, tendremos que reflexionar sobre la idoneidad de este tipo de estructura para la resolución de este problema. Por otro lado, destacamos el método loadMessages(), cuyo objetivo principal consiste en la carga de datos a partir de un archivo de texto plano.

Para más detalles de implementación, consultad las clases **Device** y **User**, así como el test proporcionado.

Como ejercicios, además de: (1) responder a las preguntas planteadas en el guion; (2) pasar correctamente el test asociado con la práctica (compuesto por distintos subtest); (3) tendréis que realizar un análisis comparativo sobre las estructuras **ArrayList<T>** y **LinkedList<T>**.

En Google podemos encontrar diferentes estudios comparativos muy interesantes, aunque la idea consiste en hacer vuestro propio análisis "**personalizado**" a partir de la información que podemos encontrar en la API de Java.