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 colecciones lineales en Java: **ArrayList<T>** y **LinkedList<T>**.
- Utilización conjunta y anidada de ambas estructuras como mecanismo eficiente para la resolución de problemas.

Requisitos

Para superar esta práctica se debe realizar lo siguiente:

- Dominar el uso de 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 (*Gestión de citas para una clínica veterinaria*)

En esta primera práctica vamos a implementar un sencillo sistema de gestión de citas para una clínica veterinaria. Básicamente, vamos a llevar un control de las valoraciones que los profesionales de la veterinaria hacen cada vez que un cliente acude a una cita con su mascota.

En nuestra versión simplificada de sistema de gestión, vamos a trabajar con tres entidades diferentes: Cita, Mascota y Cliente.

El conjunto de requisitos básicos se puede resumir en:

- Un cliente tendrá asociado un identificador de tipo numérico (5 dígitos) y su nombre.
- Un cliente puede tener muchas mascotas.
- Cada mascota será identificada por su nombre y especie, de manera que un mismo cliente podrá tener dos o más mascotas que se llamen igual siempre y cuando sean de especies distintas.
- Se llevará un registro individualizado de todas y cada una de las veces que un cliente ha llevado a una de sus mascotas a la clínica.
- En cada cita, el profesional podrá introducir en el sistema tantas valoraciones como desee.
- Aunque el profesional puede hacer uso de mayúsculas y minúsculas según su criterio personal, todas las palabras las vamos a almacenar siempre en minúsculas.
- Se implementará un conjunto de métodos que permita recuperar información como, por ejemplo; (1) obtener el total de citas en las que ha sido atendida a una mascota; (2) obtener el conjunto de valoraciones realizadas en una determinada cita; (3) obtener el conjunto de

citas en las que aparece una determinada palabra en el conjunto de valoraciones; (4) …

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 **Cita**, **Mascota** y **Cliente** tal y como se muestra a continuación.

```
public class Cita {
   private static int numCitas = 0;
   private final int citaId;
   private final ArrayList<String> valoraciones;

   public static void inicializaNumCitas() {...}
   public Cita() {...}
   public Cita(int citaId) {...}
   public int getCitaId() {...}
   public boolean addValoracion(String valoracion) {...}
   public boolean contienePalabra(String palabra){...}
   public void clear() {...}
   @Override
   public String toString() {...}
   @Override
   public boolean equals(Object o) {...}
}
```

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

- A cada cita se le asocia un identificador autonumérico (citaId). Independientemente de la mascota, la primera vez que se crea una cita se le asocia el identificador citaId=1; a la segunda cita se le asigna citaId=2 y así sucesivamente. Para llevar el contador de citas "creadas" hasta el momento se va a hacer uso de la variable estática numCitas que, como sabes, se trata de una variable de clase compartida por todos y cada uno de los objetos de tipo **Cita**.
- Así pues, una cita va a estar identificada de forma unívoca por su identificador de cita (número) . El atributo citaId es la **clave**, así que dos citas son iguales sii (si y sólo si) tienen el mismo identificador de cita.
- El conjunto de valoraciones (frases) que realiza el profesional se almacena en el contenedor lineal valoraciones. En la solución que proponemos hemos decidido implementarlo haciendo uso de la estructura **ArrayList<String>**. Os sugerimos que reflexionéis sobre su idoneidad en este caso concreto, es decir, ¿no sería mejor hacer uso de **LinkedList<String>**?

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

- El método inicializaNumCitas() se va a encargar únicamente de inicializar el atributo estático numCitas asignándole el valor 0. ¿Es necesario que este método sea estático? Si no lo fuera, ¿qué modificaciones habría que hacer en el test?
- El método contienePalabra() devolverá true si una valoración contiene la palabra que se especifica como parámetro de entrada. A la hora de implementar este método, tendremos que tener en mente la palabra **EFICIENCIA**, ¿verdad?. Si una cita tiene mil valoraciones (o cientos de miles e incluso millones, por no decir miles de millones…) y la palabra se encuentra justo en la primera valoración, ¿vamos a permitir que el método siga el proceso de

búsqueda o, de contrario, el método finalizará justo en el momento en el que encuentre la palabra?

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

```
public class Mascota implements Comparable<Mascota>, Iterable<Cita> {
   private final String nombre;
   private final String especie;
   private final LinkedList<Cita> historial;

   public Mascota(String nombre, String especie) {...}
   public Cita addCita() {...}
   public Cita getCita(int citaId) {...}
   public void clear() {...}
   public int size() {...}
   @Override
   public String toString() {...}
   public String toStringExtended() {...}
   @Override
   public boolean equals(Object other){...}
   @Override
   public int compareTo(Mascota other) {...}
   @Override
   public Iterator<Cita> iterator() {...}
}
```

Una mascota va a estar identificada por los atributos nombre (como clave principal) y especie (como clave secundaria). Es decir, para comparar dos mascotas, seguimos el siguiente proceso: (1) en primer lugar comparamos sus nombres. Si los nombres son diferentes, se concluye indicando que se trata de mascotas diferentes; (2) en caso contrario, se comparan las especies. Si son de especies diferentes, concluimos indicando que se tratan de dos mascotas diferentes; en caso contrario el método devolverá true, indicando que se trata de la misma mascota.

Para el registro de citas, se ha elegido una colección tipo LinkedList<Cita>. Una vez más debes cuestionar la idoneidad de la elección de este tipo de contenedor. Observa un detalle importante: el contenedor historial es una lista enlazada de citas, es decir, cada nodo tendrá una referencia a un objeto de tipo Cita. Y como ya conoces, cada cita tiene una colección de tipo ArrayList<String>. Se trata de una anidación de estructuras de datos, ¿verdad? Haz un esquema visual de esta anidación de estructuras de datos. Esto te ayudará en la resolución del problema.

De esta propuesta, destacamos los siguientes aspectos:

- Observa que la clase Mascota implementa la interfaz Comparable<Mascota> ya que vamos a permitir la ordenación de colecciones de mascotas. Esto nos exige la implementación del método compareTo(). Observa también que tenemos que implementar el método equals(). Tanto equals() como compareTo() deben ser coherentes, es decir, deben seguir la misma lógica para comparar dos objetos de tipo Mascota. Y, para asegurar dicha coherencia, y como verás en la implementación parcial que te proporcionamos, equals() determina que dos objetos son iguales sii compareTo() devuelve 0.
- La idea que subyace a la utilización del método iterator() (necesario cuando nuestra clase implementa la interfaz **Iterable<T>**) es la siguiente: Si desde una clase distinta a **Mascota** queremos iterar sobre el contenedor historial (**LinkedList<T>**) vamos a tener tres opciones:

1. Hacer que el atributo historial sea público (**incorrecto**)..
2. Implemento un método getHistorial() que devuelva una referencia del atributo **LinkedList<T>** (**peligroso e incorrecto**, ya que desde cualquier clase externa se podría modificar los datos sin permiso ni control de la clase propietaria).
3. Permito que se **itere** sobre esta estructura (**perfecto**). Así pues, iterar sobre **Mascota** equivale a iterar sobre las citas contenidas en el contenedor historial. De ahí que nuestra clase implemente la interfaz **Iterable<Cita>** y, consecuentemente, tengamos que implementar el método iterator().

De esta manera, podríamos hacer lo siguiente:

```
Mascota mascota = new Mascota("toby", "perro");
//… añadimos citas
//Iteramos sobre una mascota
for(Cita cita: mascota) { //Iterar sobre una mascota equivale a iterar sobre la colección historial
        …
}
```

Interesante, ¿verdad?

Y ya por último, la clase **Cliente**:

```
public class Cliente {
   private static int numClientes = 0;
   private final String nombre;
   private final ArrayList<Mascota> mascotas;

   public static void inicializaNumClientes() {...}
   public Cliente(String nombre) {...}
   public boolean addMascota(String nombre, String especie) {...}
   public Cita addCita(String nombre, String especie) {...}
   public void clear() {...}
   public int size() {...}
   public ArrayList<ArrayList<Integer>> getCitasId(){...}
   public ArrayList<ArrayList<Integer>> getCitasId(String palabra) {...}
   public ArrayList<Integer> getCitasId(String nombre, String especie){...}
   public Cita getCita(int citaId){...}
   public Mascota getMascota(int citaId){...}
   @Override
   public String toString() {...}
   public String toStringExtended() {...}
   public boolean load(String nombreArchivo) {...}
}
```

Solo comentar la funcionalidad de la clase load(), encargada de cargar los datos desde un archivo de texto (como podrás observar, proporcionamos dos ejemplos, cliente01.txt y cliente02.txt). Antes de empezar con la implementación del método load(), abre los archivos y analiza su estructura.

Observa también que ahora tenemos como atributo una colección de mascotas. Completa el esquema visual que has empezado añadiendo esta nueva anidación de estructuras de datos.

Para más detalles de implementación, consulta las clases parcialmente implementadas **Cita, Mascota** y **Cliente**, así como el test proporcionado.

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

En Google se pueden 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.