Ayuda para la solución del ejercicio

Puede hacerse todo de modo secuencial en el "**main**" o estructurarlo un poco "partiéndolo" en dos o tres rutinas. En dos, considerando una fase de adquisición y "preparación" de los datos y otra fase de elaboración/escritura de los ficheros de la salida. Puede partirse en tres si consideramos la primera fase anterior como consistente en dos partes: la primera la lectura de los ficheros de entrada y la segunda la generación de algunas estructuras de utilidad como se verá.

Primera fase: recogida de datos

La primera fase consiste en recoger todos los datos de **DB1.txt** y **DB2.txt**. El resultado será un listado de objetos "**Registro**" donde cada objeto tiene toda la información de cada línea de **DB1.txt** combinada con la correspondiente de **DB2.txt**.

Clase Registro.

La clase "Registro" guardará los datos adecuadamente si establecemos los siguientes campos:

- `String ref, titulo, año, publicacion, url;`
- `List<String> autores=new ArrayList<>();`
- `List<String>filiaciones=new ArrayList<>();`
- `String[] referencias;`

Añadiremos un constructor que admitirá un objeto **String[]** que iremos obteniendo de cada línea de **DB1.txt** mediante un **Split(";")**. Con esto se inicializarán todos los campos a excepción del **url** que se ha de obtener de **DB2.txt** y por tanto se hará posteriormente mediante un "setter"

```
public void setUrl(String url) { this.url = url;}
```

Queda por tanto ver cómo ha de ser el constructor. No hay ninguna dificultad en asignar los valores que se reciben a cada campo, con la excepción de autores y filiaciones. Estos llegan en un solo campo de la **String[]** del parámetro del constructor con la forma:

Autor "filiación", Autor "filiación",…

Así que hay que procesar esto para llevar todos los autores a la **List autores** y las filiaciones a la "**List**" **filiaciones**. Como las filiaciones contienen comas, no podemos hacer un **split(",")** y haremos un **split("\"")**, de modo que las **"strings"** que obtendremos serán autores las impares y filiaciones las pares. A excepción del primero, los demás autores comenzarán con una coma, que podremos eliminar usando el método **replace(",","")**.

Deberemos hacer siempre un **trim()** a la hora de asignar valores a los campos para eliminar caracteres en blanco que pudiera haber al principio y al final del texto.

Cómo coger toda la información en Objetos Registro

Es sencillo: hacemos un ciclo de lecturas de línea sobre un **BufferedReader** ligado al fichero DB1.txt y por cada línea construimos un objeto Registro con el resultado de hacer el **split(";")** como parámetro.

¿Dónde guardar todos estos registros?... pues en algún tipo de objeto de los disponibles en las colecciones de **java.util**. ¿Cuál es el más adecuado?... teniendo en cuenta que necesitaremos

localizar cada registro por su número asignado para añadirle al url que obtendremos al leer el fichero **DB2.txt**, lo más directo será usar un **Map** donde la clave será el número de registro y el valor el objeto Registro en sí. **Map** es un interfaz, de entre las clases que lo implementan podemos elegir el **TreeMap**

```
static Map<String, Registro> refs=new TreeMap<>();
```

Ahora ya podemos hacer un Nuevo ciclo de lectura sobre **DB2.txt** para aportar a cada registro su "url". Y terminar esta fase con toda la información adecuadamente procesada en **reg**.

Segunda fase: generación de estructuras intermedias de utilidad

Tanto los autores como las filiaciones y las publicaciones aparecen repetidas multitud de veces a lo largo de las referencias leídas, y necesitaremos disponer de unos listados donde aparezcan todas pero sin repetición. Esto es muy sencillo en Java puesto que disponemos del Interfaz Set que permite generar colecciones en las que los elementos que se incluyen más de una vez no se replican. Por tanto bastará recorrer todos los registros metiendo los autores filiaciones y publicaciones en unos conjuntos definidos para ello. Si lo hacemos concretamente en objetos de tipo **TreeSet** estarán además ordenados (conforme al orden alfabético puesto que trabajamos con "strings"). La declaración de los conjuntos será asi:

```
Set<String> aut=new TreeSet<>();
Set<String> fil=new TreeSet<>();
Set<String> publ=new TreeSet<>();
```

Una vez completadas estas tres estructuras resultará conveniente pasar la información a unos arrays, lo que puede hacerse directamente mediante el método **toArray(.)** disponible en todas las colecciones. (nota.- este paso es sencillo pero requiere entender una "sutileza" que encontrareis en la documentación) Con esto dispondremos de tres **String[]** que llamaremos **autores**, **filiaciones** y **publicaciones**, pudiendo liberar **aut, fil y publ**.

Tercera fase: generación de los ficheros de salida

Esta fase también es sencilla: por cada fichero a generar comenzaremos por establecer un **PrintStream** y recorreremos todos los registros recogiendo la información pertinente y escribiéndola en el "stream".

Sólo hay alguna dificultad en un par de casos

- En el fichero de salida articulo-autores imprimiremos una línea por cada autor que aparezca en cada referencia, pero los autores los tenemos por su nombre, no por su número asociado. Es preciso localizarlos en el "array" **autores** para obtener el índice de la posición que ocupan. Esto es, como siempre , muy sencillo mediante el método **Arrays.binarySearch(.)**

- El fichero de salida autor-filiación es el más complejo de generar. Debemos escribir una línea por cada autor y localizar de algún modo su filiación y el código correspondiente. Es decir, recorreremos el array **autores** siendo la primera columna una lista de números naturales consecutivos, pero la segunda columna encierra cierta dificultad. Para cada fila obtenemos del

array **`autores`** el nombre correspondiente; con él hemos de ir a nuestra colección de registros para recorrerla buscando al autor y cuando lo encontremos obtener su filiación en la posición correspondiente de la lista de filiaciones (para un registro, si el autor buscado está en la posición 2 de la lista autores, su filiación está en la posición 2 de la lista de filiaciones). Una vez obtenido el texto de su filiación hemos de localizar el código correspondiente en el array **`filiaciones`** del mismo modo que lo hacíamos con los autores en el punto anterior

Además hay que tener en cuenta que algunos registros no tienen referencias y hay que evitar en el fichero de artículo-referencias aparezcan líneas con la segunda columna vacía.

Y YA ESTÁ. Así de fácil.

Si no nos complicamos en ningún punto obtenemos una solución que, escrita “limpiamente” y con claridad, con sus líneas en blanco separando cada acción, rondará las 100 líneas.

Ayuda para la solución del ejercicio extra.

La diferencia con el anterior está en la "tercera fase". En lugar de generar los ficheros que posteriormente podrán leerse desde SQL, podemos enviar los datos directamente desde Java a la base de datos.

Qué necesitamos:

1. Conocer la ubicación de la base de datos: Máquina y puerto del servidor, y nombre de la base de datos.
2. Disponer de permiso de acceso (usuario y clave).
3. Cargar el "Driver" de Java capaz de "entenderse" con la base de datos MySQL
4. Establecer una conexión
5. Suponiendo que la base de datos existe, así como las tablas de datos, usar la conexión para ir enviando todos los datos línea por línea, tabla por tabla desde nuestro programa a la base de datos (o todas las líneas de una tabla de una sola vez, dependiendo de cómo lo prefiramos)

¿cómo lo resolvemos?

1. Esto se aglutina en una URL. Por ejemplo, para la máquina local, el puerto por defecto de mysql, y una base de datos llamada "referencias"

   ```
   dbURL="jdbc:mysql://localhost:3306/referencias";
   ```

   (jdbc:mysql: es el protocolo - JavaDataBaseConnection:MySQL -)
   (localhost es la máquina local)
   (3306 es el puerto por defecto de MySQL)
   (referencias es el identificador de la base de datos)
2. Los usaremos en (4)
3. Mediante la clase "Class" podemos "pedir" que se cargue el driver JDBC (JavaDataBaseConnection) que se encuentra en com.mysql.jdbc.Driver. Para ello es necesario contar con la librería MYSQL en el proyecto (botón derecho sobre la carpeta "Libraries" y "Add library..." en el menú" para seleccionarla en la lista de las disponibles). En el programa haremos:

   ```
   Class.forName("com.mysql.jdbc.Driver");
   ```

   Queda bajo el control de la clase "DriverManager".
4. A través del DriverManager conectaremos con la URL, y con el identificador de usuario y la clave obtendremos la conexión

   ```
   java.sql.Connection theConnection =
                       DriverManager.getConnection(dbURL,"usr","clv");
   ```

5. Cada acción de escritura de datos en la base de datos consiste en crear un "statement" a partir de la conexión y solicitarle que ejecute una "query" en SQL

   ```
   theConnection.createStatement().executeUpdate(sql);
   ```

   es decir, aquí "sql" es una string del tipo

   ```
   INSERT INTO table_name (column1, column2, column3,...)
   VALUES (value1, value2, value3,...)
   ```

(nota: para hacer peticiones de datos en SQL el método a ejecutar es **`executeQuery(.)`** que devuelve un objeto de clase **`ResultSet`**. No es necesario para este ejercicio)