Archivo de registros jerárquicos

Para representar datos en una computadora, los programas disponen entre dos extremos: los documentos y las bases de datos. Las bases de datos ofrecen mucho poder a las computadoras pero imponen una estructura muy formal que es difícil para las personas modificarlas. Los documentos son muy flexibles para las personas pero limitan el poder de cómputo de los programas. Existe una solución intermedia que son los documentos estructurados. Las dos notaciones más populares de documentos estructurados son el lenguaje extendido de marcado (XML, eXtensible Markup Language) y la notación de objetos de JavaScript (JSON, JavaScript Object Notation).

Para manipular un documento XML se necesita analizarlo (parsing) y para manipular un documento JSON se debe interpretar su código fuente (eval). Ambos procesos son demandantes de recursos computacionales y podrían ser poco convenientes para equipos de bajo rendimiento, tales como dispositivos del internet de las cosas (IoT, Internet of Things) y dispositivos móviles.

Una vez que se ha analizado un documento XML o interpretado un objeto JSON, el programa podría guardar los resultados en una representación que sea eficiente para cargar en dispositivos de bajo rendimiento, o incluso en la misma computadora si se quiere incrementar su rendimiento. Esta representación almacenaría el resultado del proceso de análisis o interpretación, siempre y cuando el documento original sea válido. En lo siguiente se presenta una propuesta que cumple estas características y a la que llamaremos *archivo de registros jerárquicos*. Para ejemplificarlo sea un documento XML con preguntas para un juego de Trivia:

```xml
<trivia name="general">
   <question type="numeric" answer="1000" difficulty="0.1">
      <text>Cuántos metros hay en un kilómetro?</text>
   </question>

   <question type="single_choice" answer="2" difficulty="0.3">
      <text>Quién tiene 4 estómagos?</text>
      <choices order="random">
         <choice>Una gallina</choice>
         <choice>Una vaca</choice>
         <choice>El Botija</choice>
         <choice>Una máquina tragamonedas</choice>
      </choices>
   </question>
</trivia>
```

Un archivo de registros es un archivo de texto donde cada línea representa un registro. El número de línea indica el número de registro. Un registro es un conjunto ordenado de campos. Los campos se separan por un carácter separador arbitrario, por ejemplo la barra vertical (`|`). Los campos son parejas del nombre del campo y su valor. Los elementos del ejemplo de trivia podrían traducirse a los siguientes registros:

```
type=trivia|name=general
type=question|type=numeric|answer=1000|difficulty=0.1
type=text|nodeValue=Cuántos metros hay en un kilómetro?
type=question|type=single_choice|answer=2|difficulty=0.3
type=text|nodeValue=Quién tiene 4 estómagos?
type=choices|order=random
type=choice|nodeValue=Una gallina
type=choice|nodeValue=Una vaca
type=choice|nodeValue=El Botija
type=choice|nodeValue=Una máquina tragamonedas
```

Nótese que hemos escogido el campo `type=` para indicar el nombre del elemento, y `nodeValue` para indicar su contenido textual, aunque estas decisiones son arbitrarias. Los atributos del elemento se agregaron simplemente como más campos `atributo=valor` al registro.

Como todos los registros del mismo tipo tienen siempre los mismos campos, es redundante escribir los nombres. Esta redundancia consume espacio, tanta como el archivo XML original. Si asumimos que además los campos siempre están en el mismo orden, se pueden eliminar los nombres de los mismos y bastaría con saber el tipo de registro para recuperar luego las parejas `campo=valor`:

```
trivia|general
question|numeric|1000|0.1
text|Cuántos metros hay en un kilómetro?
question|single_choice|2|0.3
text|Quién tiene 4 estómagos?
choices|random
choice|Una gallina
choice|Una vaca
choice|El Botija
choice|Una máquina tragamonedas
```

El archivo anterior es sólo una lista de registros, no tiene jerarquía. Para saber a quién pertenece cada registro se puede agregar un primer campo que indica el número de registro padre. El número de registro es su número de línea:

```
10|trivia|general
1|question|numeric|1000|0.1
2|text|Cuántos metros hay en un kilómetro?
1|question|single_choice|2|0.3
4|text|Quién tiene 4 estómagos?
4|choices|random
6|choice|Una gallina
6|choice|Una vaca
6|choice|El Botija
6|choice|Una máquina tragamonedas
```

Por ejemplo, la segunda línea tiene un registro de tipo `question` y el `1` al inicio indica que su registro padre es el que está en la línea 1 (`trivia`). En la tercera línea indica que hay un registro `text` que es el texto de la pregunta que está en la registro `2` y así sucesivamente.

El registro raíz, que siempre está en la línea 1, no tiene padre por lo no tiene sentido indicarlo. Sin embargo, se aprovecha este espacio para indicar un número muy útil: la cantidad de registros (líneas) en el archivo. Un invariante, y por tanto una restricción que siempre se debe cumplir es la siguiente: en un archivo de registros un registro padre tiene que aparecer antes que todos sus registros hijos. Por tanto, el registro raíz siempre es el primero del archivo de registros.

Si el archivo anterior se da a un programa, sabrá los valores de los campos pero no sus nombres. Para saber los nombres de los campos se le puede dar un archivo de registros aparte con esos nombres, a los cual se le llamaremos el **archivo de metadatos**:

```
|5|trivia|name
question|type|answer|difficulty
text|text
choices|order
choice|text
```

Por conveniencia se hace iniciar al archivo de metadatos con el separador de campos, de tal forma que el programa pueda usarlo para leer el resto del archivo de metadatos y el archivo de registros. Seguido al separador, un número indica la cantidad de tipos de registros en el archivo de metadatos, es decir, la cantidad de líneas de este archivo. Al igual que el archivo de registros, las líneas del archivo de metadatos indican el número de registro. Estos números pueden reemplazar el nombre del registro en el archivo de registros lo cual ayudaría a los programas a encontrar su tipo de registro de forma muy eficiente:

```
10|1|general
1|2|numeric|1000|0.1
2|3|Cuántos metros hay en un kilómetro?
1|2|single_choice|2|0.3
4|3|Quién tiene 4 estómagos?
4|4|random
6|5|Una gallina
6|5|Una vaca
6|5|El Botija
6|5|Una máquina tragamonedas
```

Este ejemplo se lee así: El archivo contiene 10 registros (líneas). El registro raíz es de tipo `1`, es decir, la primera línea del archivo de metadatos que corresponde al registro `trivia|name`. Le sigue un campo `name`, que al empatarlo con los datos se obtendrá `name=general`. El segundo registro del archivo de registros es hijo del registro `1`, por tanto, es hijo de `trivia`. Es de tipo `2` que corresponde a `question|type|answer|difficulty` en el archivo de metadatos. Al empatar los campos se obtendrán las parejas `type=numeric`, `answer=1000` y `difficulty=0.1`. Y así sucesivamente.

El archivo de registros resultante es más compacto, y mucho más rápido de cargar por un programa de computadora que sus correspondientes versiones en XML o JSON, en especial si la cantidad de datos es considerable. Se le pide que aplique sus conocimientos de programación para implementar un programa en Java que pueda cargar archivos de registros y sus correspondientes metadatos. Su programa deberá implementar lo siguiente.

Una clase `Metadata` que representa una línea del archivo de metadatos. Simplemente contiene un nombre de registro y un arreglo de nombres de campos (`String`). Es conveniente que tenga métodos para acceder al nombre del registro, a la cantidad de campos, al nombre de un campo por su índice, y un método para cargarse desde un objeto `Scanner`.

Una clase `Record` que representa un registro jerárquico del archivo de datos. Contiene una referencia a sus metadatos (`Metadata`), un arreglo de valores, y un arreglo de *registros* hijos. Es conveniente que reciba la referencia a los metadatos en el constructor, y que tenga un método para cargarse desde un objeto `Scanner` y otro método para agregar un registro a su arreglo de hijos.

Una clase `RecordLoader`. La clase tendrá dos arreglos, uno de metadatos (`Metadata`) y otro de datos (`Rercord`). Implementará un método para cargar metadatos, que recibe un objeto `Scanner` y retorna un arreglo de metadatos (puede usar `ArrayList`, si gusta). Implementará otro método para cargar datos, que recibe un objeto `Scanner` y retorna la raíz de un árbol de registros. En ambos métodos de cargado, use la cantidad de líneas proveniente del archivo para crear el arreglo del tamaño exacto. Implemente métodos de cargado en sus clases `Metadata` y `Record` que faciliten el cargado en `RecordLoader`.

Se quiere un programa que en la entrada estándar reciba un archivo de metadatos, un archivo de registros y lo convierta a tres potenciales formatos: texto puro, XML y JSON. En la primera línea se indica el formato destino deseado.

Ejemplo de entrada:

```
text

|5|trivia|name
question|type|answer|difficulty
text|text
choices|order
choice|text

10|1|general
1|2|numeric|1000|0.1
2|3|Cuántos metros hay en un kilómetro?
1|2|single_choice|2|0.3
4|3|Quién tiene 4 estómagos?
4|4|random
6|5|Una gallina
6|5|Una vaca
6|5|El Botija
6|5|Una máquina tragamonedas
```

El programa que hace las conversiones está parcialmente implementado, pero se requiere que implemente tres funciones recursivas en su clase `Record`: `toText()`, `toXML()` y `toJSON()`. El método `toText()` permite convertir el archivo de registros a formato de texto. Retorna un String resultado de concatenar todos los contenidos de texto (`nodeValue`) del registro en que se invoca y su descendencia. Recibe por parámetro un texto separador. Por ejemplo, si se usa como separador un cambio de línea (`"\n"`) producirá la siguiente salida:

Ejemplo de salida de texto:

```
Cuántos metros hay en un kilómetro?
Quién tiene 4 estómagos?
Una gallina
Una vaca
El Botija
Una máquina tragamonedas
```

[Opcional en papel, obligatorio en digital]. El segundo método recursivo de `Record` es `toXML()`. Recibe un entero que indica el nivel de anidamiento (indentación). Retorna un `String` resultado de convertir ese registro y registros hijos en texto XML, como se ve abajo. Cada registro se convierte a un elemento XML. Recuerde que un elemento se compone de tres partes: una etiqueta de apertura (opening tag), un contenido que puede ser texto u otros elementos (por simplicidad asuma que son excluyentes), y una etiqueta de cierre (closing tag).

Ejemplo de salida XML:

```
<trivia name="general">
   <question type="numeric" answer="1000" difficulty="0.1">
      <text>Cuántos metros hay en un kilómetro?</text>
   </question>
   <question type="single_choice" answer="2" difficulty="0.3">
      <text>Quién tiene 4 estómagos?</text>
      <choices order="random">
         <choice>Una gallina</choice>
         <choice>Una vaca</choice>
         <choice>El Botija</choice>
         <choice>Una máquina tragamonedas</choice>
      </choices>
   </question>
</trivia>
```

[Opcional disponible sólo en versión digital]. El tercer método recursivo de `Record` es `toJSON()`. Al igual que el método anterior, produce un `String` resultado de convertir el registro y sus hijos a formato JSON.

Código fuente dado:

```
import java.util.Scanner;

/** Reads record files and convert them to plain text, XML or JSON */
public class Solution
{
   /** Gets data from standard input */
   private Scanner input = null;

   /**
    * Start the execution of the solution
    * @param args Command line arguments
    */
   public static void main(String args[])
   {
      Solution solution = new Solution();
      solution.run();
   }

   /** Run the solution. This method is called from main() */
   public void run()
   {
      // Create object to read data from standard input
      this.input = new Scanner(System.in);

      // Read the target format and ignore the extra new line
      String targetFormat = this.input.nextLine();
      this.input.nextLine();

      // Create the object that will do the conversion
      RecordLoader recordLoader = new RecordLoader();

      // Read the metadata file and ignore the extra new line
      recordLoader.loadMetadata(this.input);
      this.input.nextLine();

      // Read the data file
      Record data = recordLoader.loadData(this.input);

      // Print the data according to the target format
      switch ( targetFormat )
      {
         case "text": System.out.print( data.toText("\n") ); break;
         case "xml" : System.out.print( data.toXML(0) );  break;
         case "json": System.out.print( data.toJSON(0) ); break;
      }

      // Close the standard input
      this.input.close();
   }
}
```