JAFramework 2 (JAF_2)

Manual del usuario

Introducción

Java Algorithm Framework (JAF_2) es un motor de persistencia creado para facilitar la enseñanza de programación de algoritmos implementados con el lenguaje Java.

JAF_2 aporta funcionalidad relacionada a la gestión de archivos de registros de longitud fija; problemática que Java no cubre directamente. Además JAF_2 pretende introducir al estudiante en el concepto de *framework*,

La operatoria de representación (*mapping*) que implementa JAF_2 es similar a la que utiliza Hibernate; cada archivo de registros al que se desea tener acceso desde un programa Java debe ser representado (*mappeado*) mediante una clase dentro de la cual, usando *annotations*, se indican todos los parámetros de relación entre dicha clase y el archivo físico que ésta representa.

JAF_2 permite manipular archivos de texto cuyos datos están almacenados en forma de "filas" y "columnas"; cada fila representa un registro y cada columna describe un campo o atributo de un registro. Luego, cada fila se compone de un conjunto de valores (atributos) que, por definición, son coherentes entre sí.

A continuación analizaremos un archivo de texto cuyo contenido respeta la estructura de "filas" y "columnas" antes mencionada. Este archivo lo utilizaremos durante todo el documento para analizar la funcionalidad de JAF_2.

En este archivo, cada registro tiene los siguientes campos: legajo, nombre, fecha de ingreso, sexo y nota.

ALUMNOS.txt

```
0010,Pedro     ,1991/08/23,M,08
0020,Amalia    ,1992/10/12,F,06
0030,Rolando   ,1993/11/01,M,09
0040,Carla     ,2001/03/21,F,10
0050,Osvaldo   ,2005/12/20,M,02
0060,Marisa    ,1990/03/15,F,04
```

Es muy importante observar que todas las columnas tienen exactamente el mismo ancho (*size*). El carácter separador puede ser cualquiera. En este caso se utiliza una ',' (coma) para separar los valores de los campos pero podría ser un ' ' (espacio en blanco), un carácter tabulador, etcétera.

Observemos también que el último registro debe tener un salto de línea (*ENTER*) al final; sólo uno. De este modo todos los registros del archivo tendrán exactamente la misma longitud; así el archivo podrá ser considerado como un archivo de registros de longitud fija y, por ende, podrá ser *mappeado* y accedido con JAF_2.

Operaciones básicas

Representar un archivo mediante una clase (*mapping*)

Para acceder al contenido de un archivo de registros de longitud fija, como es el caso de ALUMNOS.txt, debemos *mappearlo*; esto significa que tendremos que escribir una clase Java con tantos atributos como campos tiene el archivo; cada atributo con sus correspondientes métodos de acceso. Además se deben indicar algunos parámetros que JAF$_2$ utilizará para vincular el archivo (recurso físico) con la clase y con las instancias de ésta (recursos lógicos).

La clase `Alumno.java` que veremos a continuación implementa un *mappeo* básico del archivo ALUMNOS.txt. Observemos cómo utiliza la anotación `@File` para indicar el nombre físico del archivo y `@Field` sobre cada atributo para indicar el *size* (ancho de la columna) del campo que representa.

MUY IMPORTANTE: Los atributos de la clase deben aparecer **en el mismo orden** en que aparecen los campos del archivo.

Alumno.java

```
// package ...
// imports...

@File(name="ALUMNOS.txt", alias="ALUMNOS")
public class Alumno
{
   @Field(size=4)
   private int legajo;

   @Field(size=10)
   private String nombre;

   @Field(size=10)
   private String fechaIngreso;

   @Field(size=1)
   private char sexo;

   @Field(size=2)
   private int nota;

   @Override
   public String toString()
   {
      String linea = "";
      linea+="legajo="+legajo+ ", nombre="+nombre+", ";
      linea+="fechaIngreso="+fechaIngreso+", sexo="+sexo+", nota="+nota;
      return linea;
   }

   // :
   // setters y getters
   // :
}
```

Factoría y sesión de JAF$_2$

La clase `JAFactory` permite registrar los *mapping*s que vamos a utilizar; luego, nos dará la sesión a través de la cual podremos establecer el vínculo entre nuestro programa y los archivos de registros a los que vamos a acceder.

Comenzaremos registrando el único *mapping* que hasta ahora hemos desarrollado.

```
// registro los mappings
JAFactory.registerMapping(Alumno.class);
```

Habiendo registrado el *mapping* que representa el archivo al que vamos a acceder, estamos en condiciones de obtener la sesión.

```
// obtengo la session
JASession session = JAFactory.getSession();
```

Ahora que disponemos de la sesión podemos solicitar una referencia al archivo. Los archivos cuyos *mapping*s hemos registrado se representan con instancias de la clase `JAFile`.

```
// obtengo la referencia al archivo a traves de su alias
JAFile<Alumno> f = session.getFileByAlias("ALUMNOS");
```

Observemos que en el *mapping* Alumno.java, en annotation `@File` indicamos un alias para el archivo ALUMNOS.txt. En general se recomienda asignar un alias a cada uno de nuestros archivos pero en caso de no hacerlo, podemos acceder al archivo indicando su nombre físico como vemos a continuación:

```
// obtengo la referencia al archivo por su nombre fisico
JAFile<Alumno> f = session.getFile("ALUMNOS.txt");
```

Lectura secuencial de un archivo de registros

Como comentamos más arriba, cada archvo se compone de un conjunto de filas, todas con la misma cantidad de caracteres; el mismo *size*. La clase `JAFile` provee el método `read` y administra un indicador de posición que hace referencia a cual será el próximo registro al que podremos acceder invocando a dicho método.

```
// package ...
// imports...

public class RecorrerArchivo
{
   public static void main(String[] args)
   {
      JAFactory.registerMapping(Alumno.class);
      JASession session = JAFactory.getSession();

      // obtengo el archivo
      JAFile<Alumno> f = session.getFileByAlias("ALUMNOS");

      // muevo el indicador de posicion del archivo hacia el inicio
      f.reset();

      // creo una instancia de Alumno
      Alumno a = new Alumno();

      // el metodo read lee el proximo registro y retorna true si la lectura resulto correcta
      boolean ok = f.read(a);

      while( ok )
      {
         System.out.println(a);
```

```
         // leo el proximo registro
         ok = f.read(a);
      }

      // finalmente cierro el archivo
      f.close();
   }
}
```

Escritura secuencial sobre un archivo de registros

Así cómo el método `read` permite leer el contenido del próximo registro, que está siendo apuntado por el indicador de posición del archivo, el método `write` tiene un funcioamiento análogo; sólo que en lugar de leer escribe datos en dicho registro.

En el siguiente programa el usuario ingresa por consola datos que corresponden a varios alumnos; estos datos los usuamos para generar el archivo ALUMNOS.txt.

```
// package ...
// imports...

public class GrabarArchivo
{
   public static void main(String[] args)
   {
      Scanner scanner = new Scanner(System.in);

      JAFactory.registerMapping(Alumno.class);
      JASession session=JAFactory.getSession();

      // obtengo la referencia al archivo de alumnos
      JAFile<Alumno> f=session.getFileByAlias("ALUMNOS");

      // borro el contenido actual del archivo; lo dejo vacio
      f.rewrite();

      Alumno a = new Alumno();
      ingresarDatosAlumno(a);

      while( a.getLegajo()>0 )
      {
          // creo un registro y le asigno los datos ingresados
         a.setLegajo(leg);
         a.setNombre(nom);
         a.setNota(nota);

          // grabo el nuevo registro en el archivo
         f.write(a);

         ingresarDatosAlumno(a);
      }

      f.close();

      scanner.close();
   }

   public static void ingresarDatosAlumno(Alumno a)
   {
      Scanner scanner = new Scanner(System.in);
```

```
      System.out.println("-- INGRESO DE DATOS --");
      System.out.print("Legajo: ");
      int leg = scanner.nextInt();

      System.out.print("Nombre: ");
      String nom = scanner.next();

      System.out.print("Fecha de Ingreso: ");
      String fec = scanner.next();

      System.out.print("Sexo: ");
      char sexo = scanner.next();

      System.out.print("Nota: ");
      int nota = scanner.nextInt();

      a.setLegajo(leg);
      a.setNombre(nom);
      a.setFechaIngreso(fec);
      a.setSexo(sexo);
      a.setNota(nota);

      scanner.close();
   }
}
```

El programa anterior borra (si es que existía) todo el contenido del archivo ALUMNOS.txt. Si en lugar de borrar todo su contenido hubiésemos querido agregar los nuevos registros al final del archivo, entonces en lugar de invocar al método `rewrite` debemos invocar al método `seek` para mover al indicador del posición del archivo al final del último registro como veremos a continuación.

```
      // obtengo la referencia al archivo de alumnos
      JAFile<Alumno> f=session.getFileByAlias("ALUMNOS");

      // muevo el indicador al final del ultimo registro del archivo
      f.seek(f.sizeof());
```

Acceso directo a los registros del archivo

El método `seek` permite mover arbitráreamente el indicador de posición de un archivo. Veamos el siguiente programa que muestra cómo utilizando este método podemos leer cada uno de los registros de un archivo.

```
// package ...
// imports...

public class RecorrerArchivoSeek
{
   public static void main(String[] args)
   {
      // registro los mappings
      JAFactory.registerMapping(Alumno.class);

      // obtengo la session
      JASession session = JAFactory.getSession();

      // obtengo la referencia al archivo de alumnos
      JAFile<Alumno> f = session.getFileByAlias("ALUMNOS");
```

```
      int cantReg = f.fileSize();

      Alumno alumno = new Alumno();
      for(int i=0; i<cantReg; i++ )
      {
         f.seek(i);
         f.read(alumno);
         System.out.println(alumno);
      }

      f.close();
   }
}
```

Actualizar los valores de los registros

El siguiente programa actualiza las notas de todos los alumnos aprobándolos a todos; incluso, les suma 1 punto a aquellos que ya estaban aprobados.

```
// package ...
// imports...

public class ActualizarNotasAlumnos
{
   public static void main(String[] args)
   {
      JAFactory.registerMapping(Alumno.class);
      JASession session = JAFactory.getSession();

      JAFile<Alumno> f = session.getFileByAlias("ALUMNOS");
      f.reset();

      Alumno a = new Alumno();

      int i=0;
      while( f.read(a) )
      {
         int nuevaNota = Math.min(Math.max(4,a.getNota()+1),10);
         a.setNota(nuevaNota);

         f.seek(i);
         f.write(a);

         i++;
      }

      f.close();
   }
}
```

Indices

JAF2 permite definir índices e indexa automáticamente los archivos según las indicaciones que especifiquemos en el *mapping*. En el siguiente ejemplo indexamos el archivo ALUMNOS.txt por el campo `legajo`.

```
// package ...
// imports...

@File(name="ALUMNOS.txt", alias="EMPLEADOS")
@Index(key="legajo" alias="idx")
public class Alumno
{
   // ...
```

Recorrer un archivo indexado

Cuando tenemos un archivo indexado podemos recorrerlo a través de alguno de sus índices. Esto nos dará acceso a sus registros según el orden que dicho índices especifica.

```
// package ...
// imports...

public class RecorrerArchivoIndexado
{
   public static void main(String[] args)
   {
      JAFactory.registerMapping(Alumno.class);
      JASession session = JAFactory.getSession();
      JAFile<Alumno> f = session.getFileByAlias("ALUMNOS");

      // obtengo el indice
      JAIndex<Alumno> idx1 = session.getIndexByAlias(f,"idx");

      // ubico el indicador de posicion del indice en el primer registro
      idx1.reset();

      Alumno a = new Alumno();

      while( idx1.read(a) )
      {
         System.out.println(a);

         idx1.read(a);
      }

      idx1.close();
      f.close();
   }
}
```

Otras opciones para indexar el archivo ALUMNOS.txt podrían ser:

```
@Index(key="nombre+legajo" alias="idxNom")
@Index(key="-nota+nombre+legajo" alias="idxNota")
```

Busqueda

Sobre un archivo indexado podemos buscar y determinar si existe o no algún registro cuya clave coincida con un valor determinado. Veamos:

```
public class TestSearch
{
   public static void main(String[] args)
   {
      JAFactory.registerMapping(Alumno.class);
      JAFile<Alumno> f=session.getFileByAlias("ALUMNOS");

      JAIndex<Alumno> idx = session.getIndexByAlias(f,"idx");

      // defino la clave de busqueda
      Alumno k = new Alumno();
      k.setLegajo(30);

      // el metodo search retorna la posicion del registro que contiene
      // o un valor negativo se no se encontraron datos
      int pos  = idx.search(k);

      System.out.println(pos);

      if( pos>=0 )
      {
         idx.seek(pos);
         idx.read(k);

         System.out.println(k);
      }
      else
      {
         System.out.println("No se encontraron datos...");
      }
   }
}
```

Configurar múltiples índices

Un archivo puede indexarse por más de una clave, como vermos a continuación:

```
@File(name="ALUMNOS.txt", alias="ALUMNOS")
@Indexes( {@Index(key="legajo", alias="idxLeg")
         , @Index(key="nombre+legajo", alias="idxNom")
         , @Index(key="-nota+nombre+legajo", alias="idxNota")})
public class Alumno
{
   // ...
```

El siguiente programa recorre el archivo ALUMNOS.txt por cada uno de sus índices.

```
public class TestMultiplesIndices
{
   public static void main(String[] args)
   {
      JAFactory.registerMapping(Alumno.class);
      JASession session=JAFactory.getSession();
      JAFile<Alumno> f=session.getFileByAlias("ALUMNOS");
```

```
        // tomo cada uno de sus indices
        JAIndex<Alumno> iLeg= session.getIndexByAlias(f,"idxLeg");
        JAIndex<Alumno> iNom= session.getIndexByAlias(f,"idxNom");
        JAIndex<Alumno> iNot= session.getIndexByAlias(f,"idxNota");

        _mostrar(iLeg);
        _mostrar(iNom);
        _mostrar(iNot);

        f.close();
    }

    private static void _mostrar(JAIndex<Alumno> idx)
    {
        idx.reset();
        Alumno a = new Alumno();

        while( idx.read(a) )
        {
            System.out.println(a);
        }
    }
}
```