Manejo básico de archivos en Java

Hasta ahora todos los datos que creábamos en nuestros programas solamente existían durante la ejecución de los mismos. Cuando salíamos del programa, todo lo que habíamos generado se perdía.
A veces nos interesaría que la vida de los datos fuera más allá que la de los programas que los generaron. Es decir, que al salir de un programa los datos generados quedaran guardados en algún lugar que permitiera su recuperación desde el mismo u otros programas. Por tanto, querríamos que dichos datos fueran **persistentes**.
En este capítulo veremos el uso básico de archivos en Java para conseguir persistencia de datos. Para ello presentaremos conceptos básicos sobre archivos y algunas de las clases de la biblioteca estándar de Java para su creación y manipulación.
Además, el uso de esas bibliotecas nos obligará a introducir algunos conceptos "avanzados" de programación en Java: las excepciones, para tratar posibles errores durante la ejecución de un programa, y manipulación de datos a bajo nivel, para transformar nuestros datos a vectores de bytes.

1 concepto de archivo

Los programas usan variables para almacenar información: los datos de entrada, los resultados calculados y valores intermedios generados a lo largo del cálculo. Toda esta información es efímera: cuando acaba el programa, todo desaparece. Pero, para muchas aplicaciones, es importante poder almacenar datos de manera permanente.
Cuando se desea guardar información más allá del tiempo de ejecución de un programa lo habitual es organizar esa información en uno o varios ficheros almacenados en algún soporte de almacenamiento persistente. Otras posibilidades como el uso de bases de datos utilizan archivos como soporte para el almacenamiento de la información.

Los archivos desde el bajo nivel

Desde el punto de vista de más bajo nivel, podemos definir un archivo (o fichero) como:

> *Un conjunto de bits almacenados en un dispositivo, y accesible a través de un camino de acceso (pathname) que lo identifica.*

Es decir, un conjunto de 0s y 1s que reside fuera de la memoria del ordenador, ya sea en el disco duro, un pendrive, un CD, entre otros.

Esa versión de bajo nivel, si bien es completamente cierta, desde el punto de vista de la programación de aplicaciones, es demasiado simple.
Por ello definiremos varios criterios para distinguir diversas subcategorías de archivos. Estos tipos de archivos se diferenciarán desde el punto de vista de la programación: cada uno de ellos proporcionará diferentes funcionalidades (métodos) para su manipulación.

El criterio del contenido

Sabemos que es diferente manipular números que Strings, aunque en el fondo ambos acaben siendo bits en la memoria del ordenador. Por eso, cuando manipulamos archivos, distinguiremos dos clases de archivos dependiendo del tipo de datos que contienen:

- Los archivos de caracteres (o de texto)
- Los archivos de bytes (o binarios)

Un *fichero de texto* es aquél formado exclusivamente por caracteres y que, por tanto, puede crearse y visualizarse usando un editor. Las operaciones de lectura y escritura trabajarán con caracteres. Por ejemplo, los ficheros con código java son ficheros de texto.
En cambio un *fichero binario* ya no está formado por caracteres sino que los bytes que contiene pueden representar otras cosas como números, imágenes, sonido, etc.

El criterio del modo de acceso

Existen dos modos básicos de acceso a la información contenida en un archivo:

- Secuencial
- Acceso directo

En el *modo secuencial* la información del archivo es una secuencia de bytes (o caracteres) de manera que para acceder al byte (o carácter) i-ésimo se ha de haber accedido anteriormente a los i-1 anteriores. Un ejemplo de acceso secuencial lo hemos visto con la clase StringTokenizer.
El modo de *acceso directo* nos permite acceder directamente a la información del byte i-ésimo. Un ejemplo muy conocido de acceso directo lo tenemos con los vectores (arrays).

2

Tratamiento de errores: las excepciones

Las excepciones son un mecanismo que permite a los métodos indicar que algo "anómalo" ha sucedido que impide su correcto funcionamiento, de manera que quien los ha invocado puede detectar la situación errónea. Decimos en este caso, que el método ha **lanzado** (throw) una excepción. Cuando esto sucede, en vez de seguir con la ejecución normal de instrucciones, se busca hacia atrás en la secuencia de llamadas[1] si hay alguna que quiera **atraparla** (catch). Si ninguna de las llamadas decide atraparla, el programa acaba su ejecución y se informa al usuario del error que se ha producido la excepción y que nadie ha tratado.

Muchas de las excepciones que existen en Java, por ejemplo, dividir por 0, son **excepciones en tiempo de ejecución** (runtime exceptions) y no obligan a que el programador las trate explícitamente (claro que si el código no las trata y durante la ejecución del programa se producen, el programa finalizará con un "bonito" mensaje de error).

En Java, existe otro tipo de excepciones, las denominadas **excepciones comprobadas** (checked exceptions), que obligan al programador que dentro del código de un método invoca una instrucción que puede lanzarla a

- o bien atrapar dicha excepción (colocando dicha instrucción en un bloque **try-catch**)
- o bien, declarar en la cabecera del método que dicho método puede lanzar esa excepción (usando una declaración **throws**).

3

Sobre nombres de archivos, caminos de acceso y demás

Aunque a simple vista parezca una tontería, una de las cosas que más complica el código que trabaja sobre archivos no es la manipulación de su contenido sino la gestión de su nombre. El motivo es que cada sistema operativo usa convenciones diferentes para referirse a un nombre de fichero.

Por ejemplo, en sistemas tipo Unix tenemos:

```
/User/jmgimeno/Prog2/FileExample/src/Main.java
```

y en un sistema tipo Windows

```
C:\User\jmgimeno\Prog2\FileExample\src\Main.java
```

Así que hacer código que funcione independientemente del sistema es, cuando menos, tedioso.

Es por ello que, para simplificar, los nombres de ficheros que usaremos no contendrán camino alguno de acceso, lo que hará que estén ubicados en el directorio raíz del proyecto.

Si queréis aprender más sobre la manipulación de los nombres de fichero en java consultad la documentación de la clase la clase **java.io.File** que es la encargada de manipular nombres de archivo, rutas de acceso e incluso crear y listar directorios[2].

La declaración del fichero de entrada usando explícitamente la clase File sería:

```
1 FileReader input = new FileReader(new File(FILE_NAME));
```

4

Las clases FileReader y FileWriter permiten leer y escribir, respectivamente, en un fichero.

Después debemos crear un objeto de alguna de estas clases. Se pueden construir con un **objeto File o un String.**

Al crear un objeto de estas clases, deben estar dentro de un try-catch.

Recuerda que debemos controlar las excepciones.

Cuando creamos un objeto, abrimos un stream entre nuestro programa y el exterior, cuando debemos de usarlo debemos cerrar el stream con el método close().

Veamos un ejemplo, de creación de los objetos, también cerramos los streams:

```
FileWriter fw=new FileWriter("D:\\fichero1.txt");
FileReader fr=new FileReader("D:\\fichero1.txt");
```

Como vemos, escribimos la ruta del fichero. Si usas FileWriter y escribes una ruta de fichero que no existe lo crea, para FileReader si que debe existir el fichero, sino lanzara una excepción. Usamos doble barra (\\) por que es un carácter de escape, para poner \.

Ahora vamos a ver como escribir y leer texto en el fichero.

Veamos un ejemplo, primero escribiremos en el fichero y luego lo leemos:

```
import java.io.*;
//Importamos todas las clases de java.io.

public class FicheroTextoApp {
   public static void main(String[] args) {
      try{

         //Abro stream, crea el fichero si no existe
         FileWriter fw=new FileWriter("D:\\fichero1.txt");

         //Escribimos en el fichero un String y un caracter 97 (a)
         fw.write("Esto es una prueb");
         fw.write(97);

         //Cierro el stream
         fw.close();

            //Abro el stream, el fichero debe existir
         FileReader fr=new FileReader("D:\\fichero1.txt");

         //Leemos el fichero y lo mostramos por pantalla
         int valor=fr.read();
         while(valor!=-1){
            System.out.print((char)valor);
            valor=fr.read();
         }

         //Cerramos el stream
         fr.close();

      }catch(IOException e){
         System.out.println("Error E/S: "+e);
      }
   }
}
```

Otros métodos interesantes de FileReader

Si los buscáis están definidos en la clase InputStreamReader que es extendida por FileReader

- **int read(char[] buf, int offset, int length)**
  Este método lee como máximo length caracteres del archivo y los coloca en el vector buf a partir de la posición offset. Devuelve el número de caracteres leídos, o -1 indicando la finalización del archivo.
- **int read(char[] buf)**
  Como la anterior pero usando 0 como offset i buf.length como length.
- **String getEncoding()**
  Devuelve el nombre del sistema de codificación usado para convertir los 0s y 1s del fichero en caracteres.

Otros métodos interesantes de FileWriter

- **new FileWriter(String name, boolean append)**
  En caso de que ya existe un archivo de nombre name, si el booleano append es cierto, los nuevos caracteres se añadirán al fichero a partir del final. Si no, se creará el fichero vacío y se empezarán a añadir desde el principio.
- **void write(char[] cbuf, int off, int len)**
  Escribe len caracteres del vector cbuf a partir de la posición off en el archivo.
- **void write(char[] cbuf)**
  Como la anterior pero usando 0 como off y cbuf.length como len.
- **void write(String str, int off, inte len)**
  Igual que el anterior, pero en vez de un vector de caracteres tenemos un String.
- **void write(String str)**
  Como la anterior pero usando 0 como off y str.length() como len.

5

Sobre las codificaciones de caracteres

Un tema que también soslayaremos es el de las codificaciones usadas para representar los caracteres y que es otra de las grandes complicaciones existentes al tratar ficheros.
El problema es simple de enunciar:

- existen diversas maneras de asignar a un carácter[3] un patrón de bits (que es lo que acaba siendo leído o escrito en un fichero)
- está claro que para que todo funcione correctamente, quién escribe un fichero y quien lo lee han de usar el mismo criterio

En Java existen varias clases para representar estas codificaciones, y versiones de los constructores de ficheros que permiten elegir la codificación a usar.
Nosotros no indicaremos codificación alguna y, si generamos los ficheros en la misma máquina que los consumimos, no deberíamos tener problema alguno.

6

El problema de los saltos de línea

Otro de los problemas al manipular ficheros de forma uniforme entre sistemas operativos es que éstos utilizan diversos caracteres para indicar el final de una línea.
Tal y como indica la documentación de readLine, un fin de línea puede estar indicado por:

- el carácter line-feed ('\n')
- el carácter carriage-return ('\r')
- el carácter carriage-return seguido inmediatamente de line-feed

El primer caso se usa en sistemas tipu Unix, el segundo en las versiones antiguas de MacOS y el último en sistemas tipo Windows.
El método newLine escribe el final de línea usando la convención del sistema operativo de la máquina en el que se está ejecutando.

7 El concepto de Buffering

El concepto de buffering queda muy bien explicado en el siguiente párrafo extraído del libro Head First Java:

> *Si no hubiera buffers, sería como comprar sin un carrito: debería llevar los productos uno a uno hasta la caja. Los buffers te dan un lugar en el que dejar temporalmente las cosas hasta que está lleno. Por ello has de hacer menos viajes cuando usas el carrito.*

Cualquier operación que implique acceder a memoria externa es muy costosa, por lo que es interesante intentar reducir al máximo las operaciones de lectura/escritura que realizamos sobre los ficheros, haciendo que cada operación lea o escriba muchos caracteres. Además, eso también permite operaciones de más alto nivel, como la de leer una línea completa y devolverla en forma de cadena.

Leer un texto

```
public void leenos() {
 BufferedReader intermedio;
    try {//abrimos comunicación (buffer)

intermedio= new BufferedReader (new FileReader("C:\\Users\\c\\Desktop\\c/Prueba
lectura.txt"));
            String text_linea="";
            while(text_linea!= null) {
                text_linea = intermedio.readLine();
                System.out.println(text_linea);
            }
        } catch (IOException e) {

System.out.println("Mira si el archivo se encuentra en su sitio porque yo no lo veo");
            e.printStackTrace();
        }
    }
```

Escribir un texto

Para generan un buffer necesitaremos la clase BufferWriter y su método write().

```
public void CrearArchivoEscrito(){

String TEXTO = ("Este texto aparecerá en el nuevo archivo si todo lo hice bien");
        FileWriter archivoTexto; BufferedWriter Archivo_leido;
        try {
Archivo_leido =new BufferedWriter( new FileWriter("C:\\Users\\s\\v\\esto se creo por
java.txt"));
                Archivo_leido.write(TEXTO);
         } Archivo_leido.close();//CERRAMOS ARCHIVO
catch (IOException e) {
            e.printStackTrace();
        }

    }
```

Copiar un archivo y pegarlo en otro:

```
public void leenos() {
      BufferedReader intermedio;BufferedWriter Archivo_leido;
        try {//abrimos comunicación (Stream)
intermedio= new BufferedReader (new FileReader("C:\\Users\\x\\Desktop\\Prueba
lectura.txt"));
Archivo_leido =new BufferedWriter( new FileWriter("C:\\Users\\d\\Desktop\\esto se creo por
java.txt"));
            String text_linea="";
            while(text_linea!= null) {
                text_linea = intermedio.readLine();
if (text_linea != null)Archivo_leido.write(text_linea + "\r");
}intermedio.close(); Archivo_leido.close();  //cerramos el buffer
        } catch (IOException e) {
System.out.println("Mira si el archivo se encuentra en su sitio porque yo no lo veo");
            e.printStackTrace();
        }
    }
}
```

8 Ficheros secuenciales binarios en Java

Existen las clases FileInputStream y FileOutputStream para ficheros binarios en Java. Como hemos visto hasta ahora, podemos usar los ficheros de texto para almacenar o leer información, pero su mayor desventaja es que al ser legible, puede ser modificado por terceras personas.

Por ejemplo, cuando vemos un fichero que tiene una extensión concreta, solo puede leerse con un programa específico. Si abres un documento Word desde un bloc de notas, verás muchos caracteres no legibles, esto es porque Word sabe como leer esos caracteres.

La forma de crear un fichero binario es igual que vimos con FileReader y FileWriter, veamos un ejemplo:

```
FileOutputStream fos=new FileOutputStream("D:\\fichero_bin.dat");

FileInputStream fis=new FileInputStream("D:\\fichero_bin.dat");
```

Vamos a ver un ejemplo de como escribir en el fichero, usaremos el método write con el que podemos usar un numero que corresponderá a la tabla ASCII o un array de bytes. Crearemos un fichero con una extensión propia.

```
import java.io.*;
public class FicherosBinariosApp {

    public static void main(String[] args) {

        try(FileOutputStream fos=new FileOutputStream("D:\\fichero_bin.dat")){

            String texto="Esto es una prueba para ficheros binariosssss";

            //Copiamos el texto en un array de bytes
            byte codigos[]=texto.getBytes();

            fos.write(codigos);

        }catch(IOException e){

        }

    }
}
```

Ahora veremos como se lee, es prácticamente igual que vimos con FileReader, usando el método read(), cuando llega al final del fichero devuelve -1. Su diferencia básica es que con FileReader leemos caracteres y FileInputStream lee bytes.

```
import java.io.*;
public class FicherosBinariosApp {

    public static void main(String[] args) {

        try(FileInputStream fis=new FileInputStream("D:\\fichero_bin.dat")){

            int valor=fis.read();
            while(valor!=-1){
                System.out.print((char)valor);
                valor=fis.read();
            }

        }catch(IOException e){

        }

    }

}
```

9

Manipulación de datos a bajo nivel

Hasta ahora nuestros programas han estado manipulado números enteros, números en coma flotante, caracteres y Strings y en ningún momento[5] hemos comentado cómo estos datos están representados internamente (por ejemplo, cuánto valen los bits correspondientes a un determinado número entero).
Conocer esta representación nos será útil tanto a nivel conceptual, para entender las diferencias entre ficheros de texto y ficheros binarios, como en la práctica, para leer y escribir datos en formato binario.

Tamaños de los tipos primitivos en Java

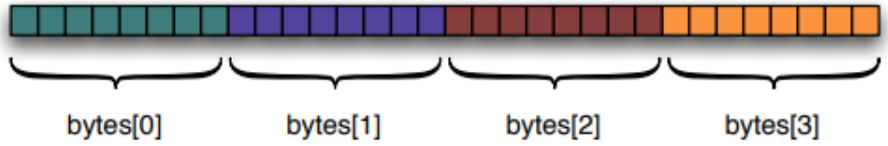
Una de las ventajas de Java es que la representación de los datos no depende ni de la arquitectura de la máquina en la estamos trabajando ni de su sistema operativo (es una de las ventajas de usar una máquina virtual). Por ello, lo que diremos funcionará en cualquier máquina y sistema operativo.
El estándar de Java define los siguientes tamaños para los tipos de datos primitivos:

Tipo primitivo	Tamaño	Valor mínimo	Valor máximo
`byte`	8-bits	-128	127
`char`	16-bits	Unicode 0	Unicode 2^{16}-1
`short`	16-bits	-2^{15} (-32.768)	$+2^{15}$-1 (32.767)
`int`	32-bits	-2^{31} (-2.147.483.648)	$+2^{31}$-1 (2.147.483.647)
`long`	64-bits	-2^{63}	$+2^{63}$-1
`float`	32-bits	32 bits IEEE-754	
`double`	64-bits	64 bits IEEE-754	
`boolean`	indefinido	true OR false	

10

¿Cómo obtener los bytes correspondientes a un entero?

Lo que queremos es conseguir una función tal que dado un entero nos lo convierta en un array formado por los cuatro bytes de su representación, es decir:



Para ello lo que haremos será usar combinaciones de :

- **(byte):** convierte un entero a bytes. Como un entero tiene más de un byte, se queda con los ocho bits menos significativos. Es necesario hacer una conversión explícita ya que al convertir de `int` a `byte` podemos perder precisión.
- **b>>8**: desplazar 8 bits a la derecha, para hacer que el siguiente byte ocupe ahora la posición más a la derecha. Lo mismo sucede al desplazar con 16 y 26 para los siguientes bytes.

En código:

```
1 private static byte[] toByteArray(int n) {
2     byte[] bytes = new byte[4];
3     bytes[0] = (byte) (n >> 24);
4     bytes[1] = (byte) (n >> 16);
5     bytes[2] = (byte) (n >> 8);
6     bytes[3] = (byte) (n);
7 return bytes;
```

11

Utilidades de empaquetamiento

Una vez entendido el caso sobre los enteros, vamos a ver la una clase cuyos métodos implementan esta transformación para otros tipos de datos básicos.

Para cada tipo de datos tendremos dos funciones: una para **empaquetar** los valores de ese tipo (packX) y otra para **desempaquetar** los valores del mismo (unpack).

En todos los casos, pasaremos como parámetros:

- un **byte[] buffer** para leer/escribir los datos
- un **int offset**, que indicará la posición inicial a partir de la que leeremos/escribiremos.

Por ejemplo, si realizamos la llamada:

```
PackUtils.packInt(416, buffer, 12);
```

se empaquetarán los 4 bytes correspondientes al entero 416 en las posiciones buffer[12], buffer[13], buffer[14] y buffer[15].

Para simplificar, las funciones **no comprueban** si accedemos a posiciones correctas del vector.

Si ahora hiciéramos

```
int n = PackUtils.unpackInt(buffer, 12);
```

el valor de n debería de ser 416.

SE ADJUNTA EN ANEXO A ESTE DOCUMENTO LA CLASE PACKUTILS, DE UTILIDAD PARA CONVERSIÓN DE DISTINTOS TIPOS DESDE/HACIA BYTES

12

Archivos binarios de acceso directo

Los archivos de acceso directo están representados por la clase `java.io.RandomAccessFile`, que permite:

- Abrir un archivo en el que se pueda solamente leer (modo "r") como tanto leer como escribir (modo "rw").
- Para leer disponemos de las operaciones:
  - **int read(byte[] buff, int off, int len)**
  - **int read(byte[] buff)**
- Para escribir disponemos de las operaciones:
  - **void write(byte[] buff, int off, int len)**
  - **void write(byte[] buff)**
- La operación que caracteriza a este tipo de archivos es:
  - **void seek(long pos)**

  que coloca la posición de lectura/escritura en el byte que ocupa la posición pos del archivo.
  Así, la siguiente operación de lectura, en vez de usar la posición dejada por la última operación, usará dicha posición como la del primer byte a leer.
  Puede colocarse más allá del final del fichero, pero el tamaño del fichero no aumentará hasta que se haya realizado una operación de escritura.

- Otros métodos relevantes de la clase son:
  - **long length(),** que devuelve el número de bytes ocupado por el archivo.
  - **void setLength(long newLength),** que define la nueva longitud del archivo como newLength. En caso de ser menor que la longitud actual, el contenido del fichero es truncado. Por ejemplo, setLength(0) hace que las subsiguientes operaciones de escritura se realicen sobre un archivo vacío.
  - **long getFilePointer(),** devuelve el valor de la posición de lectura/escritura del archivo. Puede ser útil para saber si, p.e. estamos intentado leer más allá del último registro del archivo.
- Aunque también provee de operaciones de transformación de tipos básicos a vectores de bytes y viceversa, nosotros usaremos las que hemos definido en la clase `PackingUtils`, ya que usar las predefinidas obligaría entrar en algunos detalles[7] que se escapan al contenido del curso.

13

Uso típico de archivos binarios de acceso directo

Las posibilidades de

- mezclar operaciones de escritura con operaciones de lectura
- acceder a una posición concreta del archivo

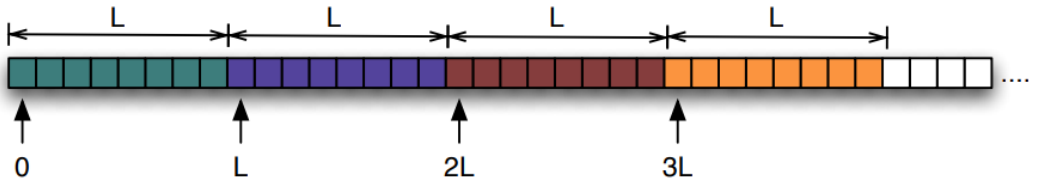
hacen que el uso principal de los archivos de acceso directo sea implementar algo muy parecido a los arrays, pero en memoria secundaria.

El concepto de registro

Si queremos guardar en un archivo de acceso directo los datos correspondientes a las instancias de una clase para poder acceder directamente a cada una de las instancias, deberemos hacer que todas ellas tengan **igual longitud**. De esta manera, si cada instancia tiene longitud L, la instancia i-ésima ocupará L bytes a partir del i*L. Gráficamente:

El concepto de registro

Si queremos guardar en un archivo de acceso directo los datos correspondientes a las instancias de una clase para poder acceder directamente a cada una de las instancias, deberemos hacer que todas ellas tengan **igual longitud**. De esta manera, si cada instancia tiene longitud L, la instancia i-ésima ocupará L bytes a partir del i*L. Gráficamente:



14

Ejemplo: un archivo de acceso directo de personas

Vamos a mostrar todo lo que hemos comentado sobre archivos de acceso directo sobre un ejemplo concreto: un archivo para representar personas.

La clase que representa los datos

La primera clase que veremos es la que representa una persona. Esta clase contiene:

- campos con información de la persona
- operaciones sobre personas, básicamente getters y el método toString
- operaciones para:
  - dada una persona, obtener un array de bytes
  - dado un array de bytes, construir la persona

Comentarios sobre las líneas destacables:

- **3-6**: declaramos los campos que tendrá cada instancia de Person
- **8**: como los registros han de ser de longitud fija, limitamos la longitud del String name a NAME_LIMIT caracteres. De hecho la limitación afectará solamente cuando leamos/escribamos los datos.
- **9**: en la constante SIZE, calculamos el tamaño que tendrá cada registro asociado a una persona en función de los campos a guardar. Como el tamaño ha de poderse conocer desde fuera, hacemos la constante pública y, **para evitar que se pueda asignar otro valor, la definimos como final**.
- **42-54**: este método, crea un array de bytes de tamaño SIZE y va empaquetando cada uno de los campos a partir de los offsets que le corresponden.
- **56-68**: operación inversa que desempaqueta el array de bytes que recibe como parámetro y crea una instancia con los valores obtenidos. Fijaos en que es un método estático, ya que claramente se refiere a cosas relacionadas con la clase Person, pero no se aplica sobre ninguna instancia (de hecho, es el propio método quién crea una instancia).

```
50      PackUtils.packInt(age, record, offset);
51      offset += 4;
52      PackUtils.packBoolean(married, record, offset);
53      return record;
54    }
55
56    public static Person fromBytes(byte[] record) {
57      int offset = 0;
58      long id = PackUtils.unpackLong(record, offset);
59      offset += 8;
60      String name = PackUtils.unpackLimitedString(NAME_LIMIT,
61                                                  record,
62                                                  offset);
63      offset += 2 * NAME_LIMIT;
64      int age = PackUtils.unpackInt(record, offset);
65      offset += 4;
66      boolean married = PackUtils.unpackBoolean(record,offset);
67      return new Person(id, name, age, married);
68    }
69
70  }
```

El programa principal

En el programa principal lo que haremos es:

- declarar una referencia al archivo de acceso directo
- crear varia personas
- escribir y leer en diversas posiciones del archivo, indexando por registro.

Comentarios de las líneas relevantes:

- **3**: declaramos el archivo de acceso aleatorio como un campo de la clase, así todos los métodos no estáticos de la misma lo podrán utilizar.
- **5-10:** escribe en el archivo el registro con posición num formado con los datos de person.
- **12-17:** lee el registro del posición num del archivo y crea una instancia de Person con los bytes obtenidos.
- **19-56:** escribimos y leemos en diferentes posiciones del archivo.

```
1  public class Main  {
2
3    private RandomAccessFile raf;
4
5    private void writePerson(long num, Person person)
6                                              throws IOException {
7      this.raf.seek(num * Person.SIZE);
8      byte[] record = person.toBytes();
9      this.raf.write(record);
```

```
10    }
11
12    private Person readPerson(long num) throws IOException {
13      this.raf.seek(num * Person.SIZE);
14      byte[] record = new byte[Person.SIZE];
15      this.raf.read(record);
16      return Person.fromBytes(record);
17    }
18
19    public void run() {
20      try {
21
22        this.raf = new RandomAccessFile("people.dat", "rw");
23
24        Person p1 = new Person(4671, "Juan", 40, false);
25        Person p2 = new Person(1819, "Pedro", 63, true);
26        Person p3 = new Person(7823, "María", 18, false);
27        Person p4 = new Person(8984, "Susi", 24, true);
28
29        this.writePerson(0, p1);
30        this.writePerson(1, p2);
31        this.writePerson(4, p3);
32
33        Person p;
34
35        p = this.readPerson(0);
36        System.out.println("p = " + p);
37
38        p = this.readPerson(1);
39        System.out.println("p = " + p);
40
41        p = this.readPerson(4);
42        System.out.println("p = " + p);
43
44        this.writePerson(3, p4);
45        p = this.readPerson(3);
46        System.out.println("p = " + p);
47
48        this.writePerson(1, p1);
49        p = this.readPerson(1);
50        System.out.println("p = " + p);
51
52      } catch (IOException e) {
53        println("Algo muy malo ha pasado :-(");
54      }
55    }
56  }
```

La ejecución del programa muestra:

```
p = Person{id=4671 name=Juan age=40 married=false}
p = Person{id=1819 name=Pedro age=63 married=true}
p = Person{id=7823 name=María age=18 married=false}
p = Person{id=8984 name=Susi age=24 married=true}
p = Person{id=4671 name=Juan age=40 married=false}
```

Visualizando el contenido del fichero

Un archivo binario no lo podemos visualizar con un editor de texto
Para ver su contenido, podemos usar la herramienta UNIX hexdum
que nos muestra los valores de los bytes del fichero.
Si lo aplicamos al fichero generado por el programa anterior,
obtenemos:

```
CleoBook >:  hexdump -C people.dat
0000000 00 00 00 00  00 00 12 3f  00 4a 00 75  00 61 00 6e
0000010 00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00
0000020 00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00
0000030 00 00 00 28  00 00 00 00  00 00 00 12  3f 00 4a 00
0000040 75 00 61 00  6e 00 00 00  00 00 00 00  00 00 00 00
0000050 00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00
0000060 00 00 00 00  00 00 00 00  28 00 00 00  00 00 00 00
0000070 00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00
0000080 00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00
0000090 00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00
00000a0 00 00 00 00  00 23 18 00  53 00 75 00  73 00 69 00
00000b0 00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00
00000c0 00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00
00000d0 00 00 18 01  00 00 00 00  00 00 1e 8f  00 4d 00 61
00000e0 00 72 00 ed  00 61 00 00  00 00 00 00  00 00 00 00
00000f0 00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00
0000100 00 00 00 00  00 00 00 12  00
0000109
```

Comentarios:

- En **amarillo** he marcado los 8 bytes correspondientes al id, que es un long. En el caso del primer registro (que empieza en la posición 0), el valor es 0x123f, que podéis comprobar que es lo mismo que 4671.
- En **azul** están los 40 bytes correspondientes a name, dónde cada par de bytes corresponde a un carácter. 0x4a corresponde al carácter 'J', 0x75 a 'u', etc.
- En **rojo** los 4 bytes correspondientes a age. Podéis comprobar que 0x28 es 40.
- En **verde** está el byte correspondiente al booleano married que, valiendo 0x00, es falso.
- En **magenta** he marcado los bytes correspondientes al registro 2, que en ningún momento se ha escrito. **Leer un registro que no contiene datos es un error.**

ANEXO CLASE PACKULTILS

```
1 public class PackUtils {
2
3   public static void packBoolean(boolean b,
4                                  byte[] buffer,
5                                  int offset) {
6     if (b) {
7       buffer[offset] = (byte) 1;
8     } else {
9       buffer[offset] = (byte) 0;
10    }
11  }

12  public static boolean unpackBoolean(byte[] buffer,
13                                      int offset) {
14    return buffer[offset] == (byte) 1;
15  }
16
17  public static void packChar(char c,
18                              byte[] buffer,
19                              int offset) {
20    buffer[offset    ] = (byte) (0xFF & (c >> 8));
21    buffer[offset + 1] = (byte) (0xFF &  c      );
22  }
23
24  public static char unpackChar(byte[] buffer, int offset) {
25    return (char) ((buffer[offset] << 8) |
26                   (buffer[offset + 1] & 0xFF));
27  }
28
29  public static void packLimitedString(String str,
30                                       int maxLength,
31                                       byte[] buffer,
32                                       int offset) {
33    for (int i = 0; i < maxLength; i++) {
34      if ( i < str.length() ) {
35        packChar(str.charAt(i), buffer, offset+2*i);
```

```
36      } else {
37        // We mark with a zero
38        packChar('\0', buffer, offset+2*i);
39        break;
40      }
41    }
42  }
43
44  public static String unpackLimitedString(int maxLength,
45                                           byte[] buffer,
46                                           int offset) {
47    String result = "";
48    for (int i = 0; i < maxLength; i++ ) {
49      char c = unpackChar(buffer, offset+2*i);
50      if ( c != '\0' ) {
51        result += c;
52      } else {
53        break;
54      }
55    }
56    return result;
57  }
58
59  public static void packInt(int n,
60                             byte[] buffer,
61                             int offset ) {
62    buffer[offset    ] = (byte) (n >> 24);
63    buffer[offset + 1] = (byte) (n >> 16);
64    buffer[offset + 2] = (byte) (n >>  8);
65    buffer[offset + 3] = (byte)  n       ;
66  }
67
68  public static int unpackInt(byte[] buffer, int offset) {
69    return ((buffer[offset    ]       ) << 24) |
70           ((buffer[offset + 1] & 0xFF) << 16) |
71           ((buffer[offset + 2] & 0xFF) <<  8) |
72           ((buffer[offset + 3] & 0xFF)      ) ;
73    }
74
75  public static void packLong(long n,
76                              byte[] buffer,
77                              int offset)  {
78    buffer[offset    ] = (byte) (n >> 56);
79    buffer[offset + 1] = (byte) (n >> 48);
80    buffer[offset + 2] = (byte) (n >> 40);
81    buffer[offset + 3] = (byte) (n >> 32);
82    buffer[offset + 4] = (byte) (n >> 24);
83    buffer[offset + 5] = (byte) (n >> 16);
84    buffer[offset + 6] = (byte) (n >>  8);
```

```
85      buffer[offset + 7] = (byte)  n       ;
86    }
87
88    public static long unpackLong(byte[] buffer, int offset) {
89      return ((long)(buffer[offset    ]       ) << 56) |
90             ((long)(buffer[offset + 1] & 0xFF) << 48) |
91             ((long)(buffer[offset + 2] & 0xFF) << 40) |
92             ((long)(buffer[offset + 3] & 0xFF) << 32) |
93             ((long)(buffer[offset + 4] & 0xFF) << 24) |
94             ((long)(buffer[offset + 5] & 0xFF) << 16) |
95             ((long)(buffer[offset + 6] & 0xFF) <<  8) |
96             ((long)(buffer[offset + 7] & 0xFF)      ) ;
97    }
98
99    public static void packDouble(double n,
100                                  byte[] buffer,
101                                  int offset) {
102     long bits = Double.doubleToRawLongBits(n);
103     packLong(bits, buffer, offset);
104   }
105
106   public static double unpackDouble(byte[] buffer,
107                                     int offset) {
108     long bits = unpackLong(buffer, offset);
109     return Double.longBitsToDouble(bits);
110   }
111 }
```