Manejo básico de archivos en Java (v1)

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. El 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. Los archivos desde Java

En Java, los distintos tipos de ficheros se diferencian por las clases que usaremos para representarlos y manipularlos. Como las clases que usaremos pertenecen a la biblioteca estándar del lenguaje, su uso es

algo más complejo que las de las clases de la ACM, ya que su diseño se ha realizado pensando en su uso industrial.
Las clases que usaremos para el tratamiento de ficheros están ubicadas en el paquete **java.io** por lo que deben ser importadas.
Además, el código que trabaja con archivos ha de considerar que muchas cosas pueden ir mal cuando se trabaja con ellos: el archivo puede estar corrupto, alguien ha desconectado el pendrive a medio ejecutar el programa, es un disco en red y ésta ha caído, o no tiene más espacio para almacenar información, etc.
Es por ello que, aunque de forma breve, deberemos introducir el mecanismo estándar en Java para tratar con los errores que pueden darse en nuestro programas: las excepciones.

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**).

[1] El concepto de secuencia de llamadas, de hecho la **pila de llamadas**, se presenta en el tema de recursividad cuando se describe en detalle la ejecución de un programa recursivo.

El objetivo es hacer que el programador no pueda "olvidarse" de tratar las muchas situaciones anómalas que se puedan producir durante la ejecución de un programa.

Tratamiento simplificado de excepciones

No es tema propio de esta asignatura profundizar en el manejo de excepciones, así que lo que veremos será un **tratamiento muy simplificado** de las mismas, haciendo lo mínimo para que Java dé por correctos nuestros programas.
Para ello, el código que manipule los ficheros, tendrá la siguiente estructura:

```
1 try {
2   Codigo que abre y trata el fichero
3 } catch (IOExcepction ex) {
4   Código que trata el error
5 }
```

La idea intuitiva de esta construcción es: **intenta** (try) ejecutar esas instrucciones y, en caso de producirse un error en el tratamiento de los ficheros (se ha lanzado una **IOException**), **atrapa** (catch) ese error y ejecuta el código de corrección. Nosotros simplificaremos el código de tratamiento del error y solamente escribiremos un mensaje.
Si en vez de tratar el error internamente, queremos indicar que un método puede lanzar excepciones, en su cabecera pondremos:

```
1 public int methodThatCanThrow(params) throws IOException {
2
3   Código que trata ficheros pero no atrapa IOException
4
5 }
```

Como todos los ejemplos que veremos sobre ficheros utilizan estos mecanismos, no añadiremos aquí ejemplos de su utilización.

3. Lectura de ficheros secuenciales de texto

De cara a presentar la manipulación de ficheros secuenciales de texto, presentaremos un problema y su solución y, sobre la solución, comentaremos las operaciones que hemos usado.

Problema: contar apariciones de diversas letras

El problema consistirá en: dado un fichero de texto, contar el número de veces que aparecen una serie de letras en él.
Como siempre, lo difícil es la estrategia, en este caso:

- Supondremos que tanto el nombre del fichero, como los caracteres a considerar son constantes en el programa.
- Leeremos cada uno de los caracteres hasta llegar al último.
- Si está en los caracteres a considerar, incrementamos el contador asociado a ese carácter.

Como la parte de contar no tiene demasiado que ver con lo de los ficheros, lo mejor es separarlo en otra clase (cuyo diseño e implementación quedará como ejercicio).
Con estas ideas, la solución del problema sería:

```
 1 public class CountingVocals extends ConsoleProgram {
 2
 3   private static String FILE_NAME = "input.txt";
 4   private static String VOCALS = "aeiou";
 5
 6   public void run() {
 7     try {
 8        CharCounter counters = new CharCounter(VOCALS, false);
 9        FileReader input = new FileReader(FILE_NAME);
10        int c = input.read();
11        while ( c != -1 ) {
12          counters.countIfTargeted((char) c);
13          c = input.read();
14        }
15        input.close();
16        println(counters.toString());
17     } catch (IOException ex) {
18        println("Something bad has happended :-(");
19     }
20   }
21 }
```

Comentemos las líneas más relevantes:

- **3-4:** Definimos las constantes para el nombre de fichero y para las vocales a contar.
- **7 y 17-19**: Como cualquiera de las instrucciones que manipulan el fichero puede dar un error, encerramos todo el código del run en un bloque try-catch.
- **8:** creamos un contador para los caracteres del String dado (en este caso las vocales). El parámetro booleano indica que no queremos ignorar diferencias entre mayúsculas y minúsculas.
- **9**: creamos una instancia de **FileReader** para leer los caracteres del fichero. En este punto decimos que el fichero está abierto y preparado para que leamos caracteres de él.

- **10:** el método **read()** lee el siguiente carácter en el fichero de entrada. Para poder indicar que se ha llegado al final del fichero, en vez de devolver un carácter, devuelve un entero. Así, puede usar el **valor -1 para indicar que no quedan más caracteres por leer**[2].
- **11-14**: mientras no hemos llegado al final del fichero hemos de tratar el carácter actual y leer el siguiente.
- **12**: contamos el carácter actual (ya que sabemos que no era el final del fichero). Toda la parte de saber si es uno de los caracteres a considerar ya la hará la clase CharCounter.
- **13**: leemos el siguiente carácter en la entrada.
- **15:** después de haber tratado todo el fichero lo cerramos usando en método **close().** Esto es especialmente importante cuando escribimos, pero mantener abiertos ficheros que ya no necesitamos malgasta recursos del sistema.
- **16**: escribimos los contadores.

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[3].

---

[2] Fijaos en que -1 es un entero válido, pero no un carácter, por lo que podemos usarlo como indicador de final de fichero.

[3] Y es un ejemplo de mala elección de nombre, ya que lo que trata son o bien los nombres de ficheros (debería llamarse FileName) o, desde un punto de vista de bajo nivel, las denominadas entradas del sistema de ficheros del sistema operativo (por lo que FileEntry también sería un buen nombre).

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

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

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.

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[4] un patrón de bits (que es lo que acaba siendo leído o escrito en un fichero)
- está claro que para que todo funcione correctamente, quien 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.

[4] Similares problemas suceden en el caso de otros tipos de datos como int, double, etc.

La versión “avanzada” del mismo problema

En la solución anterior, hemos simplificado el tratamiento de los errores a lo mínimo que hay que hacer para lograr que el programa sea un programa Java correcto.
Pero que el programa sea correcto no quiere decir que la solución sea perfecta. En este apartado os mostraré cómo sería el tratamiento de excepciones correcto y la forma idiomática en Java de hacer la lectura.
Primero el programa:

```
2 public class CountingVocals extends ConsoleProgram {
3
4   private static final String FILE_NAME = "input.txt";
5   private static final String VOCALS = "aeiou";
6
7   public void runAdvanced() {
8     FileReader input = null;
9     try {
10      CharCounter counters = new CharCounter(VOCALS, false);
11      input = new FileReader(FILE_NAME);
12      int c;
13      while ( (c = input.read()) != -1 ) {
14         counters.countIfTargeted((char) c);
15      }
16      println(counters);
17    } catch (FileNotFoundException ex) {
18      println("Problems opening " + FILE_NAME);
19    } catch (IOException ex) {
20      println("Problems reading " + FILE_NAME);
21    } finally {
22      try {
23        file.close();
24      } catch (IOException ex) {
25        println("Problems closing " + FILE_NAME);
26      }
27    }
28  }
29 }
```

4. Escritura de ficheros secuenciales de texto

Como veremos, los conceptos son similares, tan sólo cambia la clase de fichero (ahora es FileWriter) y, en vez de leer, escribimos.

Problema: dada una cadena de texto, escribirla al revés en un fichero

La estrategia en este caso es:

- Pedir una cadena al usuario.
- Recorrerla de atrás hacia delante e ir escribiendo en el fichero los caracteres que vamos encontrando
- Al final, cerrar el fichero.

Es decir:

```
1 public class BackwardsWriter extends ConsoleProgram {
2
3   private static String FILE_NAME = "backwards.txt";
4
5   public void run() {
6     try {
7       String text = readLine("Enter a text: ");
8       FileWriter output = new FileWriter(FILE_NAME);
9       for (int i = text.length()-1; i >= 0; i--) {
10          output.write(text.charAt(i));
11      }
12      output.close();
13    } catch (IOException ex) {
14      println("Something bad has happended :-(");
15    }
16  }
17 }
```

Comentemos las líneas más relevantes:

- **6 y 13:** para simplificar usaremos la misma estructura de bloque try-catch que en el caso de la lectura.
- **8**: ahora para manipular el fichero usaremos una instancia de **FileWriter** (ya que escribiremos en él).
- **10**: aquí es dónde escribimos un nuevo carácter en el fichero usando en método **write(int).** Recordad que siempre que me piden un int puedo usar un char.
- **12**: cierro el fichero (si no lo hacemos pudiera ser que algunos de los caracteres no se acabaran guardando en el fichero).

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. 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.

Problema: crear un Howler a partir de un texto

Un Howler, en el universo de Harry Potter, no es más que un correo que chilla. Como chillar en internet es escribir en mayúsculas, lo que vamos a hacer es un programa tal que dado el texto de un mail (separado en líneas), lo "howlerice" y lo convierta en mayúsculas. Como siempre, la solución:

```
1 public class HowlerMaker extends ConsoleProgram {
2
3   private static String MAIL_NAME = "mail.txt";
4   private static String HOWLER_NAME = "howler.txt";
5
6   private String howlerize(String text) {
7       return text.toUpperCase();
8   }
9
10  public void run() {
11    try {
12      BufferedReader input =
13        new BufferedReader(new FileReader(MAIL_NAME));
14      BufferedWriter output =
15        new BufferedWriter(new FileWriter(HOWLER_NAME));
```

```
16        String line = input.readLine();
17        while (line != null) {
18          String howledLine = this.howlerize(line);
19          output.write(howledLine);
20          output.newLine();
21          line = input.readLine();
22        }
23        input.close();
24        output.close();
25      } catch (IOException ex) {
26        println("MAybe you know who has come :-(");
27      }
28    }
29  }
```

Comentemos, como siempre, las líneas más relevantes:

- **12,13:** el constructor del BufferedReader en vez de recibir el nombre del fichero, recibe una instancia de FileReader. La idea es que la clase BufferedReader se centra en manejar un buffer de caracteres y cuando ha de leer cosas del fichero usa la instancia de FileReader para hacerlo[5].
- **14,15:** equivalente para BufferedWriter.
- **16:** leemos una línea entera en forma de String. En el String que nos devuelve, el marcador de fin de línea está eliminado. Para indicar que no hay más líneas devuelve null.
- **19:** usamos una versión de write que nos permite escribir una cadena.
- **20:** para que la salida tenga los mismos saltos de línea, hemos de añadirlo usando el método newLine (recordad que readLine lo había eliminado de line, por lo que al pasarla a mayúsculas en howledLine no lo tiene).

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’)

[5] Cuando el año que viene estudiéis el tema de la herencia, veréis que la clase BufferedReader puede usarse para hacer buffering de caracteres que vienen desde otros tipos de reader como son los que obtienen caracteres via comunicaciones en red.

- 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.

6. Archivos binarios de acceso directo

Este tipo de archivos sirve para guardar información codificada como grupos de bytes. Por esa razón, previo al tema de tratamiento de ficheros, estudiamos la manera de convertir valores de tipos primitivos y Strings, a arrays de bytes.
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").
  - **new RandomAccessFile(String name, String mode)**
- 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[6] que se escapan al contenido del curso.

Uso típico de archivos binarios de acceso directo

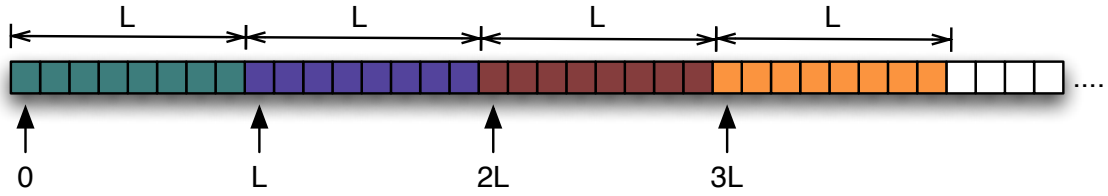
Las posibilidades de

- 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:



A cada uno de los bloques de bytes que representan los datos de una instancia se le denomina **registro**.

Este objetivo de conseguir registros de igual longitud es el que obliga a que el empaquetamiento/desempaquetamiento de cadenas de caracteres se haga limitando su longitud máxima.

[6] Para conocerlos, consultad la documentación de la clase `RandomAccessFile` y de las interfícies asociadas `DataInput` y `DataOutput`.

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).

```
1 public class Person {
2
3   private long id;
4   private String name;
5   private int age;
6   private boolean married;
7
8   private static final int NAME_LIMIT = 20;
```

```
 9  public static final int SIZE = 8 + 2 * NAME_LIMIT + 4 + 1;
10
11  public Person(long id,
12                String name,
13                int age,
14                boolean married) {
15    this.id = id;
16    this.name = name;
17    this.age = age;
18    this.married = married;
19  }
20
21  public int getAge() {
22    return age;
23  }
24
25  public long getId() {
26    return id;
27  }
28
29  public boolean isMarried() {
30    return married;
31  }
32
33  public String getName() {
34    return name;
35  }
36
37  public String toString() {
38    return "Person{" + "id=" + id + " name=" + name +
39           " age=" + age + " married=" + married + '}';
40  }
41
42  public byte[] toBytes() {
43    byte[] record = new byte[SIZE];
44    int offset = 0;
45    PackUtils.packLong(id, record, offset);
46    offset += 8;
47    PackUtils.packLimitedString(name, NAME_LIMIT,
48                                record, offset);
49    offset += 2 * NAME_LIMIT;
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 extends ConsoleProgram {
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      println("p = " + p);
37
38      p = this.readPerson(1);
39      println("p = " + p);
40
41      p = this.readPerson(4);
42      println("p = " + p);
43
44      this.writePerson(3, p4);
45      p = this.readPerson(3);
46      println("p = " + p);
47
48      this.writePerson(1, p1);
49      p = this.readPerson(1);
50      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 hexdump, que nos muestra los valores de los bytes del fichero.
Si lo aplicamos al fichero generado por el programa anterior, obtenemos:

```
CleoBook >: hexdump -v 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.**

7. Ordenación de archivos: el algoritmo MergeSort

Un procedimiento muy habitual a realizar sobre archivos es ordenarlos. Aunque para el caso de los archivos de acceso directo podríamos aplicar lo que ya conocemos para ordenar vectores (arrays), no es la mejor forma de hacerlo, ya que las lecturas/escrituras no secuenciales en archivos son muy costosas.
El algoritmo MergeSort permite ordenar un archivo solamente realizando lecturas y escrituras secuenciales, por lo que es muy eficiente en el uso de los archivos.
En este apartado veremos, en primer lugar, la lógica detrás del algoritmo y una implementación para ordenar las líneas de un archivo de texto. Dicha versión no es la más eficiente y se dejará como ejercicio su mejora.

La idea básica del algoritmo

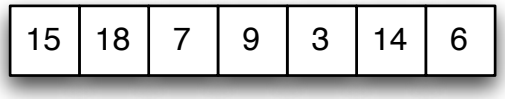
¿Os acordáis de la película de Los Inmortales? En ella se explica que los inmortales, sienten una atracción irresistible de acabar unos con otros y que al final **solamente puede quedar uno**.
La idea detrás del algoritmo MergeSort es muy simple: ir fusionando las partes ordenadas que contiene el fichero, hasta que todo él está en una única parte.
Para ello utiliza en dos métodos auxiliares:

- **split** (separar): que dado un fichero de entrada y dos de salida, va colocando las subsecuencias ordenadas del fichero de entrada alternativamente en cada uno de los de salida.
- **merge** (fusionar): que dados dos ficheros de entrada y uno de salida, fusiona dos subcadenas, una de cada fichero de entrada, y las guarda en el fichero de salida.

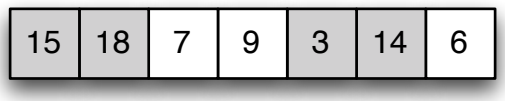
Ambas operaciones se sucederán hasta que el fichero que queremos dividir conste de una única secuencia.

Visualización del MergeSort

Visualizaremos el algoritmo sobre un fichero de números enteros. Para hacerlo supongamos que deseamos ordenar el siguiente fichero:

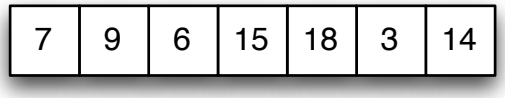


Marcaremos, alternando dos colores, las subsecuencias del fichero que ya están ordenadas. En este caso:

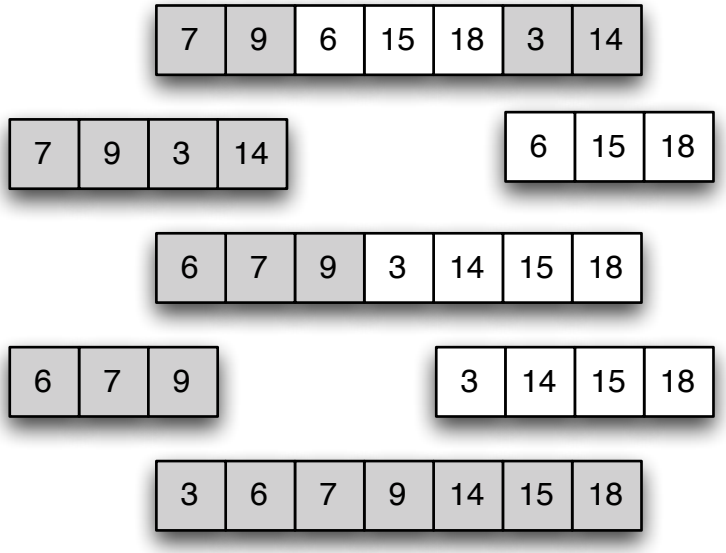


Separando (**split**) por colores en dos ficheros auxiliares, obtenemos:

Si fusionamos (**merge**) ordenadamente ambos ficheros (es decir, los recorremos en paralelo y copiamos en el fichero resultado el elemento más pequeño), queda:



Si repetimos el mismo procedimiento hasta que solamente haya una subsecuencia, obtenemos:



Que, como podemos ver, deja el fichero ordenado.

Implementación en Java

Vamos a aplicar la idea del algoritmo para ordenar las líneas de un archivo de texto. El resultado lo dejaremos en el fichero de texto original.

```
1 public class MergeSort extends ConsoleProgram {
2
3   private static final String INPUT = "input.txt";
4   private static final String AUX1  = "aux1.txt";
5   private static final String AUX2  = "aux2.txt";
6
7   public void run() {
8     try {
9       boolean sorted = split(INPUT, AUX1, AUX2);
10      while (!sorted) {
11        merge(AUX1, AUX2, INPUT);
12        sorted = split(INPUT, AUX1, AUX2);
```

```
13      }
14      println("Yataaaa!!!");
15    } catch (IOException ex) {
16      println("Some error has happened");
17    }
18  }
19
20  private boolean split(String input,
21                        String output1,
22                        String output2) throws IOException {
23
24    BufferedReader in = new BufferedReader(
25                                new FileReader(input));
26    BufferedWriter out = new BufferedWriter(
27                                new FileWriter(output1));
28    BufferedWriter other = new BufferedWriter(
29                                new FileWriter(output2));
30
31    boolean sorted = true;
32    String previous = "";
33    String current = in.readLine();
34
35    while (current != null) {
36      if (previous.compareTo(current) > 0) {
37        sorted = false;
38        BufferedWriter tmp = out;
39        out = other;
40        other = tmp;
41      }
42      out.write(current);
43      out.newLine();
44      previous = current;
45      current = in.readLine();
46    }
47
48    in.close();
49    out.close();
50    other.close();
51
52    return sorted;
53  }
54
55  private void merge(String input1,
56                     String input2,
57                     String output) throws IOException {
58
59    BufferedReader in1 = new BufferedReader(
60                                  new FileReader(input1));
61    BufferedReader in2 = new BufferedReader(
```

```
62                                  new FileReader(input2));
63      BufferedWriter out = new BufferedWriter(
64                                  new FileWriter(output));
65
66      String current1 = in1.readLine();
67      String current2 = in2.readLine();
68
69      while (current1 != null && current2 != null) {
70        if (current1.compareTo(current2) <= 0) {
71          out.write(current1);
72          out.newLine();
73          current1 = in1.readLine();
74        } else {
75          out.write(current2);
76          out.newLine();
77          current2 = in2.readLine();
78        }
79      }
80
81      while (current1 != null) {
82        out.write(current1);
83        out.newLine();
84        current1 = in1.readLine();
85      }
86
87      while (current2 != null) {
88        out.write(current2);
89        out.newLine();
90        current2 = in2.readLine();
91      }
92
93      in1.close();
94      in2.close();
95      out.close();
96    }
97  }
```

Algunos comentarios sobre el código:

- **7-18**: el programa principal es el que realiza la ordenación. Inicialmente hace un split y mientras el fichero no está ordenado, hace un merge seguido de un split.
- Fijaros en que el fichero de entrada para split es de salida para merge, y los de salida para split son los de entrada para merge.
- recordad que split, mientras encuentra valores ordenados, los va guardando en el mismo fichero, para ello guardamos dos valores:
  - previous, que es el valor anteriormente guardado
  - current, el valor que se va a aguardar actualmente

- **32**: inicializamos previous a la cadena vacía ya que así es menor que cualquier otra cadena, por lo que la primera cadena que leamos forma una subsecuencia ordenada con ella.
- **38-40**: cuando hemos de cambiar de fichero lo que hago es intercambiar las referencias out y other, y siempre escribo en out.
- **31,37**: inicialmente supongo que el fichero de entrada está ordenado, pero si alguna vez detecto un par de elementos consecutivos que están desordenados sé que no lo está.
- **69-79**: si hay elementos en ambos ficheros de entrada, se comparan, y el menor se copia en el de salida.
- **81-85, 87-91**: cuando uno de los ficheros acaba, copiamos los que quedan en el otro en el de salida.

Mejora del algoritmo

El algoritmo que hemos presentado es simple, pero realiza un número de pasos de split-merge excesivo (a cada paso el número de subsecuencias ordenadas del fichero decrece no mucho más que en una unidad). Como los archivos suelen ser grandes, su uso en ficheros reales sería inadecuado.
La idea de la optimización es muy simple, y se basa en **decrecer de forma más rápida** el número de subsecuencias que se encuentran en el fichero. Como vimos en el tema de recursividad, una forma rápida de decrecer, es a cada paso, **dividir por la mitad** el tamaño del problema a resolver (en este caso el tamaño del problema sería el número de subsecuencias ordenadas que tiene el fichero que, cuando es uno, indica que el fichero ya está ordenado).
Para ello:

- en el paso **split**, en vez de tener en cuenta si dos elementos están desordenados, separamos por bloques de tamaño fijo. En el primer split el tamaño es 1, en el siguiente 2, en el siguiente 4, etc.
    - en una implementación real del algoritmo no partiríamos de bloques de tamaño 1, sino que definiríamos un tamaño mínimo que permita su ordenación rápida en memoria principal. Así minimizaríamos el número de operaciones que realizamos sobre memoria secundaria.
- en el paso de **merge**, mezclamos los elementos, pero manteniendo los grupos. Es decir, el primer merge fusionará un grupo de tamaño 1 del primer fichero con otro de tamaño 1 del segundo fichero, así hasta acabarlos. En el siguiente merge se

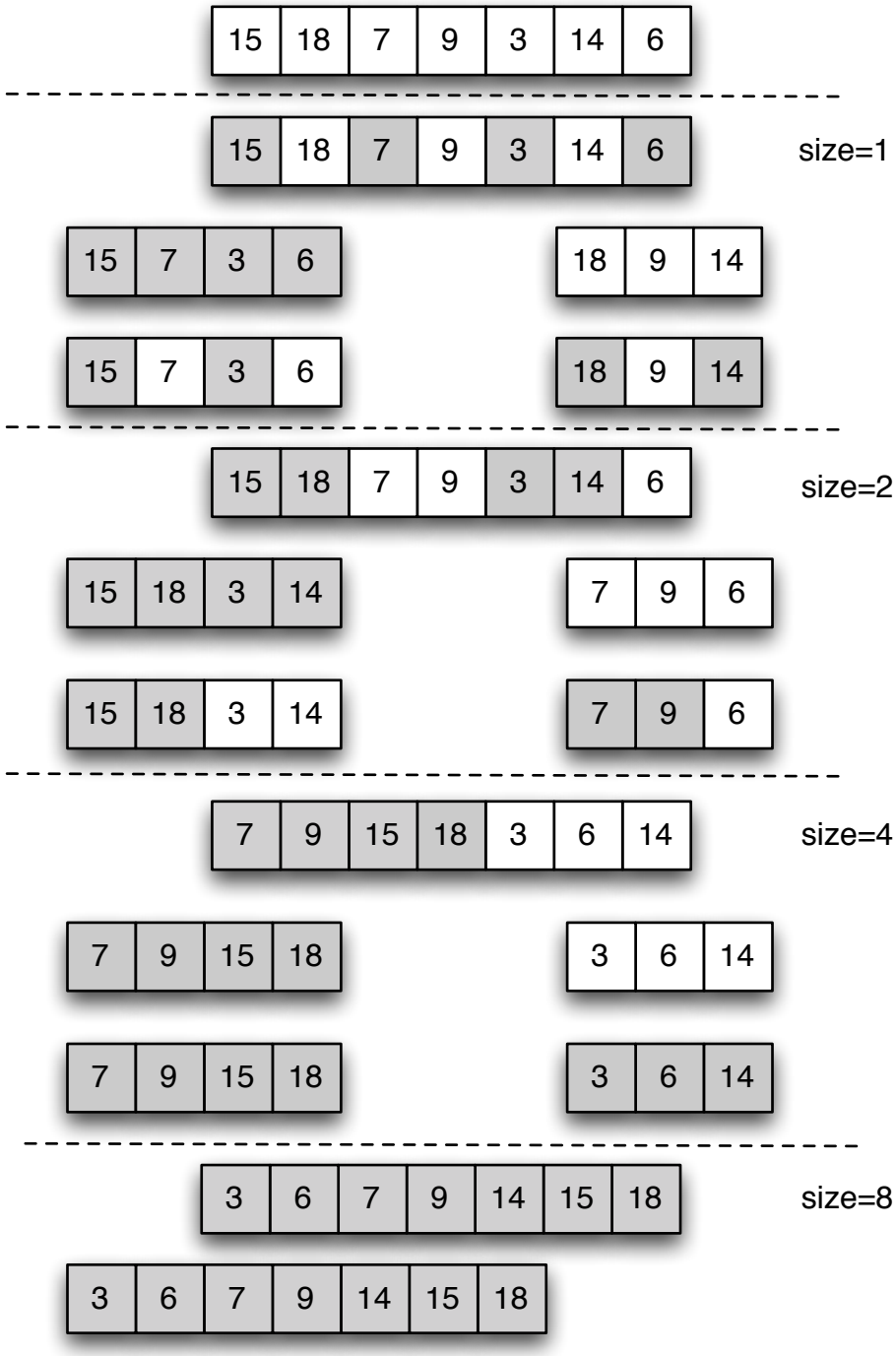
fusionarán un grupo de tamaño 2 del primer fichero, con otro de tamaño 2 del segundo fichero, hasta que no queden más grupos.

- la secuencia de split y merge se irá repitiendo, **doblando cada vez el tamaño de los grupos**, hasta que solamente quede uno.

Fijaos en que, si procedemos de esta manera: en cada paso de split+merge nos aseguramos que **el número de subsecuencias ordenadas es la mitad que en el paso anterior** (o la longitud de éstas es el doble).

¿Cuál es el número máximo de pasos que podemos realizar? Si a cada paso doblamos el tamaño de las subsecuencias ordenadas, después de logaritmo en base 2 pasos obtendremos un secuencia ordenada que contenga tantos elementos como la secuencia original.

Queda como ejercicio la implementación de esta versión del algoritmo (**que es la que se conoce realmente como MergeSort**).



8. Bibliografía

- Eric S.Roberts, The Art & Science of Java, Addison-Wesley (2008).
- The Java Tutorials (última consulta, 30 de mayo de 2011).
- The Java Language Specification (Third edition) (última consulta, 30 mayo 2011).
- Kathy Sierra & Bert Bates, Head First Java, O'Reilly (2003).