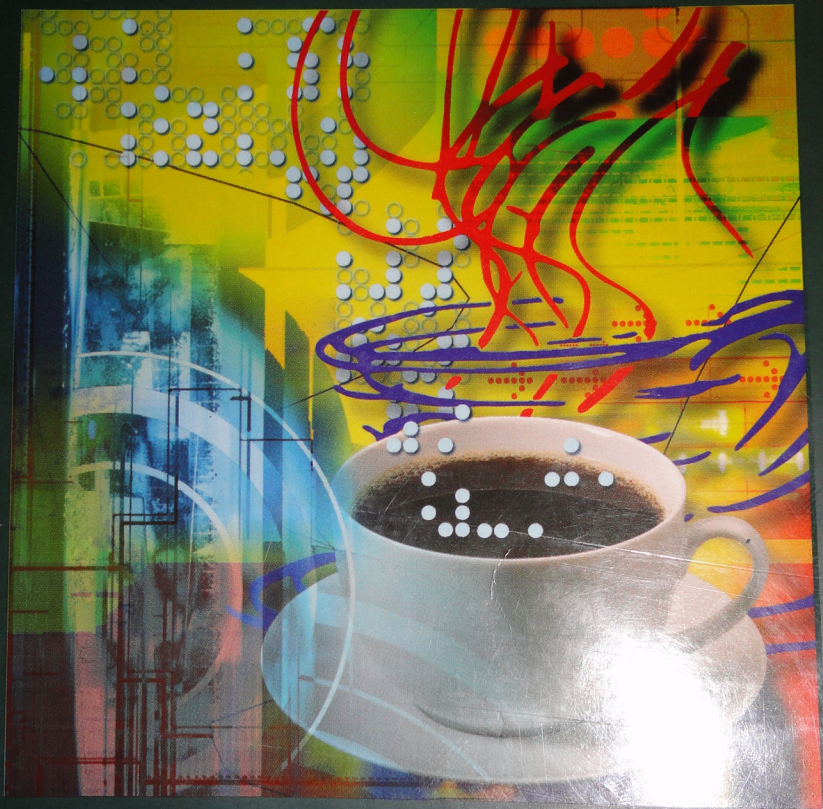


JAVA™ 2

Manual de usuario y tutorial

5ª edición



Agustín Froufe Quintas



Incluye CD-ROM
con el código completo
de los ejemplos

Alfaomega  Ra-Ma®

FICHEROS EN JAVA

En otros capítulos se muestra cómo recuperar e introducir cadenas de texto desde el teclado en una interfaz de usuario; pero hay que reconocer que lo interesante es hacerlo en ficheros y bases de datos, que permiten un almacenamiento permanente de la información. No obstante, la aproximación que se va a realizar de los ficheros no va a ser tan exhaustiva como la realizada con otros conceptos de Java, ya que se supone que el lector, a estas alturas, conoce las bases del funcionamiento de los sistemas de ficheros, así que se van a presentar ejemplos de acceso a ficheros y también algunos conceptos específicos de Java, como la forma de permitir que Java acceda a ciertos ficheros a través de applets o servlets y cómo restringir a una aplicación el acceso a ficheros. Todas estas operaciones, presentes desde la creación de Java, se han visto mejoradas, e incluso reescritas, mediante el uso más optimizado de tareas en la versión actual de la plataforma Java 2. Incluso algunas de las operaciones que los programadores echaban en falta, se han incorporado en la arquitectura de entrada/salida de Java, **NIO**.

LA CLASE FILE

Antes de realizar acciones sobre un fichero, es necesario introducir la clase **File** que proporciona muchas utilidades relacionadas con ficheros y con la obtención de información básica sobre los mismos.

Creación de un objeto File

Para crear un objeto **File** nuevo, se puede utilizar cualquiera de los constructores que se enumeran:


```

File miFichero;
miFichero = new File( "/home/kk" );
    O
miFichero = new File( "/home","kk" );
    O
File miDirectorio = new File( "/home" );
miFichero = new File( miDirectorio,"kk" );

```

El constructor utilizado depende a menudo de otros objetos **File** necesarios para el acceso. Por ejemplo, si sólo se utiliza un fichero en la aplicación, el primer constructor es el mejor. Si, en cambio, se utilizan muchos ficheros desde un mismo directorio, el segundo o tercer constructor será más cómodo. En caso de que el directorio o fichero sean variables, el tercer constructor será el más indicado de los tres.

Comprobaciones y utilidades

Una vez creado un objeto **File**, ya se puede reunir información sobre el fichero, la plataforma Java 2 proporciona muchos métodos: para obtener el nombre del fichero, para comprobar los permisos de lectura y escritura, para obtener las últimas modificaciones, para recuperar información general sobre el fichero, etc.

El ejemplo Java901.java es un programa muy simple que muestra información sobre los ficheros pasados como argumentos en la línea de comandos.

```

class Java901 {
    public static void main( String args[] ) throws IOException {
        // Se comprueba que nos han indicado algún fichero
        if( args.length > 0 ) {
            // Vamos comprobando cada uno de los ficheros que se hayan
            // pasado en la línea de comandos
            for( int i=0; i < args.length; i++ ) {
                // Se crea un objeto File para tener una referencia al
                // fichero físico del disco
                File f = new File( args[i] );
                // Se presenta el nombre y directorio donde se encuentra
                System.out.println( "Nombre: "+f.getName() );
                System.out.println( "Camino: "+f.getPath() );
                // Si el fichero existe se presentan los permisos de lectura
                // y escritura y su longitud en bytes
                if( f.exists() ) {
                    System.out.print( "Fichero existente" );
                    System.out.print( (f.canRead() ?
                        " y se puede Leer" : "" ) );
                    System.out.print( (f.canWrite() ?
                        " y se puede Escribir" : "" ) );
                    System.out.println( "." );
                    System.out.println( "La longitud del fichero es de "+
                        f.length()+" bytes" );
                }
                else
                    System.out.println( "El fichero no existe." );
            }
        }
    }
}

```

```
else
    System.out.println( "Debe indicar un fichero." );
}
```

Permisos y control de acceso

Java proporciona varios métodos en la clase **File** que permiten modificar los permisos de acceso a ficheros. Si el lector es habitual de sistemas Unix, estos métodos le resultarán familiares, porque son semejantes a la funcionalidad que proporciona el comando *chmod* de Unix.

Los permisos de acceso a ficheros son los que corresponden a la lectura, escritura y ejecución. El sistema de ficheros puede asignar cualquiera de estos permisos a un único objeto, y así mismo, un conjunto de estos permisos al propietario del fichero o a otros usuarios. Los métodos que proporciona Java corresponden a cada uno de los tres permisos sobre los ficheros y tienen dos signaturas: la primera que recibe un único parámetro de tipo booleano para indicar si ese permiso se asigna o no y, la segunda que recibe dos parámetros, el primero igual que en el caso anterior y además, un segundo parámetro para indicar si el permiso afecta solamente al propietario del fichero, si se pasa a *true*, o a todos los usuarios si se fija como *false*. En el caso de sistemas de ficheros que no distinguen entre permisos de propietario y genéricos, como es el caso de los sistemas de ficheros de Windows, este parámetro no tiene efecto y los permisos que se fijan afectarán a todos los usuarios.

Los métodos para asignar permisos, en sus dos formas, son los siguientes:

```
public boolean setExecutable( boolean ejecutar );
public boolean setExecutable( boolean ejecutar, boolean propietario );
public boolean setReadable( boolean leer );
public boolean setReadable( boolean leer, boolean propietario );
public boolean setWritable( boolean escribir );
public boolean setWritable( boolean escribir, boolean propietario );
```

La invocación de los métodos anteriores devuelve *false* si el usuario que los ejecuta no tiene autorización para cambiar los permisos de acceso al fichero. Cada uno de los métodos puede generar una excepción de tipo **SecurityException** en el caso de que se haya definido un controlador de seguridad, en el cual los métodos *checkExecute()*, *checkRead()* y *checkWrite()*, controlan el acceso al fichero.

La clase **File** también proporciona los métodos *canExecute()*, *canRead()* y *canWrite()* para poder comprobar los permisos asignados a un fichero y poder conocer las acciones que se pueden realizar sobre él.


```
JTextField campoTexto;
```

El constructor se limita a colocar los distintos componentes *Swing* sobre el panel, así que no merece la pena detenerse en su descripción, porque en capítulos posteriores de tratarán en profundidad. Pero sí resulta interesante el método *actionPerformed()*, que es el encargado de controlar las pulsaciones del ratón. En este método se utilizan objetos de tipo **FileInputStream** y **FileOutputStream** para leer y escribir datos a fichero; estas clases manejan los datos como bytes en vez de como caracteres.

Las dos líneas siguientes muestran la declaración de la variable que obtiene la cadena de texto introducida por el usuario en la ventana y el array de bytes en la que se almacena.

```
String texto = campoTexto.getText();  
byte b[] = texto.getBytes();
```

A continuación, se crea un objeto **File** para el fichero que se va a grabar y que será el que se utilice para crear el objeto **FileOutputStream** a través del cual se enviarán los datos al fichero.

```
File fichSalida = new File( dir +  
    File.separatorChar+ "textoe.txt" );  
FileOutputStream canalSalida = new FileOutputStream( fichSalida );
```

Para concluir la grabación de la cadena en el dispositivo de almacenamiento, se escribe el array de bytes a través del canal de salida y se cierra la conexión con el fichero; tal como reproducen las siguientes dos líneas de código.

```
canalSalida.write( b );  
canalSalida.close();
```

El código encargado de la lectura del fichero de texto es similar. Primero se crea un objeto de tipo **File** que es utilizado para crear un canal de entrada desde él. El código siguiente reproduce esa situación en el ejemplo.

```
File fichEntrada = new File( dir +  
    File.separatorChar+ "texto1.txt" );  
FileInputStream canalEntrada = new FileInputStream( fichEntrada );
```

Luego hay que crear un array de bytes del mismo tamaño que el fichero sobre el cual se va a leer el contenido de ese fichero.

```
byte bt[] = new byte[(int)fichEntrada.length()];  
int numBytes = canalEntrada.read( bt );
```

Y ya, el array de bytes es el origen de un objeto **String** que es el que se utiliza para crear la etiqueta que se presenta en la ventana. El canal de comunicación con el fichero se cierra una vez que se ha pasado el contenido del array a la etiqueta.

```
String cadena = new String( bt );  
etiqueta.setText( cadena );
```

LA ARQUITECTURA NIO

La funcionalidad de entrada/salida en Java siempre ha estado en manos de *streams*, bien de tipo *byte* o de tipo *character*. Sin embargo, los *streams* no proporcionan toda la funcionalidad necesaria para los programadores, sino solamente para operaciones básicas de lectura y escritura y, además, son bloqueantes; es decir, una lectura de un **InputStream** siempre intentará devolver un byte, esperando incluso a que esté disponible si es necesario, o estén disponibles el mismo número de bytes que se hayan solicitado. De igual modo, las operaciones de salida también escriben siempre todos los datos que se pasen. ➤

Este comportamiento no siempre es el deseado, porque es posible querer leer datos si están disponibles, pero seguir haciendo cosas en caso contrario, esperando la llegada de esos datos. Por ejemplo, si se trata de un reproductor de música, mientras espera a que llegue otro paquete de datos, puede ir ejecutando lo que ya se haya leído. El uso de *buffers* para lectura y escritura supone un paso más allá de las limitaciones anteriores, pero tienen el inconveniente de que no es fácil su programación cuando se necesitan realizar operaciones complejas, porque los buffers están ocultos al programador. Es decir, o el programador implementa la lógica de procesamiento de buffers en base a la lectura de arrays de bytes o utiliza un **BufferedReader**, teniendo en cuenta que solamente podrá extraer objetos de tipo **String**.

La arquitectura de entrada/salida de Java **nio** intenta en parte solucionar esos problemas. Para ello proporciona una serie de clases de tipo **Buffer** que tratan los arrays de bytes y se aprovechan de las características de los nuevos sistemas operativos que son capaces de mapear el contenido de un fichero en un buffer de memoria interna, circunstancia que permite accesos aleatorios muy rápidos.

La transferencia de datos hacia y desde el buffer se realiza a través de *canales*. Un canal, o **Channel**, representa el origen y/o el destino de datos, por ejemplo **SocketChannel** y **FileChannel**. También están disponibles objetos de tipo **Pipe** que permiten mover datos entre diferentes partes de un mismo programa.

Una diferencia entre *streams* y *canales* es que estos últimos permiten realizar operaciones de lectura y escritura al mismo tiempo. Y también que los canales pueden ser no bloqueantes. En el caso de que los canales se abran sobre un fichero siempre son bloqueantes, pero en el caso de canales sobre sockets es posible seleccionar cualquiera de las dos características. Si se activa la transferencia no bloqueante, cuando se invoca al método de lectura es posible que se realice ésta cuando todavía no se han recibido todos los datos; y lo mismo sucede con la escritura, en cuyo caso puede que no se transfiera el contenido completo del buffer. Probablemente esto suene anárquico al lector, pero hay situaciones en las que resulta la única solución posible, por ejemplo en las transmisiones de vídeo, en donde se va escribiendo en los canales todo el contenido del buffer, pero no a expensas del procesamiento de ese vídeo.

En general, toda la funcionalidad ofrecida por los objetos de tipo **InputStream** y **OutputStream** se proporciona ahora a través de canales, de un modo más optimizado. No obstante, la clase **Channel** proporciona métodos estáticos para interactuar con el antiguo API de entrada/salida de Java, que no está descartado en absoluto; al contrario, hay algunas partes que han sido mejoradas e incluso reescritas. La más notable es la correspondiente a las operaciones de entrada/salida a través de *bloques*, que ahora funciona en un entorno de tareas mucho más eficiente y más sencillo de controlar.

Buffers

Los buffers corresponden fundamentalmente a una forma sencilla de representar un array de bytes. Para añadir datos a un buffer se utiliza el método *put()* del propio buffer, o el método *read()* de los canales. La lectura de datos se realiza a través del método *get()*, o llamando al método *write()* de los canales.

Todos los buffers tienen cuatro propiedades básicas, que se describen a continuación.

1. *Capacidad*. Ésta es una propiedad que no se puede modificar una vez que se haya creado el buffer. Indica el contenido máximo de datos asignado a ese buffer.
2. *Límite*. Es una marca de fin de buffer, que se puede desplazar dentro de un buffer con capacidad ya determinada para permitir cambiar dinámicamente el tamaño de la parte utilizable del buffer. Por defecto, corresponde al mismo valor que la capacidad. No se puede leer y escribir en el buffer más allá del límite y no se puede mover el límite más allá, obviamente, de la capacidad del buffer.
3. *Posición*. Corresponde a la localización dentro del buffer donde se realizará la siguiente acción relativa de lectura o escritura. Todas estas operaciones de lectura/escritura pueden realizarse de forma *absoluta*, especificando un desplazamiento desde el origen del buffer, o de forma *relativa*, referida a la posición actual. Funciona de forma semejante a un *puntero* en un archivo o a un *cursor* en una tabla de base de datos.
4. *Marca*. Es una posición etiquetada en el buffer. Siempre debe estar situada antes de la posición actual y puede utilizarse el método *reset()* para volver a ella. Por defecto, no hay ninguna marca definida, entendiéndose que siempre existe una marca implícita asignada a la posición cero.

Un problema siempre presente con los procesos de entrada/salida tradicionales en Java es el no poder interactuar con los flujos de datos binarios de fuentes que no sean Java. El mapeo de caracteres es adecuado para aplicaciones basadas en texto, como un servidor HTTP, pero cuando se trata de imágenes, sonido, vídeo o incluso datos numéricos, Java siempre ha tenido problemas en el control, fundamentalmente debido a la característica multiplataforma de Java. Un ejemplo claro es el orden de los bytes; por ejemplo, a la hora de manipular un dato de tipo short, formado por dos bytes, para

Java el primer byte es el más significativo, lo cual no es cierto en todas las plataformas; por lo tanto si se codifica la lectura de un fichero diseñado para una plataforma en la cual el segundo byte es el más significativo, el programador debería suplir la carencia de Java.

Ahora **NIO** ofrece una solución, ya que permite indicar el orden de los bytes mediante la clase **ByteOrder**, que define las constantes **BIG_ENDIAN** y **LITTLE_ENDIAN**, de forma que se consiga un mapeado correcto en Java cuando se escriben o extraen datos de cualquier archivo o flujo de datos.

El uso de buffers tiene especial importancia cuando se trata de analizar archivos XML utilizando analizadores de tipo DOM, permitiendo mapear grandes cantidades de datos del archivo en buffers de memoria y luego en objetos **DOM**, cuando se requiera, en lugar de hacerlo en un único paso.

Canales

La interfaz **Channel** es muy simple, solamente dispone de los métodos *isOpen()* y *close()*. Un canal se abre en el momento que se crea y nunca puede ser vuelto a abrir una vez que se haya cerrado. Para leer y escribir datos en un canal se utilizan los métodos *read()*, definido en la interfaz **ReadableByteChannel**, y *write()*, definido en la interfaz **WritableByteChannel**.

La interfaz de conveniencia **ByteChannel** consolida las dos anteriores, y es la que implementan los dos canales más importantes: **SocketChannel** y **FileChannel**.

Un canal de tipo **FileChannel** es un canal de lectura o escritura dependiendo de cómo se abra. Será un canal de escritura si se ha creado desde un objeto **FileOutputStream** o un objeto **RandomAccessFile** abierto como lectura/escritura, y será un canal de lectura si se ha creado a partir de un **FileInputStream** o desde un **RandomAccessFile**.

El tamaño del fichero está disponible desde el canal a través del método *size()*. El tamaño puede incrementarse si se escribe más allá del tamaño actual del fichero y se puede reducir llamando al método *truncate()*. El canal también dispone de un puntero cuya posición se puede conocer y fijar mediante el método *position()*. La posición en donde se encuentre el puntero será la siguiente en leer o escribir, según la acción que se realice.

El ejemplo Java908.java es muy sencillo y solamente trata de copiar un fichero, para lo cual se deben indicar tanto el archivo de origen como el archivo de destino.

La línea más interesante del programa es la llamada al método *transferFrom()*, que ejecuta una transferencia optimizada de bytes entre los dos canales que se han abierto hacia el archivo de entrada, lectura, y hacia el archivo de salida, escritura.


```
// Éste es el método que realiza la copia del contenido  
// del archivo origen en el archivo destino  
salida.transferFrom( entrada,0L,(int)entrada.size() );
```

Otras partes que incorpora el API NIO, además de buffers y canales, corresponden a **Charsets**, que permiten el tratamiento de flujos de caracteres, disponiendo de codificadores y decodificadores asociados que son capaces realizar la traslación entre bytes y caracteres Unicode. Además, NIO también incorpora **Selectors**, que si se asocian a los canales permiten la implementación de un sistema de entrada/salida multiplexado y no bloqueante. Por último, NIO también proporciona soporte para expresiones regulares que permiten trabajar con patrones de reconocimiento, búsqueda, sustitución, etc.

En este último caso, los desarrolladores pueden utilizar la posibilidad de tratamiento de expresiones regulares, por ejemplo, para imprimir todas las líneas de un fichero de texto que contengan una cadena patrón que describe a su vez un conjunto de cadenas. El ejemplo `Java909.java` realiza esta tarea, leyendo cada línea del fichero de texto que se pasa como argumento en la línea de comandos e imprime aquellas que coinciden con la expresión que también debe indicarse en la línea de comandos. Por ejemplo, para imprimir todas las líneas que contengan la palabra “Buscar” en el fichero `Java909.java`, se ejecutaría el comando:

```
% java Java909 Buscar Java909.java
```

Y para imprimir las líneas que contengan “Buscar” y también “buscar”, es decir, independientemente de si la primera letra de la palabra es mayúscula o minúscula, el comando de invocación sería:

```
% java Java909 "Buscar|buscar" Java909.java
```

Las partes más interesantes del código del ejemplo son aquellas en las que NIO utiliza las capacidades de reconocimiento de patrones, en base al tratamiento de expresiones regulares. Para realizar esta tarea, la clase utiliza varias sentencias.

```
Pattern patron = Pattern.compile( args[0] );
```

Esta sentencia compila la expresión regular que se pasa desde la línea de comandos, en un patrón más eficiente para realizar la búsqueda, en base al reconocimiento de patrones, en el fichero de texto. Devuelve una referencia al patrón ya compilado.

```
Matcher m = patron.matcher( "" );
```

Aquí se obtiene un objeto de tipo **Matcher** que se encargará de realizar las acciones de comprobación del texto. Se le pasa el argumento vacío ("") al método del objeto **Pattern**, para indicar que el patrón real de búsqueda se proporcionará en una sentencia posterior.

A continuación, el bucle `while` se encarga de recorrer todas las líneas del fichero.

```
while( (linea=br.readLine()) != null ) {  
    // Reseteamos el objeto Matcher para que comience la búsqueda  
    // al principio de la línea  
    m.reset( linea );  
    // Buscamos el patrón. Obtenemos true si se localiza  
    if( m.find() ) {  
        // Imprimimos la línea de texto que contiene el patrón  
        System.out.println( linea );  
    }  
}
```

Dentro del bucle se realizan dos acciones. La primera invoca al método *reset()* del objeto **Matcher**. Al contrario de lo que ocurre con los objetos de tipo **Pattern**, los objetos **Matcher** mantienen información de estado, incluyendo la línea de texto en que buscar y la información de dónde comienza la cadena que está buscando. El método *reset()* salva la línea de texto y asegura que el objeto **Matcher** siempre comience a buscar en la posición inicial.

La otra sentencia del bucle es la invocación del método *find()* del objeto **Matcher**, que es el que realmente realiza la búsqueda de la cadena patrón en la línea salvada por *reset()*. Si la encuentra, devuelve true y se imprime en la salida estándar.