

Tema 1

Persistencia en ficheros

Acceso a Datos (Desarrollo de aplicaciones multiplataforma) Carlos Alberto Cortijo Bon



Esta obra está bajo una [Licencia Creative Commons Atribución-NoComercial-SinDerivadas 4.0 Internacional](https://creativecommons.org/licenses/by-nc-nd/4.0/)

Índice

1. Ficheros	1
1.1. Sistemas de ficheros	1
1.2. Tipos de fichero	3
1.3. Ficheros binarios y de texto	3
1.4. Codificaciones de texto	3
2. La clase File de Java	4
3. Acceso secuencial y aleatorio a ficheros	7
4. Streams binarios (InputStream y OutputStream)	7
5. Streams de texto (Reader y Writer)	13
6. Jerarquías de clases para streams	15
7. Lectura y escritura de texto línea a línea (BufferedReader y BufferedWriter)	18
8. Lectura y escritura de texto con codificaciones de texto determinadas	19
9. Lectura y escritura de texto en streams binarios (InputStreamReader y OutputStreamWriter)	20
10. Lectura y escritura de datos elementales (DataInputStream y DataOutputStream)	22
11. Lectura y escritura de objetos (ObjectInputStream y ObjectOutputStream)	24
12. Acceso aleatorio a ficheros	27

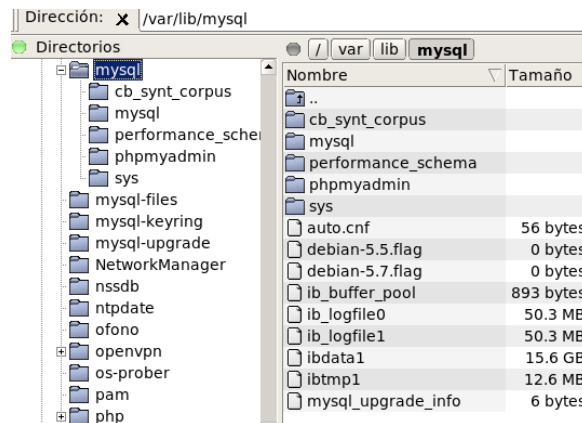
1. Ficheros

Un **fichero** es una secuencia de bytes almacenada en un medio de almacenamiento, con lo que en principio puede almacenar cualquier tipo de información.

1001010000 1001010011 1010010101 0001011100 0010010110

Los ficheros son el medio de almacenamiento de información más elemental. Pero en última instancia, por sofisticados que sean, todos los sistemas de almacenamiento de datos se basan en ficheros en los que se almacenan los datos.

@PEND: almacenamiento de datos de un SGBD MySQL. Ahí hay un fichero de 15.6GBytes de tamaño, que contiene los datos. Este fichero tiene una compleja organización interna. El sistema gestor de bases de datos es responsable de llevar a cabo las operaciones de consulta y modificación que múltiples aplicaciones clientes agrupan en transacciones. Para ello son necesarios sofisticados mecanismos de sincronización, de manera que se puedan realizar tantas operaciones simultáneas como sea posible, pero manteniendo la consistencia e integridad de los datos. En cualquier caso, en última instancia, los datos se almacenan en ficheros.



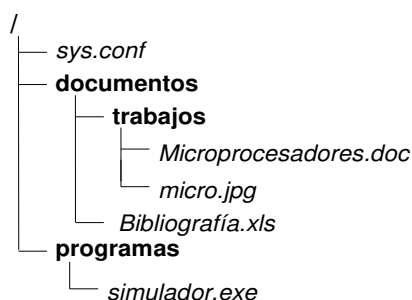
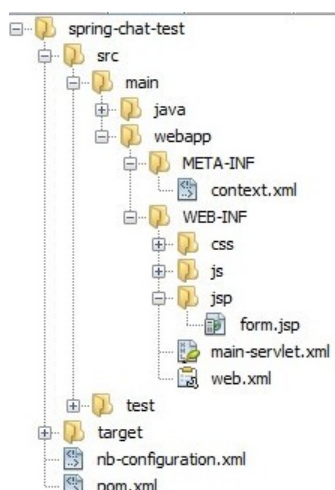
@PEND. Esto ilustra el inconveniente fundamental de los ficheros. Son un medio de almacenamiento muy elemental y proporcionan solo operaciones de muy bajo nivel. Son una solución viable para aplicaciones sencillas. Pero plantean un importante problema de escalabilidad. Es decir, no son viables en entornos en los que hay que dar servicio a múltiples procesos que realizan simultáneamente operaciones de lectura y modificación, y en los que hay que garantizar la consistencia e integridad de los datos.

1.1. Sistemas de ficheros

En un medio de almacenamiento de datos, como por ejemplo en un disco duro, se puede crear un sistema de ficheros. Los ficheros se almacenan en una estructura jerárquica dentro de un sistema de ficheros.

- Un sistema de ficheros tiene un directorio raíz.
- Un directorio puede contener tanto ficheros como directorios. Estos últimos, a su vez, pueden contener tanto ficheros como directorios.

@PEND: utilizar el siguiente ejemplo, pero completar con directorios anteriores hasta llegar al directorio raíz /.



Se puede designar de manera inequívoca un directorio o fichero mediante su ruta, *path*, o nombre completo (o absoluto). Este se forma concatenando los nombres de directorios desde el directorio raíz hasta llegar a él, separados por /. El nombre completo de un fichero se forma concatenando el nombre completo del directorio donde está situado y el nombre del fichero, separados por /.

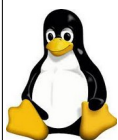
Tanto ficheros como directorios tienen:

- Un nombre.
- Una ubicación dentro del sistema de ficheros.



Los ficheros tienen, además, un contenido que, como se ha dicho, consiste en una secuencia de bytes.

En la tabla siguiente se muestran los nombres completos de todos los directorios y ficheros de la jerarquía anterior.

Directorios	Ficheros
/	/sys.conf
/documentos	/documentos/trabajos/Microprocesadores.doc
/documentos/trabajos	/documentos/trabajos/micro.jpg
/programas	/documentos/Bibliografía.xls
	/programas/simulador_micro.exe



En sistemas operativos **Linux**, el sistema operativo maneja una única jerarquía de directorios en la que se enganchan los sistemas de ficheros de los distintos dispositivos de almacenamiento en directorios especiales llamados puntos de anclaje. El propio directorio raíz / es un punto de anclaje para el directorio raíz de un sistema de ficheros contenido en un dispositivo de almacenamiento determinado, en el que está instalado el sistema operativo. Cuando se introduce un dispositivo extraíble, se crea un punto de anclaje en un directorio especial, que suele ser /media.

	En sistemas operativos Windows, se asigna una letra de dispositivo a cada sistema de ficheros. La letra C suele estar asociada al sistema de ficheros donde está instalado Windows.
	El directorio raíz dentro de la unidad C se denomina C : \
	En las rutas o nombres completos de ficheros y directorios se utiliza \ en lugar de /. El sistema operativo suele estar instalado en el directorio C:\Windows.
	Si se introduce un dispositivo extraíble, como por ejemplo una memoria flash usb, se le asigna automáticamente una letra de unidad.

1.2. Tipos de fichero

@PEND: Se suele indicar el tipo de fichero en el propio nombre, después de un punto y al final del todo.

Existen infinidad de tipos de ficheros. Cualquier usuario habitual de ordenadores está familiarizado con muchos de ellos.

txt

doc, docx, xls

odt, ods

png, gif

pdf

@PEND. Enlace a sitio web en el que se puedan consultar extensiones de fichero más habituales.

1.3. Ficheros binarios y de texto

Los ficheros pueden contener información de cualquier tipo, pero a grandes rasgos cabe distinguir entre ficheros de texto y ficheros binarios.

- Ficheros de texto: Contienen única y exclusivamente una secuencia de caracteres. Estos pueden ser caracteres visibles tales como letras, números, signos de puntuación, etc., y también espacios y separadores tales como tabuladores y saltos de línea. Su contenido se puede visualizar y modificar con programas editores de texto.
- Ficheros binarios: Son el resto de ficheros. Pueden contener cualquier tipo de información. En general hacen falta programas especiales para mostrar la información que contienen.

Hay que dejar bien claro que el que un fichero se considere binario o de texto no depende del fichero en sí, sino de la interpretación que se haga de su contenido. Cada tipo de fichero se puede considerar o bien de texto o bien binario. Los ficheros con extensión html, por ejemplo, suelen contener código en lenguaje HTML y son, por tanto, ficheros de texto. Los ficheros con extensión java suelen contener código en lenguaje Java y son, por tanto, ficheros de texto. Los ficheros con extensión pdf suelen ser documentos en formato PDF y son ficheros binarios. Los ficheros con extensión txt suelen contener texto. Pero un mismo texto puede representarse con diferentes secuencias de bytes según la codificación de caracteres que se emplee, como se ve en el siguiente apartado.

1.4. Codificaciones de texto

Un texto es una secuencia de caracteres. Los caracteres, como cualquier tipo de datos, en última instancia, se representan con una secuencia de bytes.

Una **codificación de texto** (*text encoding*) es una correspondencia entre caracteres y secuencias de bytes, que hace corresponder a cada carácter una secuencia de bytes que lo representan. Y a la inversa, una codificación de texto permite obtener un carácter a partir de la secuencia de bytes que lo representa. En muchos casos es un único carácter, pero pueden ser varios.

Un texto se representa con la concatenación de las secuencias de bytes que representan cada uno de sus caracteres.



Algunas codificaciones, las primeras que surgieron, hacían corresponder un byte a cada carácter. Esto limitaba a 255 el número de caracteres representables. Esto permitía representar muy pocos caracteres.

La codificación **ASCII** original no permitía representar caracteres que no existían en la lengua inglesa, tales como vocales acentuadas, o letras existentes en otros idiomas, como ñ, ç, ß, ø, etc.

Después surgieron variantes de esta codificación que incluían caracteres de otras lenguas, como el español, francés, alemán, italiano, etc. Pero cada variante servía solo para una lengua o conjunto de lenguas.

Más tarde surgieron otras codificaciones que utilizaban dos o más bytes para representar algunos caracteres. Ello permitió representar caracteres de lenguas con alfabetos complejos como las asiáticas (chino simplificado y tradicional, japonés, coreano, árabe, etc). Pero seguían siendo distintas codificaciones para distintas lenguas. No había una codificación universal.

Finalmente, se introdujo el estándar **Unicode** para representar todos los caracteres existentes en cualquier lengua habido y por haber, y con cualquier alfabeto o conjunto de caracteres (o dicho más técnicamente, sistema de escritura o *scripts*). Y no solo caracteres de texto, sino de todo tipo, por ejemplo *emojis*. Se puede consultar todo esto en <https://www.unicode.org/charts>. Y buscar caracteres de Unicode en <https://www.fileformat.info/info/unicode/char/search.htm>.

Unicode no es una codificación de caracteres. El estándar Unicode incluye varias codificaciones de caracteres, entre ellas:

- UTF-8. Es con mucho la más importante hoy día. Tiene la enorme ventaja de que es compatible con ASCII. Quiere esto decir que cualquier carácter que está en el código ASCII se representa igual en UTF-8, por lo que cualquier texto válido en ASCII es un texto válido en UTF-8. Utiliza de 1 a 4 bytes para representar un carácter, dependiendo del carácter.

Esta es la codificación que se suele utilizar para ficheros de texto.

- UTF-16. Utiliza 2 ó 4 bytes para representar un carácter, dependiendo del carácter.

Es la utilizada internamente por defecto por la máquina virtual de Java para almacenar en memoria las cadenas de caracteres.

2. La clase File de Java

La clase `File` de Java permite obtener información y realizar operaciones sobre ficheros y directorios como un todo. Es decir: crear, borrar y renombrar directorios y ficheros, además de obtener información acerca de ellos.

No permite leer la información contenida en el fichero, ni tampoco modificarla. Para eso se pueden utilizar las clases que se verán a continuación: clases *stream* para acceso secuencial y `RandomAccessFile` para acceso aleatorio.

Hay que recalcar, antes que nada, que crear un objeto de la clase `File` para un nombre de fichero determinado no tiene nada que ver con que exista un fichero con ese nombre o no, y no hace que se cree el fichero.

ro. Pero si ya existe un fichero o directorio con ese nombre, los métodos de la clase `File` permitirán obtener información de él, y también borrarlo. Y si no existe, se podrá crear utilizando métodos de esta clase.

Cuadro 1.1: Métodos de la clase `File`

Constructor	<code>File (String pathname)</code>	Crea un objeto <code>File</code> para la ruta indicada, que puede corresponder a un directorio o a un fichero.
Consulta de propiedades	<code>boolean exists()</code>	Devuelve <code>true</code> si existe el fichero o directorio. El caso de uso más frecuente es que se crea el objeto de clase <code>File</code> especificando un nombre con el constructor, y se ejecuta este método para verificar si existe un fichero o directorio con ese nombre.
	<code>boolean isDirectory()</code> <code>boolean isFile()</code>	Devuelven <code>true</code> , respectivamente, si se trata de un fichero o un directorio.
	<code>boolean canRead()</code> <code>boolean canWrite()</code> <code>boolean canExecute()</code>	Devuelven <code>true</code> si el programa que se está ejecutando tiene permiso, respectivamente, de lectura, escritura o ejecución para el fichero. Para un directorio, el permiso de ejecución significa que el directorio se puede establecer como directorio actual.
	<code>long length()</code>	Devuelve la longitud del fichero. El valor que se devuelve es indeterminado si se trata de un directorio.
	<code>String getParent()</code> <code>File getParentFile()</code>	Devuelven el directorio padre. El primer método devuelve su nombre. El segundo un objeto <code>File</code> que lo representa.
	<code>String getName()</code>	Devuelve el nombre del fichero o el directorio.
	<code>String getAbsolutePath()</code>	Devuelve la ruta absoluta del fichero.
Enumeración	<code>String[] list()</code> <code>File[] listFiles()</code>	El primero método devuelve un <i>array</i> con los nombres de los ficheros y directorios dentro del fichero actual. El segundo devuelve un <i>array</i> con objetos de tipo <code>File</code> que los representan.
Creación, borrado y renombrado	<code>boolean createNewFile()</code>	Crea un nuevo fichero, vacío, si no existe ya uno con el nombre.
	<code>static File createTempFile(String prefix, String suffix)</code> <code>static File createTempFile(String prefix, String suffix, File directory)</code>	Crea un nuevo fichero temporal. Se puede especificar un prefijo y un sufijo para su nombre. También un directorio para crear el fichero temporal.
	<code>boolean delete(File dest)</code>	Borra el fichero o directorio.
	<code>boolean renameTo()</code>	Renombra el fichero.
	<code>boolean mkdir()</code>	Crea el directorio.
Otras	<code>java.nio.file.Path toPath()</code>	Devuelve un objeto de la clase <code>java.nio.file.Path</code> correspondiente con el fichero, que permite acceder a información y funcionalidad adicionales.

El separador utilizado para las distintas partes de una ruta es distinto según el sistema operativo. Para Linux, y en general para todos los sistemas operativos excepto Windows, es `" / "`, mientras que para Windows es `" \ "`. Por suerte, este valor está en un atributo estático de la clase `File`. Utilizando este atributo, se puede escribir código genérico que funcionará en cualquier sistema operativo.

<code>static final String separator</code> <code>static final char separatorChar</code>	El carácter separador de elementos de una ruta, cuyo valor puede depender del sistema operativo. Está en dos atributos, uno de tipo <code>char</code> y otro de tipo <code>String</code> .
--	--

El siguiente programa de ejemplo muestra información acerca del directorio o fichero cuyo nombre se proporciona por parámetro de línea de comandos. Si no se especifica ninguno, se muestran los contenidos del directorio actual ("."). Si se introduce el nombre de un fichero, se muestra un mensaje que dice que es un fichero. Si se introduce en nombre de un directorio, se muestran sus contenidos.

```
import java.io.File;

public class MuestraFichODir {

    public static void main(String[] args) {

        String ruta = ".";
        if (args.length >= 1) {
            ruta = args[0];
        }

        File fich = new File(ruta);
        if (!fich.exists()) {
            System.out.println("No existe el fichero o directorio (" + ruta + ").");
            return;
        }
        if (fich.isFile()) {
            System.out.printf("%s es un fichero.\n", ruta);
        } else if (fich.isDirectory()) {
            System.out.printf("%s es un directorio. Contenidos: \n", ruta);
            File[] ficheros = fich.listFiles(); // Ficheros o directorios
            for (File f : ficheros) {
                System.out.print(f.getName());
                if (f.isDirectory()) {
                    System.out.print("/");
                }
                System.out.println();
            }
        }
    }
}
```

Actividad 1.1

Crea un fichero temporal utilizando el método `createTempFile`, sin especificar el directorio donde se crea. Escribe la ruta absoluta o completa para el fichero, una vez creado. Verifica que el fichero está en ese directorio.

Actividad 1.2

Crea un directorio, y dentro de él dos ficheros y un directorio. Para cada uno los ficheros y directorios que has creado, muestra su nombre, una vez obtenido mediante un método de la clase `File`. Para el primer directorio que has creado, muestra una lista con todos sus contenidos, mostrando para cada fichero y directorio su nombre y si es un fichero o un directorio.

Verifica que todo se ha creado correctamente.

3. Acceso secuencial y aleatorio a ficheros

La forma más sencilla de acceso a un fichero es secuencial. Con esta forma de acceso, se accede a los contenidos del fichero byte a byte, empezando con el primero y terminando con el último.

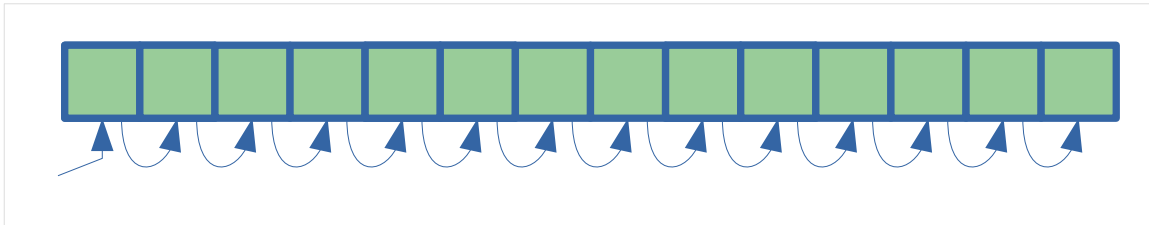


Ilustración 1.1: Acceso secuencial a ficheros

Para ello se utilizan *streams* (o flujos) en Java. Los *streams* son una abstracción que se utiliza no solo para ficheros, sino para cualquier fuente de datos de las que se leen los datos secuencialmente y byte a byte, o bien en el que se escriben los datos también secuencialmente, byte a byte. Por ejemplo, se puede utilizar un *stream* para la lectura de datos que se reciben por una conexión de red del protocolo de transporte TCP. En ella, cada uno de los dos procesos entre los que se ha establecido una conexión de TCP pueden enviar datos byte a byte al otro proceso, que los lee byte a byte.

Los *streams* se implementan en Java en varias jerarquías de clases, cuyas clases raíces son las siguientes, y que se verán con detalle en apartados siguientes.

	Lectura	Escritura
Flujo binario	<code>InputStream</code>	<code>OutputStream</code>
Flujo de texto	<code>Reader</code>	<code>Writer</code>

No es esta la única manera de acceder a los contenidos de un fichero. También es posible el acceso directo a una posición determinada del fichero, especificada por un desplazamiento en bytes desde el inicio del fichero, o bien desde la última posición a la que se accedió. Esta forma de acceso se llama acceso aleatorio, y está implementada en Java en la clase `RandomAccessFile`.

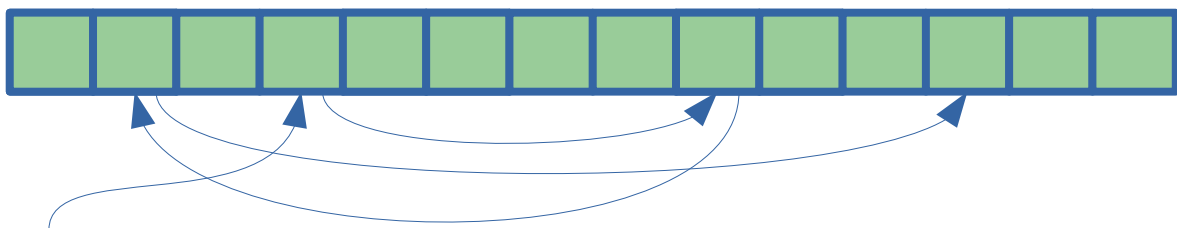
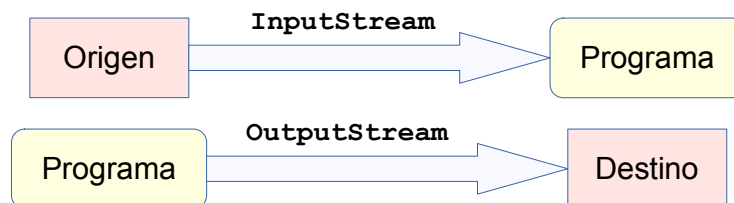


Ilustración 1.2: Acceso aleatorio a un fichero

4. Streams binarios (`InputStream` y `OutputStream`)

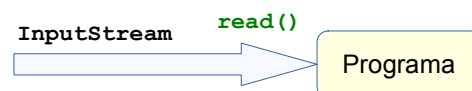
Un *stream* es un flujo de datos secuencial. Con respecto al programa que lo usa, un *stream* puede ser de entrada o de salida. La clase de Java `InputStream` permite leer bytes de un flujo de entrada. La clase de Java `OutputStream` permite escribir bytes a un flujo de salida.



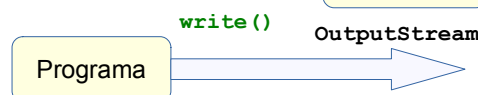
Por un *stream* de salida se pueden enviar datos de cualquier tipo y, en correspondencia, por un *stream* de entrada se pueden recibir datos de cualquier tipo. Pero en última instancia, la información que se transmite son bytes. Para enviar un dato por un *stream*, hay que convertirlo a una secuencia de bytes. Cuando se recibe un dato por un *stream*, hay que interpretar la secuencia de bytes que se recibe, para generar el dato correspondiente. El lenguaje de programación Java proporciona mecanismos que hacen esto sencillo, y que se verán más adelante. Por ahora, se hablará solo de *streams* de bytes.

Los bytes que se envían a un *stream* de bytes por un extremo se reciben por el otro en el mismo orden en el que se han enviado.

- Un `InputStream` representa un *stream* de entrada de bytes y tiene una operación `read()` para recibir un byte.

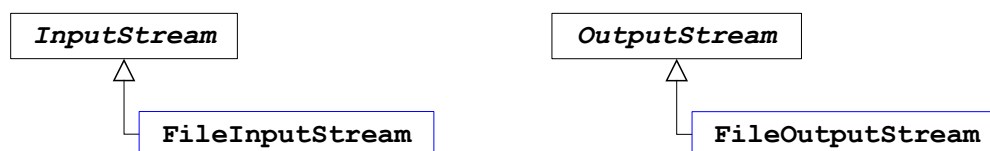


- Un `OutputStream` representa un *stream* de salida de bytes y tiene una operación `write()` para enviar un byte.



El origen y el destino de los bytes pueden ser muchas cosas distintas. Por ejemplo: un fichero, una zona de la memoria, o una conexión de red. Las clases `InputStream` y `OutputStream` son clases abstractas. Existen clases hijas suyas que implementan los métodos `read` y `write` para tipos particulares de orígenes y de destinos de datos.

Por ejemplo: las clases `FileInputStream` y `FileOutputStream` permiten, respectivamente, escribir bytes a un fichero y leer bytes de un fichero. Permiten el acceso secuencial a ficheros, porque con ellos se puede escribir una secuencia de bytes en un fichero, o recibir una secuencia de bytes con los contenidos de un fichero.



El siguiente programa de ejemplo obtiene los bytes que conforman un `String` y los escribe, uno a uno, en un fichero. Cada carácter se escribe con una llamada al método `write`.

```

import java.io.FileOutputStream;
import java.io.IOException;

public class EscribeBytesAFichero {

    private static final String NOM_FICH_SALIDA = "fichero.txt";

    public static void main(String[] args) {

        String cadena = "Hola, soy una secuencia de bytes.";
        byte[] bytes = cadena.getBytes();

        try (FileOutputStream fos = new FileOutputStream(NOM_FICH_SALIDA)) {

```

```

        for(byte unByte: bytes) {
            fos.write(unByte);
        }
    } catch (IOException ex) {
        System.out.printf("ERROR: escribiendo a fichero: %s\n", ex.getMessage());
    }
}
}

```

Bloques try con recursos

Para utilizar un *stream*, lo más conveniente es utilizar un bloque `try` con recursos. Con ello, se asegura que el stream se cierra correctamente aunque se produzca una excepción. A continuación se muestra el código equivalente a un bloque `try` con recursos de ejemplo.

<pre> try (var fos = new FileOutputStream(NOM_FICH)) { // (...) } catch (IOException ex) { // (...) } </pre>	<pre> FileOutputStream fos = null; try { fos = new FileOutputStream(NOM_FICH) // (...) } catch (IOException ex) { // (...) } finally { if(fos != null) { fos.close(); } } </pre>
--	--

En el bloque de inicialización de recursos que viene entre paréntesis después de `try`, se pueden crear objetos de cualquier clase que implemente una de las interfaces `Closeable` o `AutoCloseable`. Estas incluyen un método `void close()` para cerrar el *stream*.

El bloque `finally` se ejecuta siempre, incluso aunque se se produzca una excepción, de manera que el *stream* siempre se cierra.

Es muy importante cerrar los *streams* cuando se ha terminado de utilizarlos. Dejar ficheros abiertos cuando se han terminado de utilizar, por ejemplo, puede causar problemas. Lo mismo que dejar conexiones abiertas con un sistema gestor de bases de datos, o conexiones del protocolo de red TCP. Todos estos tipos de recursos se gestionan mediante clases que tienen un método `close` que debe ejecutarse una vez se ha terminado de utilizarlos.

El siguiente programa de ejemplo lee los bytes de un fichero con el mismo nombre que el que genera el programa anterior y escribe su valor numérico en salida estándar. También escribe el carácter equivalente. Esto lo hace pasando el mismo valor, que es un byte, y que por lo tanto puede tomar valores entre 0 y 255, para dos marcadores o *placeholders* distintos la cadena de formato de `printf`, uno con `%d` y otro con `%c`.

Cada byte (8 bits) se lee con una llamada al método `read`, que devuelve un `int` (32 bits). Si se ha podido leer un byte del fichero, `read` devuelve el valor del byte. Si no se ha podido leer porque se ha llegado al final del fichero, devuelve un valor negativo, -1.

```

import java.io.FileInputStream;
import java.io.IOException;

```

```
public class LeeBytesDeFichero {  
  
    private static final String NOM_FICH_ENTRADA = "fichero.txt";  
  
    public static void main(String[] args) {  
  
        try ( FileInputStream fis = new FileInputStream(NOM_FICH_ENTRADA) ) {  
            int unByte;  
            while ( (unByte = fis.read()) != -1 ) {  
                System.out.printf("%3d(%c)\n", unByte, (char) unByte);  
            }  
        } catch (IOException ex) {  
            System.out.printf("ERROR: leyendo de fichero: %s\n", ex.getMessage());  
        }  
    }  
}
```

El anterior programa escribe correctamente los caracteres leídos siempre que se almacenen en el fichero un único byte, lo que no siempre tiene por qué ser así. No lo es, en general, con vocales acentuadas o con diéresis o con caracteres que no existen en lengua inglesa, como por ejemplo: ñ, €, ç, ~, ß o æ, por ejemplo.

Para realizar las siguientes actividades puede ser necesario consultar la documentación de Java (los Java-docs) de las clases `InputStream` y `OutputStream`.

Actividad 1.3

Crea y ejecuta el primer programa. Después crea y ejecuta el segundo programa. Para que este último funcione, copia antes en su directorio de trabajo el fichero generado por el primer programa.

Prueba introduciendo en el texto del primer programa vocales acentuadas y caracteres como ñ, ç, ÿ, y otros que no existan en inglés. Verifica si el texto se ha escrito correctamente en el fichero y cómo el segundo programa muestra los caracteres.

Actividad 1.4

Crea un programa al que se le pase por parámetro de línea de comandos un nombre de fichero. Este puede tener un *path* absoluto, lo que es útil para acceder a él si no está en el mismo directorio en que se ejecuta el programa. Por ejemplo: `/etc/fstab`

Si el fichero no existe, se mostrará un mensaje de error y se terminará la ejecución del programa.

En otro caso, se creará una copia del fichero. El nombre de la copia será igual que el del fichero original, añadiendo `.bak` al final. Esta copia se creará en el mismo directorio en el que se ejecuta el programa. No hay que hacer nada especial para ello. Solo asegurarse de que se toma solo el nombre del fichero, y la ruta completa, para añadirle al final `.bak`. Por ejemplo, si el fichero original que hay que copiar es `/etc/fstab`, entonces el nombre de la copia será `fstab.bak`.

Para crear la copia, el programa debe crear un `FileInputStream` para leer del fichero de origen, byte a byte, y un `FileOutputStream` para escribir en el fichero de destino (la copia), cada byte que se va leyendo del fichero de origen. La estructura del programa será la siguiente.

```

public static void main(String[] args) {
    // (completar aquí ...)
    // (Después de hacer lo anterior, nomFich debe contener el nombre del fichero
    // que se va a copiar, y nomFichCopia el nombre del fichero en el que se copia)

    try ( InputStream is = new FileInputStream(nomFich);
          FileOutputStream fos = new FileOutputStream(nomFichCopia)) {
        // (completar aquí ...)
    } catch (IOException ex) {
        System.out.printf("Error de E/S obteniendo contenidos de URL.\n");
        ex.printStackTrace();
    }
}

```

Actividad 1.5

Las clases `InputStream` y `OutputStream` tienen métodos que permiten leer y escribir, respectivamente, los datos de un *array* de bytes (**byte**[]). Cambia el primer programa de ejemplo para que escriba directamente el *array* de bytes en el fichero.

Cambia el segundo programa para que lea el fichero generado por el anterior programa utilizando un *array* de bytes con longitud 5. La longitud del *array* se debe definir como una constante de clase (`final static`). El programa leerá repetidas veces del fichero hacia el *array*. En la última lectura, cuando se llegue al final del fichero, leerá un número de bytes inferior a la longitud del *array*, y así sabrá que ha llegado al final del fichero. La salida del programa será igual que la del programa anterior a partir del cual se ha creado.

Actividad 1.6

Crea un programa que realice un volcado (*dump*) hexadecimal del contenido de un fichero. El fichero se debe proporcionar como un argumento de línea de comandos. La salida puede ser similar a la del comando `hexdump -C` de Linux. Se muestra un ejemplo de su uso a continuación. No es necesario que el formato de la salida sea exactamente el mismo, pero sí se debería mostrar la misma información.

```

$ hexdump -C quijote.txt
00000000 45 6e 20 75 6e 20 6c 75 67 61 72 20 64 65 20 6c |En un lugar de l|
00000010 61 20 4d 61 6e 63 68 61 2c 20 64 65 20 63 75 79 |a Mancha, de cuy|
00000020 6f 20 6e 6f 6d 62 72 65 20 6e 6f 20 71 75 69 65 |o nombre no quie|
00000030 72 6f 20 61 63 6f 72 64 61 72 6d 65 2c 20 6e 6f |ro acordarme, no|
00000040 20 68 61 20 6d 75 63 68 6f 20 74 69 65 6d 70 6f | ha mucho tiempo|
00000050 20 71 75 65 20 76 69 76 c3 ad 61 20 75 6e 20 68 | que viv..a un h|
00000060 69 64 61 6c 67 6f 20 64 65 20 6c 6f 73 20 64 65 |idalgo de los de|
00000070 20 6c 61 6e 7a 61 20 65 6e 20 61 73 74 69 6c 6c | lanza en astill|
00000080 65 72 6f 2c 20 61 64 61 72 67 61 20 61 6e 74 69 |ero, adarga anti|
00000090 67 75 61 2c 20 72 6f 63 c3 ad 6e 20 66 6c 61 63 |gua, roc..n flac|
000000a0 6f 20 79 20 67 61 6c 67 6f 20 63 6f 72 72 65 64 |o y galgo corred|
000000b0 6f 72 2e 20 55 6e 61 20 6f 6c 6c 61 20 64 65 20 |or. Una olla de |

```

A la izquierda se muestra:

- La posición del primer carácter que se muestra en la línea. Este es un número que se muestra en formato hexadecimal. En cada línea se muestran 16 bytes. Por ello, las posiciones que se muestran son 0, 10, 20, etc.
- El valor de 16 bytes. Este valor se muestra en formato hexadecimal. De nuevo, no es estrictamente necesario mostrar este valor en hexadecimal, se puede mostrar en decimal.
- El valor de estos 16 bytes mostrados como texto ASCII. En este caso, la información que aparece es legible, porque se trata de un texto en castellano. Las letras que no existen en inglés, como es el caso de las vocales acentuadas, no se muestran correctamente, porque se representan con más de un byte.

No es estrictamente necesario mostrar los números en formato hexadecimal. Se pueden mostrar en formato decimal. De todas formas, no es difícil mostrarlos en formato hexadecimal, utilizando `printf` con las opciones de formato

apropiadas.

A modo de resumen, y para concluir, se incluye una tabla con algunos los principales métodos de la clase `InputStream`. Si has hecho las actividades anteriores, probablemente ya conoces y has utilizado algunos de ellos. El propósito de esta tabla es dar una idea de todas las posibilidades que existen para leer bytes de un *stream*. Para cada programa en particular, habrá que utilizar el método más conveniente teniendo en cuenta la funcionalidad requerida y el rendimiento.

Esta tabla contiene, además, los métodos `skip` y `skipNBytes`, que se salta un número de bytes especificado para poder seguir leyendo después de ellos.

Cuadro 1.2: Métodos de la clase `InputStream`

Lectura	<code>abstract int read()</code>	Lee el siguiente byte. Si el valor devuelto es -1, eso significa que se ha llegado al fin del <i>stream</i> y que, por tanto, no existen más bytes para leer.
	<code>int read(byte[] b)</code>	Lee varios bytes y los almacena en el <i>array</i> de bytes <code>b</code> . Devuelve el número de bytes que se han podido leer y almacenar en <code>b</code> . Si la longitud de <code>b</code> es 0, el método devuelve 0. En otro caso, si no se puede leer ningún byte del <i>stream</i> porque se ha llegado a su final, devuelve -1. En otro caso, lee tantos bytes como sea posible leer del <i>stream</i> y almacenar en <code>b</code> , y devuelve ese número de bytes.
	<code>int read(byte[] b, int off, int len)</code>	Similar al método anterior. Pero indica el máximo número de bytes que hay que leer (<code>len</code>), e indica la posición del <i>array</i> a partir de la cual hay que copiar los bytes leídos. Este método lanza una excepción de tipo <code>IndexOutOfBoundsException</code> si no es posible almacenar en el todos los bytes leídos en el <i>array</i> a partir de esta posición. Este será el caso cuando <code>off</code> tiene valor negativo, cuando <code>len</code> tiene valor negativo, y cuando <code>len</code> tiene valor mayor que <code>b.length - off</code> .
	<code>byte[] readNBytes(int len)</code>	Lee como máximo el número de bytes especificado (<code>len</code>) y los almacena en un <i>array</i> que crea para ello y que devuelve una vez finalizada la lectura. Este <i>array</i> tendrá como máximo longitud igual al número de bytes que se ha solicitado leer (<code>len</code>), pero puede tener una longitud menor si se ha llegado al final del <i>stream</i> antes de leer ese número de bytes.
	<code>int readNBytes(byte[] b, int off, int len)</code>	Lee como máximo el número de bytes especificado (<code>len</code>) y los almacena en el <i>array</i> <code>b</code> a partir de la posición <code>off</code> . Devuelve el número de bytes leídos y almacenados en el <i>array</i> , que será como máximo <code>len</code> .
	<code>byte[] readAllBytes()</code>	Lee todos los bytes que quedan por leer en un <i>array</i> que crea para almacenarlos y que devuelve una vez que los ha leído. Hay que utilizar este método con precaución, solo si se tiene la seguridad de que el número de bytes que faltan por leer no es muy grande.
Saltar bytes	<code>long skip(long n)</code>	Salta un máximo de <code>n</code> bytes del <i>stream</i> . Si quedan menos de <code>n</code> bytes por leer en el <i>stream</i> , salta menos bytes, porque antes llegará al final. Devuelve el número de bytes que se han saltado.
	<code>void skipNBytes(long n)</code>	Salta <code>n</code> bytes del <i>stream</i> . Si quedan menos de <code>n</code> bytes para leer del <i>stream</i> , lanza una excepción de tipo <code>EOFException</code> .

Ten en cuenta

Un programa debe funcionar correctamente. Es decir, debe hacer lo que se espera que haga. Pero el rendimiento es un aspecto muy importante. Lo debe hacer de la manera más rápida posible y utilizando la mínima cantidad de recursos del sistema posible (por ejemplo, memoria).

Si para realizar alguna tarea se necesita leer de un `InputStream`, habrá que utilizar los métodos más apropiados para la tarea particular.

Por ejemplo, si se quieren leer los *bytes* de un fichero en bloques de 16 bytes, se pueden utilizar los dos métodos siguientes.

```
int read(byte[] b)
byte[] readNBytes(int len)
```

Para utilizar el primero hay que crear un *array* con una longitud de 16 bytes, por ejemplo con `byte[] b = new byte[16]`. Con el segundo, esto no es necesario, porque el propio método crea el *array*.

Pero si se quieren leer repetidamente bloques de 16 bytes, es mejor utilizar el primer método, porque solo hace falta crear un *array* al principio y se utilizará para todas las lecturas. Si se utiliza el segundo, se creará un *array* cada vez que se quiera leer un bloque de bytes. Por supuesto, esto no es un inconveniente si solo se quiere leer un único bloque de bytes. Entonces se puede utilizar el segundo método, que puede resultar más cómodo.

5. Streams de texto (Reader y Writer)

En Java, la clase `String` de Java se utiliza para representar cadenas de caracteres y utiliza la codificación UTF-16 de Unicode para almacenar internamente los bytes que las representan.

El tipo `char` se utiliza para representar un carácter con la codificación UTF-16 de Unicode.

Los *streams* de texto permiten leer y escribir texto.

Las clases `Reader` y `Writer` son clases abstractas que representan, respectivamente, un *stream* de entrada de texto y un *stream* de salida de texto. Tienen, respectivamente, métodos `read` y `write` que permiten leer un carácter (`char`) y escribir un carácter. Existen clases hijas suyas que implementan los métodos `read` y `write` para tipos particulares de orígenes y de destinos de datos.

Por ejemplo: las clases `FileReader` y `FileWriter` permiten, respectivamente, escribir caracteres a un fichero y leer caracteres de un fichero.



Por defecto, se escriben los caracteres en ficheros con la codificación de texto UTF-8. Y se leen asumiendo que el texto que contienen está codificado en UTF-8. En Java, el texto en memoria está codificado en UTF-16. Por tanto, la lectura y escritura de caracteres en ficheros implica normalmente un proceso de recodificación de texto. Esto se hace de manera transparente y el programador normalmente no tiene que preocuparse por ello.

El siguiente programa de ejemplo escribe uno a uno los caracteres de un texto en un fichero, utilizando la clase `FileWriter`.

```
import java.io.FileWriter;
import java.io.IOException;

public class EscribeCaracteresAFichero {

    private static final String NOM_FICH_SALIDA = "fichero.txt";

    public static void main(String[] args) {

        String cadena = "Hola, soy una espléndida y muy reseñable secuencia de bytes.";

        try ( FileWriter fw = new FileWriter(NOM_FICH_SALIDA) ) {
            for (int i = 0; i < cadena.length(); i++) {
                fw.write(cadena.charAt(i));
            }
        } catch (IOException ex) {
            System.out.printf("ERROR: escribiendo a fichero: %s\n", ex.getMessage());
        }
    }
}
```

Los caracteres se escriben uno a uno a un `FileWriter` con el método `write(char c)`. Si se produce una `IOException`, se captura y se gestiona. El *stream* siempre se cierra, pase lo que pase.

El siguiente programa lee el texto del fichero carácter a carácter muestra cada carácter leído. Los caracteres leídos se muestran correctamente aunque no existan en el idioma inglés.

```
import java.io.FileReader;
import java.io.IOException;

public class LeeCaracteresDeFichero {

    private static final String NOM_FICH_ENTRADA = "fichero.txt";

    public static void main(String[] args) {

        try ( FileReader fr = new FileReader(NOM_FICH_ENTRADA) ) {
            int unCar;
            while ((unCar = fr.read()) != -1) {
                System.out.printf("%c\n", (char) unCar);
            }
        } catch (IOException ex) {
            System.out.printf("ERROR: leyendo de fichero: %s\n", ex.getMessage());
        }
    }
}
```

Actividad 1.7

Creas y ejecutas el programa anterior que escribe el texto carácter a carácter en un fichero. Después creas y ejecutas el segundo programa. Para que este último funcione, copias antes en su directorio de trabajo el fichero generado por el primer programa. Verificas que el segundo programa escribe el texto correctamente, también las vocales acentuadas y

la letra ñ.

Actividad 1.8

Existe un constructor de `FileWriter` con un parámetro que permite añadir contenido al final de un fichero en lugar de sobrescribir sus contenidos. Modifica el programa anterior para que lo use. Ejecútalo varias veces y verifica que se añade texto al final cada vez que se ejecuta.

Actividad 1.9

La clase `FileWriter` tiene métodos que permiten escribir un `String` de una vez en un fichero. Cambia el primer programa de ejemplo para que escriba directamente el `String` en el fichero.

Cambia el segundo programa para que lea el fichero generado por el anterior programa utilizando un *array* de `char` con longitud 5. La longitud del *array* se debe definir como una constante de clase (`final static`). El programa leerá repetidas veces del fichero hacia el *array*. En la última lectura, cuando se llegue al final del fichero, leerá un número de caracteres inferior a la longitud del *array*, y así sabrá que ha llegado al final del fichero. La salida del programa será igual que la del programa anterior a partir del cual se ha creado.

Actividad 1.10

Averigua la codificación de texto con la que el programa creado para ambas actividades anteriores ha generado el fichero.

En Linux puedes utilizar el comando `file`. Tanto en Linux como en Windows se puede abrir el fichero con un editor de texto y seleccionar la opción “Guardar como”, que normalmente muestra la codificación utilizada para el fichero y permite seleccionar una distinta para guardarlo.

En la documentación de la clase `String` (<https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/lang/String.html>) se explica que la cadena de caracteres se almacena internamente codificada en UTF-16. ¿Se ha cambiado la codificación en el fichero con respecto a la que tenía el `String` en memoria? Si es así, ¿en qué momento crees que puede haberse realizado esta recodificación en cada programa?

6. Jerarquías de clases para *streams*

Una vez que ya se han visto las clases básicas que permiten la lectura y escritura en *streams* binarios y de texto, y antes de continuar, se dará una visión general de las clases para lectura y escritura en *streams*, explicando el planteamiento general, e introduciendo las clases que faltan por ver.

Existen cuatro jerarquías de clases para *streams*, que se muestran a continuación.

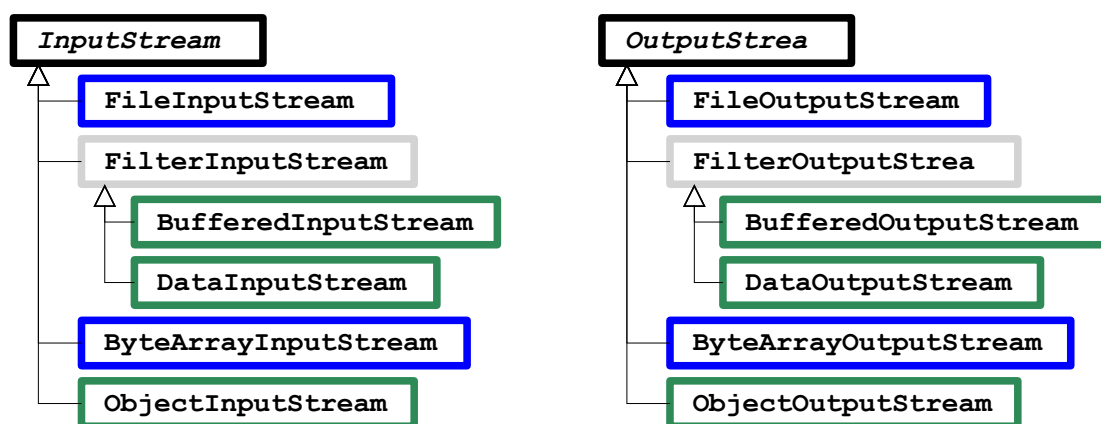


Ilustración 1.3: Streams binarios

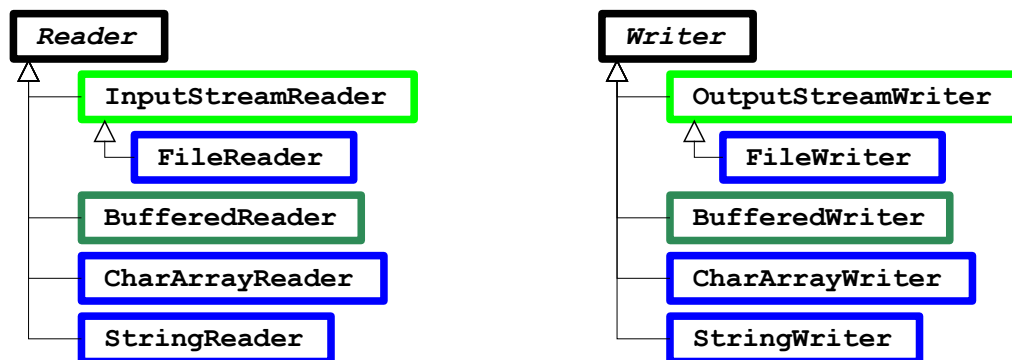


Ilustración 1.4: Streams de texto

El color del borde de las clases proporciona más información acerca de la funcionalidad proporcionada por las clases.

Clases raíces

Las clases en la raíz de cada una de estas cuatro jerarquías son clases abstractas. Su borde es de color negro, y tienen métodos abstractos con la funcionalidad básica, que deben ser implementados por clases hijas sobre distintos tipos de medios de almacenamiento.

- Para *streams* binarios existen clases para lectura y escritura de bytes (`InputStream` y `OutputStream`).
- Para *streams* de texto existen clases para lectura y escritura de caracteres (`Reader` y `Writer`).

Clases para medios de almacenamiento particulares

Existen clases que implementan las clases abstractas de la raíz de cada una de las jerarquías para medios de almacenamiento particulares. Su borde está en color azul marino. El nombre del medio se antepone al nombre de la clase en la raíz de la jerarquía.

- `File` indica que el medio de almacenamiento es un fichero.
- Existen varios prefijos cuando el medio de almacenamiento es la memoria.
 - Para *streams* binarios se tiene `ByteArray` para un *array* de bytes (`byte`) en memoria.
 - Para *streams* de texto se tiene:
 - `CharArray` para un *array* de caracteres (`char`) en memoria.
 - `String` para una cadena de caracteres (`String`) en memoria.

Clases para entrada y salida de entidades de más alto nivel

Existen clases que proporcionan funcionalidad para entrada y salida para entidades de más alto nivel que simples bytes o caracteres. Su borde está en color verde oscuro. Los prefijos para ellas son los siguientes.

- `Buffered` para clases que proporcionan *buffering*. Cabe distinguir entre *streams* de texto y binarios.
 - Para *streams* de texto, la clase `BufferedReader` permite la lectura de líneas completas de texto mediante el método `readLine`. La clase `BufferedWriter`, en correspondencia, tiene un método `newLine` que escribe un salto de línea.
 - Tanto para *streams* binarios como de texto, las clases `Buffered` implementan el *buffering*, que es un mecanismo que acelera las operaciones de lectura y escritura mediante el uso de un *buffer*. Un *buffer* es una zona de almacenamiento intermedio que permite reducir a un mínimo los accesos al medio de almacenamiento para las operaciones de lectura y escritura secuenciales que se llevan a cabo con los *streams*. Esto es muy importante cuando el medio de almacenamiento es un fichero, porque el acceso a un fichero es mucho más lento que a memoria. Con la primera operación de lectura, se lee no solo lo que se ha pedido, sino también en adelante y hasta llenar el *buffer* en memoria. De esa manera, lo que se quiera leer en sucesivas operaciones de lectura ya estará en el *buffer*, y no hará falta acceder de nuevo al medio de almacenamiento. Solo cuando se hayan consumido todos los contenidos del *buffer* se volverá a leer del medio de almacenamiento. Para escritura se actúa de manera análoga. No se escribe directamente en el medio de almacenamiento, sino el en *buffer*. Cuando se llena el *buffer*, se escriben sus contenidos en el medio de almacenamiento.
- `Data` para clases que permiten la lectura y escritura de datos de tipos elementales, a saber: `int`, `long`, `short`, `float`, `double`, `byte`, `char`, `boolean`.
- `Object` para clases que permiten la lectura y escritura de objetos. Es decir, de instancias de la clase `Object` o de cualquier subclase de `Object`.

Clases de enlace entre jerarquías de streams binarios y de texto

Estas clases son `InputStreamReader` y `OutputStreamWriter`.

Un `InputStreamReader` es un *stream* para lectura de texto (`Reader`) desde uno para lectura de bytes (`InputStream`).

Un `OutputStreamWriter` es un *stream* para escritura de texto (`Writer`) en uno para escritura de bytes (`OutputStream`).

Es decir, estas clases permiten leer o escribir texto sobre cualquier *stream* en el que se puedan leer o escribir bytes. Por ello, actúan como enlace entre ambas jerarquías.

La codificación de texto por defecto es UTF-8. Pero en sus constructores se puede especificar otra.

7. Lectura y escritura de texto línea a línea (`BufferedReader` y `BufferedWriter`)

Un `BufferedReader` se construye sobre un `Reader` y proporciona un método `readLine()` que permite leer texto línea a línea. Se puede ver su funcionamiento en el siguiente ejemplo, que escribe una a una, y numeradas, las líneas de un fichero de texto.

```
import java.io.FileReader;

import java.io.BufferedReader;
import java.io.IOException;

public class Prueba {

    public static void main(String[] args) {

        if (args.length < 1) {
            System.out.println("Indicar nombre de fichero.");
            return;
        }
        String nomFich = args[0];

        try ( FileReader fr = new FileReader(nomFich);
              BufferedReader fbr = new BufferedReader(fr) ) {
            int i = 0;
            String linea;
            while ((linea = fbr.readLine()) != null) {
                System.out.format("[%5d] %s", i++, linea);
                System.out.println();
            }
        } catch (IOException e) {
            System.out.println("Error de E/S: " + e.getMessage());
        }
    }
}
```

En el bloque de inicialización de recursos se construye primero un `Reader` y después, sobre este, un `BufferedReader`, que permite leer una línea entera de texto con `readLine()`.

Los recursos creados en el bloque de inicialización de recursos se cierran en orden inverso a aquel en que se han creado. Es decir, primero se cierra el `BufferedReader` con `br.close()` y después el `FileReader` con `fr.close()`.

Existe, en correspondencia, una clase `BufferedWriter`, que se construye sobre un `Writer` y proporciona un método `newLine()`, que escribe un salto de línea. El siguiente programa escribe un triángulo de caracteres en un fichero de texto.

```
import java.io.FileWriter;

import java.io.BufferedWriter;
import java.io.IOException;

public class Prueba {

    private static final String NOM_FICH_SALIDA = "fichero.txt";

    public static void main(String[] args) {

        try ( FileWriter fw = new FileWriter(NOM_FICH_SALIDA);
              BufferedWriter bw = new BufferedWriter(fw)) {
            for(int i=0; i<10; i++) {
                bw.write("*".repeat(i));
                bw.newLine();
            }
        } catch (IOException e) {
            System.out.println("Error de E/S: " + e.getMessage());
        }
    }
}
```

Actividad 1.11

En realidad, lo que hace `newLine` es, básicamente, escribir un salto de línea, `'\n'`. Prueba a modificar el programa anterior para que utilice solo la clase `FileWriter`.

8. Lectura y escritura de texto con codificaciones de texto determinadas

Cuando se lee texto desde ficheros o se escribe texto en ellos, se utiliza la codificación de texto (*charset*) por defecto de la máquina virtual de Java. Esta suele ser UTF-8. Pero, como se ha visto, la clase `String` utiliza internamente UTF-16 para almacenar texto en memoria.

Siempre que hay una transferencia de texto desde ficheros a memoria, o viceversa, se hace una recodificación de texto si la codificación utilizada es distinta para memoria y para ficheros, lo que suele ser el caso.

Existen constructores de las clases `FileReader` y `FileWriter` que permiten especificar una codificación de caracteres. Algunas de las opciones son las siguientes.

UTF-8	Codificación UTF-8 de Unicode.
UTF-16	Codificación UTF-16 de Unicode.
ISO-8859-1	Codificación ISO Latin Alphabet No. 1, también conocida como ISO-LATIN-1. No es una codificación universal. Es decir, solo se puede utilizar para un conjunto reducido de idiomas, entre ellos el español. Hoy en día está obsoleta, pero se pueden encontrar ficheros que la utilizan.
US-ASCII	Codificación ASCII de 7 bits. Solo tiene letras existentes en inglés. Podría ser interesante utilizarlo si se sabe que el texto no tiene otros caracteres o se quiere que se produzca una excepción en caso de que hubiera cualquier otro.

Actividad 1.12

Crea un programa que genere un fichero de texto con codificación de texto ISO-8859-1, y con varias líneas, en las que aparezcan vocales acentuadas, las letras ñ y ç, y el símbolo del euro (€). Verifica por medios externos al propio programa que el fichero se ha generado con esta codificación. Si entregas esta actividad, adjunta evidencia (captura de pantalla) de esta verificación.

Crea otro programa que lea línea a línea este fichero y que escriba en salida estándar cada una de las líneas leídas. Verifica que todos los caracteres se muestran correctamente.

9. Lectura y escritura de texto en *streams* binarios (InputStreamReader y OutputStreamWriter)

A veces se necesita leer texto de una fuente de datos y solo se dispone de un *stream* binario que permite leer bytes de ella.

De la misma forma que un `FileReader` (Reader para un fichero o *file*) permite leer texto de un fichero, que es una secuencia de *bytes*, un `InputStreamReader` (Reader para un `InputStream`) permite leer texto de un `InputStream`, que proporciona una secuencia de bytes.

El siguiente programa de ejemplo recupera línea a línea el contenido de una página de la wikipedia en español. Lo interesante es que utiliza un `InputStreamReader` para poder leer texto a partir de un `InputStream`.

```
import java.net.URL;

import java.io.InputStreamReader;
import java.io.BufferedReader;
import java.net.MalformedURLException;
import java.io.IOException;
import java.io.InputStream;

public class MuestraCodigoPaginaWeb {

    public static void main(String[] args) {
        String urlStr = "https://www.wikipedia.org/wiki/Luna";
        try {
            URL url = new URL(urlStr);
            try ( InputStream is = url.openConnection().getInputStream();
                  InputStreamReader isr = new InputStreamReader(is);
                  BufferedReader br = new BufferedReader(isr)) {
                System.out.printf("Contenidos de %s\n", urlStr);
                System.out.printf("-----\n");
                String linea;
```

```
        while ((linea = br.readLine()) != null) {
            System.out.println(linea);
        }
        System.out.println("-----\n");
    } catch (IOException ex) {
        System.out.printf("Error de E/S obteniendo contenidos de URL.\n");
        ex.printStackTrace();
    }
} catch (MalformedURLException ex) {
    System.out.printf("URL mal formada: %s.\n");
    ex.printStackTrace();
}
}
```

Al igual que la clase `FileReader`, la clase `InputStreamReader` tiene constructores que utilizan por defecto la codificación de caracteres UTF-8, y constructores que permiten especificar otra codificación de caracteres.

Actividad 1.13

Por supuesto, también se pueden leer datos binarios de un URL utilizando el `InputStream` obtenido con `url.openConnection().getInputStream()` en el ejemplo anterior. Esto tendrá sentido, claro, si de ese URL se obtienen datos binarios. Ese sería el caso, por ejemplo, si se tratara de un fichero de tipo PDF.

Crea un programa que descargue un fichero PDF a partir de su URL. En realidad, el programa no debe hacer nada específico para el tipo de fichero PDF. Sencillamente debe descargar un fichero a partir de un URL.

Puedes probarlo descargando el fichero disponible en <https://www-curator.jsc.nasa.gov/antmet/mmc/nakhla.pdf>.

El programa debe leer bytes uno a uno desde el `InputStream` obtenido del URL, y escribir cada byte leído en un `FileOutputStream`, para generar así un fichero con los mismos bytes que recibe. El nombre del fichero es lo que venga después del último carácter / del URL. Se da como ayuda el código del programa para completarlo. Además de añadir el código que falta donde se indica, habrá que añadir algún bloque `try ... catch` para gestionar todas las excepciones que se puedan producir en el método `main`.

```
public static void main(String[] args) {
    String urlStr = "https://www-curator.jsc.nasa.gov/antmet/mmc/nakhla.pdf";
    URL url = new URL(urlStr);
    String nomFich = urlStr.substring(urlStr.lastIndexOf("/") + 1);
    try ( InputStream is = url.openConnection().getInputStream();
        FileOutputStream fos = new FileOutputStream(nomFich) ) {
        // (completar aquí ...)
    } catch (IOException ex) {
        System.out.printf("Error de E/S obteniendo contenidos de URL.\n");
        ex.printStackTrace();
    }
}
```

10. Lectura y escritura de datos elementales (DataInputStream y DataOutputStream)

Las clases **DataInputStream** y **DataOutputStream** permiten leer y escribir, respectivamente, datos elementales. Tienen para ello métodos de la forma `readXXX()` y `writeXXX()` para leer y escribir datos de diversos tipos.

Se construyen sobre un `InputStream` y un `OutputStream`, respectivamente. Y además son clases hijas de estas últimas. Eso significa que tienen también los métodos de estas clases para leer uno o varios bytes. Es decir, son clases que amplían la funcionalidad de `InputStream` y un `OutputStream` para poder leer, además de bytes en bruto, datos de tipos elementales. Cabe destacar el método `readUTF()`, que obtiene un `String` a partir de una secuencia de bytes que corresponde a su codificación en UTF-8.

Con estos métodos, se puede plantear un mecanismo sencillo de persistencia de objetos. Es decir, mecanismos para transferir objetos de memoria a un medio de almacenamiento permanente, y viceversa. En este caso, el soporte para la persistencia serían ficheros. Se pondrá un ejemplo para una clase `Producto`, cuyo código fuente es el siguiente. La solución sería análoga para cualquier clase.

```
class Producto {  
  
    private int cod;  
    private String descr;  
    private double prUnit;  
  
    public Producto(int cod, String descr, double prUnit) {  
        this.cod = cod;  
        this.descr = descr;  
        this.prUnit = prUnit;  
    }  
  
    public int getCod() {  
        return cod;  
    }  
  
    public String getDescr() {  
        return descr;  
    }  
  
    public double getPrUnit() {  
        return prUnit;  
    }  
  
}
```

A continuación se muestran programas que leen y escriben, respectivamente, los datos de varios productos (objetos de clase `Producto`) en un fichero

El siguiente programa escribe los datos de varios clientes en un fichero. Los clientes están en un *array*.

```
import java.io.FileOutputStream;  
  
import java.io.DataOutputStream;  
import java.io.IOException;
```



```
public class EscribeFichDatos {

    public static final String NOM_FICH_SALIDA = "prod.dat";

    public static void main(String[] args) {

        Producto[] prods = {
            new Producto(5, "LAPIZ", 0.62),
            new Producto(10, "CUADERNO", 3.40)
        };

        try (FileOutputStream os = new FileOutputStream(NOM_FICH_SALIDA);
            DataOutputStream dos = new DataOutputStream(os)) {
            for (Producto unProd: prods) {
                dos.writeInt(unProd.getCod());
                dos.writeUTF(unProd.getDescr());
                dos.writeDouble(unProd.getPrUnit());
            }
        } catch (IOException ex) {
            System.out.printf("ERROR: escribiendo a fichero: %s\n", ex.getMessage());
        }
    }
}
```

El siguiente programa lee los datos de un fichero del tipo de los generados por el anterior programa. Para cada cliente, se lee el valor de sus atributos. Cuando se intenta leer un dato con cualquiera de los métodos readXXX y se ha llegado al final del fichero, se produce una excepción de tipo EOFException. Cuando esta se produce, se captura fuera del bucle, y se sigue la ejecución sin más. Una vez leídos los datos de un producto, sería muy sencillo crear un objeto de la clase Producto con ellos.

```
import java.io.FileInputStream;

import java.io.DataInputStream;
import java.io.EOFException;
import java.io.IOException;

public class LeeFichDatos {

    public static final String NOM_FICH_SALIDA = "prod.dat";

    public static void main(String[] args) {

        try (FileInputStream is = new FileInputStream(NOM_FICH_SALIDA);
            DataInputStream dis = new DataInputStream(is)) {

            for(int i=0; true; i++) {
                int cod = dis.readInt();
                String descr = dis.readUTF();
                double prUnit = dis.readDouble();
                System.out.printf("Prod. %d: %d, \"%s\", %f\n", i, cod, descr, prUnit);
            }
        }
    }
}
```

```
    } catch (EOFException ex) {  
        System.out.println("(Fin de fichero alcanzado)");  
    } catch (IOException ex) {  
        System.out.printf("ERROR: %s.\n", ex.getMessage());  
    }  
}  
}
```

Actividad 1.14

Todo lo que se pide en esta actividad debe estar en un mismo proyecto.

Crea una clase `Empleado`, con atributos `numEmp` de tipo `int`, `dni` de tipo `String`, `nombre` de tipo `String`, `salBrutoAnual` de tipo `double`, y `tParcial` de tipo `boolean`.

Crea un programa que cree tres objetos de la clase `Empleado` y escriba sus datos en un fichero, utilizando la clase `DataOutputStream`. Debes crear una nueva clase para ello, con un método `main` para que sea ejecutable.

Crea un programa que lea un fichero con datos de empleados, utilizando la clase `DataInputStream`, y que a partir de ellos cree los correspondientes objetos de la clase `Empleado`, y que los introduzca en una lista (objeto de una clase cualquiera que implemente la interfaz `List`), y después itere sobre la lista para mostrar los datos de cada empleado. Debes crear una nueva clase para ello, con un método `main` para que sea ejecutable.

11. Lectura y escritura de objetos (`ObjectInputStream` y `ObjectOutputStream`)

La solución para persistencia de objetos mostrada en el apartado anterior tiene los siguientes inconvenientes.

- Hay que crear código de programa específico para la persistencia de cada clase.
- Es difícil (mucho) la persistencia de objetos que tienen referencias a otros objetos.
- También plantea dificultades la persistencia de objetos que incluyen *arrays*.

El siguiente nivel de abstracción, una vez que se pueden escribir y leer datos elementales, es hacerlo con objetos. De manera que se tenga un mecanismo general, válido para cualquier clase. Y de manera que funcione de manera sencilla para objetos que tienen referencias a otros objetos, de manera que estas referencias se sigan y, cuando se grabe o lea un objeto, se graben o lean todos los objetos referenciados desde él.

Las clases `ObjectInputStream` y `ObjectOutputStream` proporcionan un mecanismo de este tipo. El único requisito que imponen sobre las clases para poder escribir y leer instancias suyas es que implementen la interfaz `Serializable`. Pero basta con declarar que la implementa, no hace falta implementar ningún método.

Es decir, que el único cambio que hay que hacer para ello sobre la clase `Producto` utilizada en los ejemplos anteriores es el siguiente.

```
import java.io.Serializable;  
  
class Producto implements Serializable {  
    // (...)  
}
```

Las clases `ObjectInputStream` y `ObjectOutputStream` son subclases, respectivamente, de `InputStream` y `OutputStream`. Y sus instancias se crean sobre instancias de estas clases.

Los siguientes programas leen y escriben, respectivamente, los datos de varios productos (objetos de clase Producto) en un fichero.

El siguiente programa escribe objetos de un *array* de objetos de la clase Producto en un fichero.

```
import java.io.FileOutputStream;
import java.io.ObjectOutputStream;
import java.io.IOException;

public class GrabaFichObj {

    public static final String NOM_FICH_SALIDA = "prod.dat";

    public static void main(String[] args) {

        Producto prods[] = {
            new Producto(5, "LAPIZ", 0.62),
            new Producto(10, "CUADERNO", 3.40)
        };

        try (FileOutputStream os = new FileOutputStream(NOM_FICH_SALIDA);
            ObjectOutputStream oos = new ObjectOutputStream(os)) {

            for (Producto unProd : prods) {
                oos.writeObject(unProd);
            }
        } catch (IOException ex) {
            System.out.printf("ERROR: %s.\n", ex.getMessage());
        }
    }
}
```

El siguiente programa lee objetos de la clase Producto de un fichero del tipo de los generados por el anterior programa.

```
import java.io.EOFException;
import java.io.FileInputStream;
import java.io.IOException;
import java.io.ObjectInputStream;

public class LeeFichObj {

    public static final String NOM_FICH_ENTRADA = "prod.dat";

    public static void main(String[] args) {

        try (FileInputStream is = new FileInputStream(NOM_FICH_ENTRADA);
            ObjectInputStream ois = new ObjectInputStream(is)) {

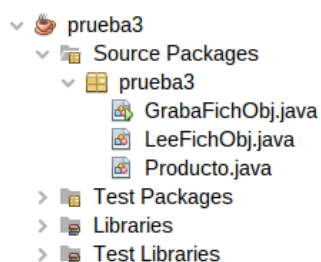
            for (int i = 1; true; i++) {
                Producto unProd = (Producto) ois.readObject();
                System.out.printf("Prod. %d: %d, \"%s\", %f\n", i,
                    unProd.getCod(), unProd.getDescr(), unProd.getPrUnit());
            }
        }
    }
}
```

```

    }

    } catch (EOFException ex) {
        System.out.println("Fin de fichero alcanzado");
    } catch (IOException ex) {
        System.out.printf("ERROR: %s.\n", ex.getMessage());
    } catch (ClassNotFoundException ex) {
        System.out.printf("ERROR: %s.\n", ex.getMessage());
    }
}
}
}

```



Es de reseñar que en la información de la clase está incluido el paquete al que pertenece. Para que todo el código funcione bien junto, debe estar en el mismo paquete. O al menos, los programas que escriben y leen objetos de la clase `Producto` deben hacer referencia a una misma clase `Producto` en un mismo paquete.

Actividad 1.15

Todo lo que se pide en esta actividad debe estar en un mismo proyecto. Puedes empezar con una copia del proyecto desarrollado para la actividad anterior.

Sobre la clase `Empleado` no se harán en principio cambios, salvo los estrictamente necesarios para que se puedan escribir instancias suyas utilizando la clase `ObjectOutputStream`.

El proyecto debe incluir un programa que cree tres objetos de la clase `Empleado` y escriba sus datos en un fichero, utilizando la clase `ObjectOutputStream`. Debe haber una clase para ello, con un método `main` para que sea ejecutable.

El proyecto debe incluir un programa que lea un fichero con objetos de la clase `Empleado`, utilizando la clase `ObjectInputStream`, los introduzca en una lista (objeto de una clase cualquiera que implemente la interfaz `List`), y después itere sobre la lista para mostrar los datos de cada empleado. Debe haber una nueva clase para ello, con un método `main` para que sea ejecutable.

Actividad 1.16

(Avanzada, opcional)

Las clases `ZipInputStream` y `ZipOutputStream` permiten, respectivamente, leer y crear ficheros comprimidos de tipo zip. Consulta la página <https://www.thecoderscorner.com/team-blog/java-and-jvm/12-reading-a-zip-file-from-java-using-zipinputstream/> y las alcanzables a partir de los enlaces que contiene para ver ejemplos de generación de ficheros zip y de extracción de sus contenidos.

Crea un programa al que se le pase como parámetro de línea de comandos la ruta de un directorio, y que cree en el directorio de ejecución del programa un fichero zip con el mismo nombre que el directorio y terminado en `.zip`. Si no se le pasa una ruta de directorio que corresponda a un directorio que exista, el programa debe mostrar un mensaje de error y terminar su ejecución. El programa debe ser consistente con el estilo de programación utilizado en general hasta ahora. Por ejemplo: en lugar de utilizar un `Logger`, debe escribir en la salida estándar y/o de error. En la medida de lo posible, para trabajar con ficheros, debe utilizar la clase `java.io.File` y no clases del paquete `java.nio.file`.

Crea un programa al que se le pase como parámetro de línea de comandos la ruta de un fichero zip y que lo descomprima en el directorio de ejecución del programa el fichero zip. El fichero podría ser uno generado por el

programa anterior.

12. Acceso aleatorio a ficheros

Cuando se utiliza acceso aleatorio a un fichero, en cada operación de lectura y escritura se puede acceder a cualquier posición del fichero para leer o escribir información en esa posición.

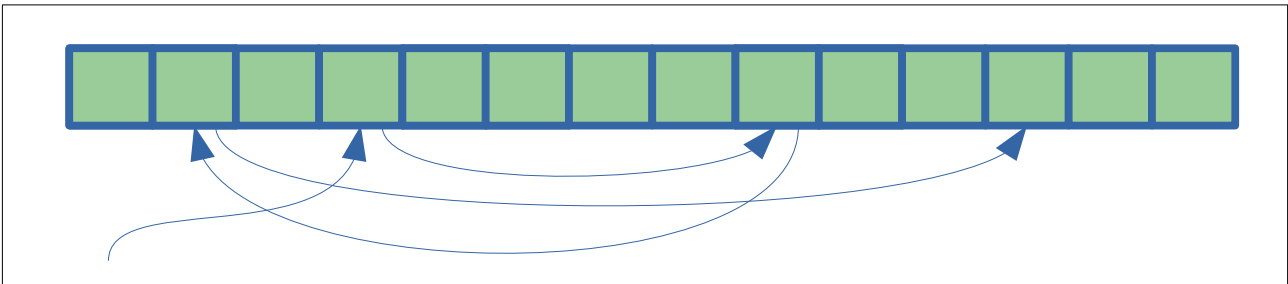


Ilustración 1.5: Acceso aleatorio a un fichero

El acceso aleatorio a ficheros se implementa en Java en la clase `RandomAccessFile`. Esta clase implementa las interfaces `Closeable` y `AutoCloseable`, por lo que, al igual que las clases para *streams* ya vistas, se puede (y se debería) utilizar con bloques `try` con recursos. Al contrario que las clases para *streams* vistas hasta ahora, la clase `RandomAccessFile` permite realizar tanto operaciones de lectura como de escritura. Además se pueden alternar operaciones de lectura y de escritura. En los constructores de esta clase se indica el modo de apertura, que determina el conjunto de operaciones que se podrán realizar.

Cuadro 1.3: Constructores de la clase `RandomAccessFile`

<pre>RandomAccessFile(File file, String mode); RandomAccessFile(String name, String mode);</pre>	<p>Abre el fichero en el modo indicado. Los modos son los siguientes.</p> <ul style="list-style-type: none"> • "r": Solo lectura. • "rw": Lectura y escritura. • "rwd", "rws": Como "rw" pero con escritura síncrona. Todas las operaciones de escritura (de datos con "rwd" y de datos y metadatos con "rws") deben haberse completado cuando termina la función. La llamada a la función puede tardar más, pero una vez concluida, se tiene la seguridad de que la operación se ha realizado. <p>Para poder abrir el fichero en el modo solicitado, por supuesto es necesario que el usuario para el que se ejecuta el programa disponga de los permisos necesarios en el sistema operativo. Si este no dispone de permiso de escritura, por ejemplo, no podrá realizar operaciones de escritura en el fichero aunque lo haga con el modo "rw".</p>
--	--

La característica distintiva de esta clase es que tiene un método **seek** que permite situar el cursor en cualquier posición del fichero para a continuación realizar operaciones de lectura o de escritura en esa posición.

Cuadro 1.4: Método `seek` de la clase `RandomAccessFile`

<pre>void seek(long pos)</pre>	<p>Desplaza el puntero de fichero a la posición indicada por pos. La siguiente operación de lectura o de escritura se realizará en esta posición.</p> <p>El valor de pos indica el desplazamiento en bytes desde el principio del fichero. Con un valor 0, el cursor se sitúa al principio del fichero.</p>
--------------------------------	---

Esta clase tiene muchos métodos disponibles en clases ya conocidas:

- `FileInputStream` y `FileOutputStream`. Para lectura y escritura de bytes en ficheros (métodos `read()` y `write()`, así como para saltar un número determinado de bytes (métodos `skip` y `skipNBytes`).
- `DataInputStream` y `DataOutputStream`. Para lectura y escritura de bytes de diversos tipos. Son los métodos `readXXX()` y `writeXXX()`, donde `XXX` es el tipo de dato. Entre estos métodos están `readUTF()` y `writeUTF` para cadenas de caracteres codificadas en UTF-8.

No solo tiene esta estos métodos ya conocidos de otras clases para realizar operaciones de lectura y de escritura, y además el método `seek` para poder realizarlas en cualquier posición del fichero. Además se pueden alternar operaciones de lectura y de escritura si el fichero se abre en modo `"rw"`.

A continuación se muestra una de las posibles aplicaciones de los ficheros de acceso aleatorio. Se muestra una clase `FicheroAccesoAleatorio` que permite gestionar un fichero con registros de longitud fija y todos con la misma estructura, definida por una lista de campos. Los datos se almacenan todos como texto. Por tanto, los ficheros que permite crear y gestionar esta clase son ficheros de texto. Para facilitar su lectura, al final de cada registro se añade un retorno de carro.

Por ejemplo, se podría utilizar un fichero para almacenar los datos de clientes. Este fichero contendrá registros con los siguientes campos.

Fichero de clientes	
Nombre	Longitud
cod_cliente	10
dni	9
nom_cliente	60

Antes de mostrar la clase, se muestra un programa que la utiliza para crear un fichero con datos de varios clientes.

```
package fichaccesoaleatorio;

import java.io.IOException;
import java.util.ArrayList;
import java.util.HashMap;
import java.util.List;
import java.util.Map;

import util.Par;

public class DrvFichAccesoAleatorio {

    public static void main(String[] args) {

        List campos = new ArrayList();
        campos.add(new Par("cod_cliente", 10));
        campos.add(new Par("dni", 9));
        campos.add(new Par("nom_cliente", 60));

        try {
            FicheroAccesoAleatorio faa = new FicheroAccesoAleatorio("clientes.dat", campos);
            Map reg = new HashMap();
            reg.put("cod_cliente", "100");
            reg.put("dni", "01234567Z");
            reg.put("nom_cliente", "ZARRA");
        }
    }
}
```

```
        faa.insertar(reg);
        reg.clear();
        reg.put("cod_cliente", "101");
        reg.put("dni", "23456789B");
        reg.put("nom_cliente", "BENÍTEZ");
        faa.insertar(reg);
        reg.clear();
        reg.put("cod_cliente", "102");
        reg.put("dni", "34567890C");
        reg.put("nom_cliente", "CERVERA");
        faa.insertar(reg);
        reg.clear();
        reg.put("cod_cliente", "103");
        reg.put("dni", "12345678A");
        reg.put("nom_cliente", "ASTORGA");
        faa.insertar(reg, 1);
    } catch (IOException e) {
        System.err.println("Error de E/S: " + e.getMessage());
    }
}
```

La clase Par es una sencilla clase auxiliar que contiene una asociación de un valor para una clave.

```
package util;

public class Par<K, V> {

    private final K clave;
    private final V valor;

    public Par(K clave, V valor) {
        this.clave = clave;
        this.valor = valor;
    }

    public K getClave() {
        return clave;
    }

    public V getValor() {
        return valor;
    }
}
```

Por último, se muestra la clase FicheroAccesoAleatorio.

```
package fichaccesoaleatorio;

import java.io.File;
import java.io.IOException;
import java.io.RandomAccessFile;
import java.util.List;
```

```
import java.util.Map;
import util.Par;

public class FicheroAccesoAleatorio {

    private final File f;
    private final List<Par<String, Integer>> campos;
    private long longReg;
    private long numReg = 0;

    FicheroAccesoAleatorio(
        String nomFich, List<Par<String, Integer>> campos) throws IOException {
        this.campos = campos;
        this.f = new File(nomFich);
        longReg = 0;
        for (Par<String, Integer> campo : campos) {
            this.longReg += campo.getValor();
        }
        if (f.exists()) {
            this.numReg = f.length() / this.longReg;
        }
    }

    public long getNumReg() {
        return numReg;
    }

    public void insertar(Map<String, String> reg) throws IOException {
        insertar(reg, this.numReg++);
    }

    public void insertar(Map<String, String> reg, long pos) throws IOException {
        try ( RandomAccessFile faa = new RandomAccessFile(f, "rws")) {
            faa.seek(pos * this.longReg);
            for (Par<String, Integer> campo : this.campos) {
                String nomCampo = campo.getClave();
                Integer longCampo = campo.getValor();
                String valorCampo = reg.get(nomCampo);
                if (valorCampo == null) {
                    valorCampo = "";
                }
                String valorCampoForm = String.format("%1$-" + longCampo + "s", valorCampo);
                faa.write(valorCampoForm.getBytes("UTF-8"), 0, longCampo);
            }
        }
    }
}
```

Puedes probar la clase anterior y pensar en qué fallos podría tener. Por ejemplo: ¿Qué sucede si el valor que se da para un campo cuando se crea un registro con el método `insertar` tiene una longitud mayor que la longitud del campo? ¿Funciona bien el método?

Las siguientes actividades consisten en determinadas ampliaciones y mejoras de la clase `FicheroAccesoAleatorio`.

Actividad 1.17

Añadir a la clase `FicheroAccesoAleatorio` un método que permita cambiar el valor de un campo de un registro. Para identificar el registro se proporcionará su posición, de la misma manera que se hace con el método `insertar`. Es decir, el primer registro tiene posición 0. Al método se le pasará también el nombre del campo y el nuevo valor para el campo.

Actividad 1.18

Añadir a la clase `FicheroAccesoAleatorio` un método que permita obtener el valor de un campo de un registro. Al método se le proporcionará la posición de un registro y el nombre del campo. El valor devuelto será de tipo `String`. Si la posición no es válida, o si no existe un campo con el nombre dado, el método devolverá `null`.

Actividad 1.19

Añadir a la clase `FicheroAccesoAleatorio` un método que permita buscar un registro que tiene un valor determinado para un campo. Al método se le proporcionará el nombre del campo y su valor, y devolverá la posición del registro en un valor de tipo `Integer`. Si se proporciona cualquier valor incorrecto o sin sentido para cualquier parámetro del método, este devolverá el valor `null`. Si no existe ningún registro con el valor dado para el campo dado, devolverá el valor -1.

Actividad 1.20

Añadir un parámetro al método anterior para la posición inicial a partir de la cual hay que buscar el valor dado en el campo dado. De esta manera, se podrán encontrar todos los registros con el valor dado para el campo dado llamando al método inicialmente con el valor 0 para la posición y repetidamente con la posición devuelta en la llamada anterior más 1, hasta que en una llamada se obtenga el valor -1.

Crear un método con el mismo nombre pero sin este nuevo parámetro, que llame al método anterior con el valor 0 para la posición.

El planteamiento es análogo a los siguientes métodos de la clase `String`.

<code>int indexOf(int ch)</code>	Devuelve el índice o posición dentro del <code>String</code> de la primera ocurrencia del carácter <code>ch</code> . El primer carácter tiene índice o posición 0.
<code>int indexOf(int ch, int fromIndex)</code>	Devuelve el índice o posición dentro del <code>String</code> de la primera ocurrencia del carácter <code>ch</code> a partir de la posición <code>fromIndex</code> .

Actividad 1.21

En el método `insertar` de la clase `FicheroAccesoAleatorio` no se ha incluido la verificación de que la posición del registro que se quiere escribir no sea mayor que el número de elementos. Se puede verificar que, si es el caso, entonces se crean registros “vacíos” cuyos bytes tienen todos valor 0. Tampoco se verifica, por cierto, que el valor para la posición no sea negativo.

Puedes incluir estas verificaciones y hacer que el método `insertar` devuelva un valor de tipo `boolean` que indique si se ha creado correctamente el registro en la posición dada. Si la posición es mayor que el número de registros no se cambiará nada en el fichero y se devolverá `false`. Si la posición es igual al número de registros, se añadirá el nuevo registro al final y se devolverá `true`. Si es menor, se sobrescribirá un registro existente y se devolverá `true`.

Actividad 1.22

(Avanzada)

Modificar la clase `FicheroAccesoAleatorio` de manera que se permitan borrar registros. Para borrar un registro, se asignará el valor 0 a todos sus bytes. Algunos registros podrán entonces estar “vacíos”, por lo que en la clase deberá haber una variable de instancia para el número de registros del fichero (de los cuales algunos podrán estar vacíos) y otra para el número de registros no vacíos, que de hecho contienen información. Deben modificarse los métodos de la clase de manera apropiada para que estas variables de instancia tengan siempre un valor correcto.

Una vez que se pueden tener registros vacíos, se pueden introducir las mejoras que se proponen a continuación.

Añadir un método que permita saber si un registro, identificado por su posición, está vacío.

Añadir métodos que permitan saber el primer registro vacío (si lo hay) y el primer registro vacío a partir de una posición dada.

Añadir un método que permita conocer el número de registros vacíos.

Cambiar el método `insertar` que no tiene un parámetro para la posición, de manera que, si en el fichero hay algún registro vacío, cree el nuevo registro en el primer registro vacío.

Para mejorar el rendimiento del método anterior, añade a la clase, y utiliza, una variable de instancia que contenga la posición del primer registro vacío, si lo hay. Si no hay ningún registro vacío, su valor será -1.