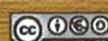


Unidad 10: Entrada y Salida en Java. Ficheros

Fundamentos de Programación. 1º de ASI



Esta obra está bajo una licencia de Creative Commons.
Autor: Jorge Sánchez Asenjo (año 2010) <http://www.jorgesanchez.net>
e-mail:info@jorgesanchez.net

Esta obra está bajo una licencia de Reconocimiento-NoComercial-CompartirIgual de Creative Commons
Para ver una copia de esta licencia, visite:
<http://creativecommons.org/licenses/by-nc-sa/2.5/es/legalcode.es>
o envíe una carta a:
Creative Commons, 559 Nathan Abbot



Reconocimiento-NoComercial-CompartirIgual 2.5 España

Usted es libre de:



copiar, distribuir y comunicar públicamente la obra

hacer obras derivadas

Bajo las condiciones siguientes:



Reconocimiento. Debe reconocer los créditos de la obra de la manera especificada por el autor o el licenciador (pero no de una manera que sugiera que tiene su apoyo o apoyan el uso que hace de su obra).



No comercial. No puede utilizar esta obra para fines comerciales.



Compartir bajo la misma licencia. Si altera o transforma esta obra, o genera una obra derivada, sólo puede distribuir la obra generada bajo una licencia idéntica a ésta.

- Al reutilizar o distribuir la obra, tiene que dejar bien claro los términos de la licencia de esta obra.
- Alguna de estas condiciones puede no aplicarse si se obtiene el permiso del titular de los derechos de autor
- Apart from the remix rights granted under this license, nothing in this license impairs or restricts the author's moral rights.

Advertencia

Los derechos derivados de usos legítimos u otras limitaciones reconocidas por ley no se ven afectados por lo anterior.

Este es un resumen legible por humanos del texto legal (la licencia completa) disponible en los idiomas siguientes:

Catalán Castellano Euskera Gallego

Para ver una copia completa de la licencia, acudir a la dirección
<http://creativecommons.org/licenses/by-nc-sa/2.5/es/legalcode.es>

(10)

entrada y salida

en Java. ficheros

esquema de la unidad

(10.1) entrada y salida en Java	6
(10.2) archivos	6
(10.2.1) introducción	6
(10.2.2) clase File	6
(10.3) clases para la entrada y la salida	10
(10.3.1) corrientes de bytes. InputStream/ OutputStream	11
(10.3.2) Reader/Writer	12
(10.3.3) InputStreamReader/ OutputStreamWriter	13
(10.3.1) DataInputStream/DataOutputStream	14
(10.3.2) ObjectInputStream/ObjectOutputStream	14
(10.3.3) BufferedReader/BufferedWriter	15
(10.3.4) PrintWriter	15
(10.3.5) PipedInputStream/PipedOutputStream	15
(10.4) entrada y salida estándar	15
(10.4.1) las clases in y out	15
(10.4.2) conversión a forma de Reader	16
(10.4.3) lectura con readLine	17
(10.5) lectura y escritura en archivos	18
(10.5.1) clases FileInputStream y FileOutputStream	18
(10.5.2) lectura y escritura byte a byte de un archivo	18
(10.6) lectura y escritura de archivos de texto	19
(10.7) archivos binarios	21
(10.7.1) escritura en archivos binarios	21
(10.7.2) lectura en archivos binarios	22
(10.8) archivos de acceso aleatorio	22
(10.8.1) RandomAccessFile	23
(10.9) serialización	23

El primer formato utiliza una ruta absoluta y el segundo una ruta relativa. En Java el separador de archivos tanto para Windows como para Linux es el símbolo /.

Otra posibilidad de construcción es utilizar como primer parámetro un objeto **File** ya hecho. A esto se añade un segundo parámetro que es una ruta que cuenta desde la posición actual.

```
File carpeta1=new File("c:/datos");//ó c|\datos  
File archivo1=new File(carpeta1,"bd.txt");
```

Si el archivo o carpeta que se intenta examinar no existe, la clase **File** no devuelve una excepción. Habrá que utilizar el método **exists**. Este método recibe **true** si la carpeta o archivo es válido (puede provocar excepciones **SecurityException**). También se puede construir un objeto **File** a partir de un objeto **URI**.

el problema de las rutas

(Las Rutas en Windows, Unix y MAC.pdf)

Cuando se crean programas en Java hay que tener muy presente que no siempre sabremos qué sistema operativo utilizará el usuario del programa. Esto provoca que la realización de rutas sea problemática porque la forma de denominar y recorrer rutas es distinta en cada sistema operativo.

Por ejemplo en Windows se puede utilizar la barra / o la doble barra invertida \\ como separador de carpetas, en muchos sistemas Unix sólo es posible la primera opción. En general es mejor usar las clases **Swing** (como **JFileDialog**) para especificar rutas, ya que son clases en las que la ruta de elige desde un cuadro y, sobre todo, son independientes de la plataforma.

También se pueden utilizar las **variables estáticas** que posee **File**. Estas son:

propiedad	uso
static char separatorChar	El carácter separador de nombres de archivo y carpetas. En Linux/Unix es / y en Windows es \, que se debe escribir como \\, ya que el carácter \ permite colocar caracteres de control, de ahí que haya que usar la doble barra. Pero Windows admite también la barra simple (/)
static String separator	Como el anterior pero en forma de String
static char pathSeparatorChar	El carácter separador de rutas de archivo que permite poner más de un archivo en una ruta. En Linux/Unix suele ser ":", en Windows es ";"
static String pathSeparator	Como el anterior, pero en forma de String

Para poder garantizar que el separador usado es el del sistema en uso:

```
String ruta="documentos/manuales/2003/java.doc";  
ruta=ruta.replace('/',File.separatorChar);
```

<code>boolean delete()</code>	Borra la carpeta y devuelve <code>true</code> si puedo hacerlo
<code>String[] list()</code>	Devuelve la lista de archivos de la carpeta representada en el objeto File.
<code>static File[] listRoots()</code>	Devuelve un array de objetos File, donde cada objeto del array representa la carpeta raíz de una unidad de disco.
<code>File[] listfiles()</code>	Igual que la anterior, pero el resultado es un array de objetos File.

métodos de archivos

método	uso
<code>boolean isFile()</code>	Devuelve <code>true</code> si el objeto File es un archivo y <code>false</code> si es carpeta o si no existe.
<code>boolean renameTo(File f2)</code>	Cambia el nombre del archivo por el que posee el archivo pasado como argumento. Devuelve <code>true</code> si se pudo completar la operación.
<code>boolean delete()</code>	Borra el archivo y devuelve <code>true</code> si puedo hacerlo
<code>long length()</code>	Devuelve el tamaño del archivo en bytes (en el caso del texto devuelve los caracteres del archivo)
<code>boolean createNewFile() throws IOException</code>	Crea un nuevo archivo basado en la ruta dada al objeto File. Hay que capturar la excepción <code>IOException</code> que ocurriría si hubo error crítico al crear el archivo. Devuelve <code>true</code> si se hizo la creación del archivo vacío y <code>false</code> si ya había otro archivo con ese nombre.
<code>static File createTempFile(String prefijo, String sufijo) throws IOException</code>	Crea un objeto File de tipo archivo temporal con el prefijo y sufijo indicados. Se creará en la carpeta de archivos temporales por defecto del sistema. El <code>prefijo</code> y el <code>sufijo</code> deben de tener al menos tres caracteres (el sufijo suele ser la extensión), de otro modo se produce una excepción del tipo <code>IllegalArgumentException</code> Requiere capturar la excepción <code>IOException</code> que se produce ante cualquier fallo en la creación del archivo
<code>static File createTempFile(String prefijo, String sufijo, File directorio)</code>	Igual que el anterior, pero utiliza el directorio indicado.
<code>void deleteOnExit()</code>	Borra el archivo cuando finaliza la ejecución del programa

- TEMA 08. ENTRADA Y SALIDA

- Paquete java.io contiene todas las funciones de Entrada y Salida en Java.
- Entrada y Salida jerarquía de classes.pdf.

* FICUEROS.

- Hasta ahora se almacenaron datos en memoria. Los archivos se puede almacenar en formato de forma persistente.

* FUNCIONAMIENTO LECTURA/ESCRITURA FICUEROS

- Algoritmos de Lectura y Escritura .pdf.

* La clase FILE

- Solo operaciones de información. No métodos de acceso.
- Métodos para escribir, crear carpetas, borrar archivos, renombrarlos.
- Manual Java. (La clase File)
 - Constructores:
 - Método exists()

- Práctica de archivos.

- Los Rulos en Windows, Unix y MAC.pdf
- System.getProperty ("os.name")
- Métodos de la clase FILE (Java.io.)
- Clase File. Métodos. Resumen.pdf

de datos en cada extremo. Normalmente las secuencias son de bytes, pero se pueden formatear esos bytes para permitir transmitir cualquier tipo de datos.

Los datos fluyen en serie, byte a byte. Se habla entonces de un **stream** (corriente de datos, o mejor dicho, corriente de bytes). Pero también podemos utilizar streams que transmiten caracteres Java (tipo **char** Unicode, de dos bytes), se habla entonces de un **reader** (si es de lectura) o un **writer** (escritura).

En el caso de las excepciones, todas las que provocan las excepciones de E/S son derivadas de **IOException** o de sus derivadas. Además son habituales ya que la entrada y salida de datos es una operación crítica porque con lo que la mayoría de operaciones deben ir inmersas en un **try**.

(10.3.1) corrientes de bytes. InputStream/ OutputStream

Los Streams de Java son corrientes de datos binarios accesibles byte a byte. Estas dos clases **abstractas**, definen las funciones básicas de lectura y escritura de una secuencia de bytes pura (sin estructurar). Estas corrientes de bits, no representan ni textos ni objetos, sino datos binarios puros. Poseen numerosas subclases; de hecho casi todas las clases preparadas para la lectura y la escritura, derivan de estas.

Los métodos más importantes son **read** (leer) y **write** (escribir), que sirven para leer un byte del dispositivo de entrada o escribir un byte respectivamente.

métodos de InputStream

método	uso
int available()	Devuelve el número de bytes de entrada
void close()	Cierra la corriente de entrada. Cualquier acceso posterior generaría una IOException .
void mark(int bytes)	Marca la posición actual en el flujo de datos de entrada. Cuando se lea el número de bytes indicado, la marca se elimina.
boolean markSupported()	Devuelve verdadero si en la corriente de entrada es posible marcar mediante el método mark .
int read()	Lee el siguiente byte de la corriente de entrada y le almacena en formato de entero. Devuelve -1 si estamos al final del fichero
int read(byte[] búfer)	Lee de la corriente de entrada hasta llenar el array búfer .
void reset()	Coloca el puntero de lectura en la posición marcada con mark .
long skip()	Se salta de la lectura el número de bytes indicados

método	uso
<code>void reset()</code>	Coloca el puntero de lectura en la posición marcada con mark .
<code>long skip()</code>	Se salta de la lectura el número de bytes indicados

métodos de Writer

método	uso
<code>void close()</code>	Cierra la corriente de salida. Cualquier acceso posterior generaría una IOException .
<code>void flush()</code>	Vacía los búferes de salida de la corriente de datos
<code>void write(int byte)</code>	Escribe un byte en la corriente de salida
<code>void write(byte[] bufer)</code>	Escribe todo el array de bytes en la corriente de salida
<code>void write(byte[] buffer, int posInicial, int numBytes)</code>	Escribe el array de bytes en la salida, pero empezando por la posición inicial y sólo la cantidad indicada por numBytes .
<code>void write(String texto)</code>	Escribe los caracteres en el String en la corriente de salida.
<code>void write(String buffer, int posInicial, int numBytes)</code>	Escribe el String en la salida, pero empezando por la posición inicial y sólo la cantidad indicada por numBytes .

(10.3.3) InputStreamReader/ OutputStreamWriter

Son clases que sirven para adaptar la entrada y la salida. La razón es que las corrientes básicas de E/S son de tipo **Stream**. Estas clases consiguen adaptarlas a corrientes **Reader/Writer**.

Puesto que derivan de las clases **Reader** y **Writer**, ofrecen los mismos métodos que éstas.

Para ello poseen un constructor que permite crear objetos **InputStreamReader** pasando como parámetro una corriente de tipo **InputStream** y objetos **OutputStreamWriter** partiendo de objetos **OutputStream**.

- ◆ **readObject.** Devuelve un objeto Object de los datos de la entrada. En caso de que no haya un objeto o no sea serializable, da lugar a excepciones. Las excepciones pueden ser:
 - **ClassNotFoundException**
 - **InvalidClassException**
 - **StreamCorruptedException**
 - **OptionalDataException**
 - **IOException** genérica.

La clase ObjectOutputStream posee el método de escritura de objetos **writeObject** al que se le pasa el objeto a escribir. Este método podría dar lugar en caso de fallo a excepciones **IOException**, **NotSerializableException** o **InvalidClassException**.

(10.3.3) **BufferedInputStream/BufferedOutputStream/ BufferedReader/BufferedWriter**

La palabra **buffered** hace referencia a la capacidad de almacenamiento temporal en la lectura y escritura. Los datos se almacenan en una memoria temporal antes de ser realmente leídos o escritos. Se trata de cuatro clases que trabajan con métodos distintos pero que suelen trabajar con las mismas corrientes de entrada que podrán ser de bytes (**Input/OutputStream**) o de caracteres (**Reader/Writer**).

La clase **BufferedReader** aporta el método **readLine** que permite leer caracteres hasta la presencia de **null** o del salto de línea.

(10.3.4) **PrintWriter**

Clase pensada para secuencias de datos orientados a la impresión de textos. Es una clase escritora de caracteres en flujos de salida, que posee los métodos **print** y **println**, que otorgan gran potencia a la escritura.

(10.3.5) **PipedInputStream/PipedOutputStream**

Permiten realizar canalizaciones entre la entrada y la salida; es decir lo que se lee se utiliza para una secuencia de escritura o al revés.

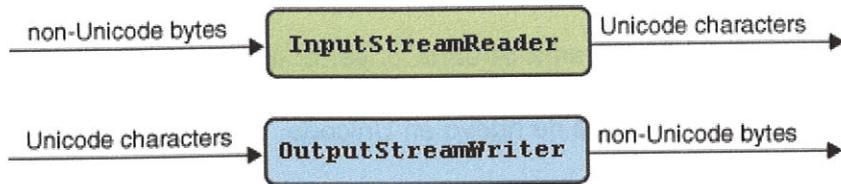
(10.4) **entrada y salida estándar**

(10.4.1) **las clases in y out**

java.lang.System es una clase que poseen multitud de pequeñas clases relacionadas con la configuración del sistema. Entre ellas están la clase **in** que es un objeto de tipo **InputStream** que representa la entrada estándar (normalmente el teclado) y **out** que es un **OutputStream** que representa a la

Carácter y Byte Streams

El paquete **java.io** ofrece clases que le permiten convertir entre corrientes de caracteres Unicode y flujos de bytes de texto no Unicode. Con la clase **InputStreamReader**, puede convertir flujos de bytes a streams de caracteres. Utilice la clase **OutputStreamWriter** para traducir corrientes de caracteres en corrientes de bytes. La siguiente figura ilustra el proceso de conversión:



Al crear objetos **InputStreamReader** y **OutputStreamWriter**, se especifica la codificación de bytes que desea convertir. Por ejemplo, para traducir un archivo de texto en la codificación UTF-8 en Unicode, se crea una **InputStreamReader** de la siguiente manera:

```
FileInputStream fis = new FileInputStream("test.txt");
InputStreamReader isr = new InputStreamReader(fis, "UTF8");
```

Si se omite el identificador de codificación, **InputStreamReader** y **OutputStreamWriter** se basan en la codificación predeterminada. Se puede determinar qué codificación usa **InputStreamReader** y **OutputStreamWriter** invocando el método **getEncoding**, de la siguiente manera:

```
InputStreamReader defaultReader = new InputStreamReader(fis);
String defaultEncoding = defaultReader.getEncoding();
```

El programa convierte una secuencia de caracteres Unicode de un objeto **String** en un **FileOutputStream** de bytes codificados en UTF-8. El método que realiza la conversión se llama **writeOutput**:

```
static void writeOutput(String str) {
    try{
        FileOutputStream fos=new FileOutputStream("test.txt");
        Writer out=new OutputStreamWriter(fos,"UTF8");
        out.write(str);
        out.close();
    }
    catch (IOException e){
        e.printStackTrace();
    }
}
```

El método **readInput** lee los bytes codificados en UTF-8 desde el archivo creado por el método **writeOutput**. Un objeto **InputStreamReader** convierte los bytes de UTF-8 a Unicode y devuelve el resultado en una cadena. El método **readInput** es como sigue:

```
static String readInput() {
    StringBuffer buffer = new StringBuffer();
    try {
        FileInputStream fis = new FileInputStream("test.txt");
        InputStreamReader isr = new InputStreamReader(fis, "UTF8");
        Reader in = new BufferedReader(isr);
        int ch;
        while ((ch = in.read()) > -1) {
```

clases **Reader** y **Writer**, hubo que resolver el problema de tener que usar las dos anteriores.

La solución fueron dos clases: **InputStreamReader** y **OutputStreamWriter**. Se utilizan para convertir secuencias de byte en secuencias de caracteres según una determinada configuración regional. Permiten construir objetos de este tipo a partir de objetos **InputStream** u **OutputStream**. Puesto que son clases derivadas de **Reader** y **Writer** el problema está solucionado.

El constructor de la clase **InputStreamReader** requiere un objeto **InputStream** y, opcionalmente, una cadena que indique el código que se utilizará para mostrar caracteres (por ejemplo “ISO-8914-1” es el código Latín 1, el utilizado en la configuración regional). Sin usar este segundo parámetro se construye según la codificación actual (es lo normal).

Lo que hemos creado de esa forma es un objeto *convertidor*. De esa forma podemos utilizar la función **read** orientada a caracteres Unicode que permite leer caracteres extendidos. Esta función posee una versión que acepta arrays de caracteres, con lo que la versión **writer** del código anterior sería:

```
InputStreamReader stdin=new InputStreamReader(System.in);
char caracter[]=new char[1024];
int numero=-1;
try{
    numero=stdin.read(caracter);
}
catch(IOException e){
    System.out.println("Error en la lectura");
}
for(int i=0;i<numero;i++)
    System.out.print(caracter[i]);
```

(10.4.3) lectura con readLine

El uso del método **read** con un array de caracteres sigue siendo un poco enrevesado, además hay que tener en cuenta que el teclado es un dispositivo con buffer de lectura. Por ello para leer cadenas de caracteres se suele utilizar la clase **BufferedReader**. La razón es que esta clase posee el método **readLine** que permite leer una línea de texto en forma de String, que es más fácil de manipular. Esta clase usa un constructor que acepta objetos **Reader** (y por lo tanto **InputStreamReader**, ya que desciende de ésta) y, opcionalmente, el número de caracteres a leer.

Hay que tener en cuenta que el método **readLine** (como todos los métodos de lectura) puede provocar excepciones de tipo **IOException** por lo que, como ocurría con las otras lecturas, habrá que capturar dicha lectura.

Ejemplo, suponiendo que existe un archivo de texto llamado prueba.txt, este código le muestra por pantalla:

```
File f=new File("prueba.txt");
try {
    FileInputStream fis=new FileInputStream(f);
    int x=0;
    while(x!= -1){
        x=fis.read();
        System.out.print((char)x);
    }
}
catch (IOException e) {
    e.printStackTrace();
}
```

(10.6) lectura y escritura de archivos de texto

Como ocurría con la entrada estándar, se puede convertir un objeto **FileInputStream** o **FileOutputStream** a forma de **Reader** o **Writer** mediante las clases **InputStreamReader** y **OutputStreamWriter**. Y esto es más lógico cuando manejamos archivos de texto.

Existen además dos clases que manejan caracteres en lugar de bytes (lo que hace más cómodo su manejo), son **FileWriter** y **FileReader**.

La construcción de objetos del tipo **FileReader** se hace con un parámetro que puede ser un objeto **File** o un **String** que representarán a un determinado archivo.

La construcción de objetos **FileWriter** se hace igual sólo que se puede añadir un segundo parámetro booleano que, en caso de valer **true**, indica que se abre el archivo para añadir datos; en caso contrario se abriría para grabar desde cero (se borraría su contenido).

Para escribir se utiliza **write** que es un método void que recibe como parámetro lo que se desea escribir en formato **int**, **String** o en forma de array de caracteres.

Ejercicio nº 51

- Crea con el Bloc de notas un fichero de texto con el contenido que quieras.
Crea un programa Java que muestra su contenido por pantalla, sin espacios y sin retornos de carro.

Ejercicio nº 52

- Pdf adjunto

(19)

Ejercicio nº. 50

Crea una aplicación Java que pida el nombre de dos ficheros de texto y la ruta para un fichero de destino.

Debes copiar el contenido de los dos ficheros en uno, este tendrá el nombre de los dos ficheros separados por un guion bajo, y se guardará en la ruta que hayamos indicado.

Para unir los dos ficheros en uno, crea un método donde le pases como parámetro todas las rutas. En este método, aparte de copiar debe comprobar si existe el fichero de destino, mostrando, en este caso, un mensaje informándonos de si queremos o no sobrescribirlo.

Por **ejemplo**, si tengo un fichero **FicheroA.txt** con "ABC" como contenido, un fichero **FicheroB.txt** con "DEF" como contenido, y una ruta de destino **D:/**, el resultado será un fichero llamado **FicheroA_FicheroB.txt** en la ruta **D:/** con el contenido "**ABCDEF**".

La escritura se realiza con el método **write** que permite grabar caracteres, Strings y arrays de caracteres. **BufferedWriter** además permite utilizar el método **newLine** que escriba un salto de línea en el archivo; lo que arregla el problema de la compatibilidad entre plataformas por que los caracteres para el cambio de párrafo son distintos según cada sistema operativo (o incluso por diferentes circunstancias).

(10.7) archivos binarios

Para archivos binarios se suelen utilizar las clases **DataInputStream** y **DataOutputStream**. Estas clases están mucho más preparadas para escribir datos de todo tipo.

(10.7.1) escritura en archivos binarios

El proceso sería:

- (1) Crear un objeto **FileOutputStream** a partir de un objeto **File** que posea la ruta al archivo que se desea escribir (para añadir usar el segundo parámetro del constructor indicando **true**)
- (2) Crear un objeto **DataOutputStream** asociado al objeto anterior. Esto se realiza mediante el constructor de **DataOutputStream**.
- (3) Usar el objeto del punto 2 para escribir los datos mediante los métodos **writeTipo** donde **tipo** es el tipo de datos a escribir (**int**, **double**, ...). A este método se le pasa como único argumento los datos a escribir.
- (4) Se cierra el archivo mediante el método **close** del objeto **DataOutputStream**.

Ejemplo:

```
File f=new File("d:/prueba.out"); Random r=new Random();
double d=18.76353;
try{
    FileOutputStream fis=new FileOutputStream(f);
    DataOutputStream dos=new DataOutputStream(fis);
    for (int i=0;i<234;i++){ //Se repite 233 veces
        dos.writeDouble(r.nextDouble());//Nº aleatorio
    }
    dos.close();
}
catch(FileNotFoundException e){
    System.out.println("No se encontro el archivo");
}
catch(IOException e){
    System.out.println("Error al escribir");
}
```

(21)

Ejercicio nº 53

- Pdf adjunto

Ejercicio nº. 5

Crea una aplicación que almacene los datos básicos de un vehículo:

- Matricula(String)
- Marca (String)
- Tamaño de deposito (double)
- Modelo (String)

en ese orden y de uno en uno usando la clase DataOutputStream.

Los datos anteriores se pedirán por teclado y se irán añadiendo al fichero (no se sobrescriben los datos) cada vez que ejecutemos la aplicación.

Tras añadir un vehículo se mostrarán por pantalla todos los datos de los coches almacenados en el fichero.

Ejercicio nº. 52

Crear una aplicación Java que permita gestionar las fichas de los alumnos de un Instituto.

La gestión se llevará a cabo a través de un Menú con las siguientes opciones:

- 1.- Añadir ficha.
- 2.- Eliminar ficha.
- 3.- Mostrar todas las fichas.
- 4.- Mostrar las fichas correspondientes a un año de nacimiento.
- 5.- Mostrar las fichas correspondientes a un grupo.
- 6.- Salir.

El fichero tendrá la siguiente estructura:

Campo
Nombre
E-Mail
Fecha de Nacimiento
Grupo

(10.8.1) RandomAccessFile

Esta clase permite leer archivos en forma aleatoria. Es decir, se permite leer cualquier posición del archivo en cualquier momento. Los archivos anteriores son llamados secuenciales, se leen desde el primer byte hasta el último.

Esta es una clase primitiva que implementa las interfaces **DataInput** y **DataOutput** y sirve para leer y escribir datos.

La construcción requiere de una cadena que contenga una ruta válida a un archivo o de un archivo File. Hay un segundo parámetro obligatorio que se llama **modo**. El modo es una cadena que puede contener una **r** (lectura), **w** (escritura) o ambas, **rw**.

Como ocurría en las clases anteriores, hay que capturar la excepción **FileNotFoundException** cuando se ejecuta el constructor para el caso de que haya problemas al crear el objeto File.

```
File f=new File("D:/prueba.out");
RandomAccessFile archivo = new RandomAccessFile(f, "rw");
```

Los métodos fundamentales son:

- ◆ **void seek(long pos)**. Permite colocarse en una posición concreta, contada en bytes, en el archivo. Lo que se coloca es el puntero de acceso que es la señal que marca la posición a leer o escribir.
- ◆ **long getFilePointer()**. Posición actual del puntero de acceso
- ◆ **long length()**. Devuelve el tamaño del archivo
- ◆ **readBoolean, readByte, readChar, readInt, readDouble, readFloat, readUTF, readLine**. Funciones de lectura, equivalentes a las disponibles en la clase **DataInputStream**. Leen un dato del tipo indicado. En el caso de **readUTF** lee una cadena en formato Unicode.
- ◆ **writeBoolean, writeByte, writeBytes, writeChar, writeChars, writeInt, writeDouble, writeFloat, writeUTF, writeLine**. Funciones de escritura. Todas reciben como parámetro, el dato a escribir. Escriben encima de lo ya escrito. Para escribir al final hay que colocar el puntero de acceso al final del archivo.

Ejercicio u'54) PDF Adjunto.

(10.9) serialización

Es una forma automática de guardar y cargar el estado de un objeto. Se basa en la interfaz **Serializable** (en el paquete **java.io**) que es la que permite esta operación. Si una clase implementa esta interfaz puede ser guardado y restaurado directamente en un archivo

Cuando se desea utilizar un objeto para ser almacenado con esta técnica, debe ser incluida la instrucción **implements Serializable** la cabecera de

Ejercicio nº. 5

Crear una aplicación Java similar a la del Ejercicio nº. 5, utilizando **Serialización**.

Crea una clase Vehículo con los atributos matrícula, marca, tamaño del depósito y modelo, con sus respectivos métodos get y el constructor, que será invocado con todos los atributos.

El atributo tamañoDeposito no se incluirá en el fichero (aunque se pidan sus datos por teclado)

Ejercicio nº. 56

En una empresa se disponen de 8 líneas de teléfono internas, se desea crear un fichero de texto donde se vayan registrando todas las llamadas que se efectúan entre línea y línea de acuerdo al siguiente formato:

Lineaquellema_Lineadestino

Lineaquellema_Lineadestino

Se creará un fichero para 20 llamadas.

Se desea poder ver el contenido del fichero de llamadas una vez creado.

Se desea también, poder ver resumen de llamadas por línea, y poder averiguar si una línea ha efectuado una llamada a otra.

```
Fin de carga de arrays
Linea 1: 4 llamadas : 6 7 3 7
Linea 2: 4 llamadas : 6 3 3 8
Linea 3: 0 llamadas :
Linea 4: 4 llamadas : 5 6 8 3
Linea 5: 2 llamadas : 1 3
Linea 6: 2 llamadas : 5 4
Linea 7: 6 llamadas : 6 5 6 3 2 1
Linea 8: 3 llamadas : 4 7 3
Linea de llamada: 6
Linea de destino: 7
Resultado: false
Linea de llamada: 1
Linea de destino: 7
Resultado: true
Linea de llamada: -
```

Ejercicio nº. 5

Crear y probar la siguiente clase implementando los métodos que a continuación se listan. Gestionar las posibles excepciones que se pudieran producir a la hora de efectuar las operaciones:

Class Datos_Alumnos{

void conectar(){ } //Este método se encargará de establecer la conexión a la base de datos.

void crearTablas(){ } //Este método se encargará de crear la tabla **datos**, con los campos: nombre, edad y tasa_matricula.

void grabarDatos(Persona datos){ } //Esté método se encargará de añadir información de un alumno a la BBDD. Persona será una clase con 3 atributos que corresponden a los 3 campos definidos anteriormente (nombre, edad, tasa_matricula).

Añadir los métodos para realizar los siguientes accesos para consulta, actualización y borrado de algún registro de la tabla creada, gestionando las excepciones pertinentes.

- a) Consultas: Alumnos cuya edad sea menor que una dada.
Consultas: Todos los alumnos.
- b) Actualizaciones: Incrementar un porcentaje introducido por teclado, la tasa_matrícula de todos los alumnos cuya edad sea mayor de 31.
- d) Borrar de la tabla a los mayores de una edad introducida por teclado.

Ejercicio nº. 56

Crear una tabla llamada **coches**, con los siguientes campos:

MATRICULA	TEXTO
MARCA	TEXTO
MODELO	TEXTO
CILINDRADA	NUMERICO
COLOR	TEXTO
DNI	TEXTO

Introducir, al menos estos 10 registros.

MATRIC.	MARCA	MODELO	CILINDRADA	COLOR	DNI
3434-BJJ	SEAT	TOLEDO	2000	GRIS	1.234.567
1234-BJK	SEAT	IBIZA	1400	VERDE	33.333.333
5432-BCD	RENAULT	MEGANE	1900	AZUL	44.444.444
6545-BMW	RENAULT	MEGANE	1800	VERDE	33.333.333
9929-BSK	CITROEN	XSARA	1600	ROJO	5.555.555
1028-BBB	FORD	FIESTA	1400	BLANCO	9.999.999
1566-BYD	FORD	FOCUS	1600	ROJO	44.444.444
2222-BPR	SEAT	IBIZA	1600	NEGRO	5.555.555
6574-CCP	TOYOTA	COROLLA	1200	ROJO	1.111.111
8594-BBC	TOYOTA	AVENSIS	1000	AMARILLO	88.888.888

Crear y probar las siguientes clases implementando los métodos que a continuación se listan. Gestionar las posibles excepciones que se pudieran producir a la hora de efectuar las operaciones:

Class coche {

 Un **atributo** por cada campo de la tabla.

coche(){ } //Método constructor que podrá o no recibir como parámetro los posibles atributos de la clase coche.

}

Class concesionario {

ArrayList <coche> Coches;

Concesionario(){ } //Método constructor que guarda en el atributo Coches, un objeto para cada registro de la tabla coches.

void Persistencia_coches(){ } //Guarda, de nuevo, el ArrayList Coches en la tabla coches, con los cambios que se hayan podido realizar sobre los datos.

}

Cómo Crear un Jar con Java desde la Consola

Para crear los archivos .jar debemos primero crear un Archivo de Manifiesto en el cual se deberá poner el nombre de la clase que contenga el método main, este archivo puede ser creado a partir de cualquier editor de texto:

```
Main-Class: HolaMundo  
Sealed: true
```

Donde HolaMundo es el Nombre de la Clase que tiene mi método main, una vez escrito eso lo guardamos con el siguiente nombre en el mismo lugar donde tenemos nuestros archivos .java y .class:

temp.mf

Desde la línea de comando, ejecutamos la siguiente sentencia:

jar -cf HolaMundo.jar HolaMundo.class

Después de haber ejecutado la sentencia anterior ya tendremos el archivo HolaMundo.jar en nuestro directorio, y contendrá el fichero de manifiesto, la clase o clases que tenga nuestro proyecto.

Ahora al archivo .jar que tenemos, vamos a modificarlo para indicarle que archivo de manifiesto debe usar:

jar cmf temp.mf HolaMundo.jar HolaMundo.class

Se permiten comodines (por ejemplo, para incluir todos los archivos class de la carpeta o que cumplan con alguna condición dada):

- **jar cmf temp.mf HolaMundo.jar fichero\$.class**
- **jar cmf temp.mf HolaMundo.jar *.class**

Ya podemos ejecutar nuestro archivo en la línea de comandos:

java -jar HolaMundo.jar