Tema 8. Ficheros

Contenido

1 FICHEROS

Un fichero es una agrupación de datos.

Cada dato es un registro, y cada registro se compone de campos.

1.1 CLASIFICACIÓN DE FICHEROS

1.1.1 Según tipo de valores permitidos

DE TEXTO

Los datos se almacenan usando código **ASCII** y se pueden procesar con editores de texto.

BINARIOS

Los datos se almacenan en binario. Permiten todos los valores para cada byte.

1.1.2 Según tipo de organización

SECUENCIALES

Los registros que lo forman se han escrito o creado en posiciones físicas contiguas en el soporte de almacenamiento y en la misma secuencia en que se introdujeron los datos.

Estos ficheros son soportados por la mayoría de sistemas operativos y lenguajes de programación, y su modo de acceso es exclusivamente secuencial, y por lo tanto para leer el registro N hay que pasar ó leer los N-1 precedentes. Los registros serán accedidos **en el mismo orden en que se escribieron en el archivo**, salvo que se apliquen técnicas de clasificación sobre dicho archivo.

ALEATORIOS

Un archivo posee organización directa (también se llaman archivos de acceso aleatorio), cuando los registros pueden ser escritos o leídos posicionándose en el archivo mediante un numero ordinal (**en Java va de 0 en adelante**) que denominaremos dirección lógica del registro o simplemente dirección de registro.

Es decir en un archivo de organización directa yo puedo leer (si existe) el registro número 7 o escribir directamente en el registro numero 1000. Los registros de estos archivos son homogéneos y de tamaño fijo lo cual ayuda al compilador a calcular el byte por el cual tiene que empezar a leer.

Estos archivos permiten también el acceso secuencial.

2 FLUJOS - STREAM

Cualquier programa que vaya a llevar a cabo entrada o salida de información necesita usar flujos. En Java, Stream.

- El Stream representa la conexión entre el programa y el dispositivo de entrada o de salida.
- De dicha conexión se encarga Java mediante una serie de clases.
- Los programas leen o escriben en el flujo, que puede estar conectado a un dispositivo (por ejemplo la consola) o a otro (por ejemplo un disco magnético u óptico).
- El flujo es, por tanto, una abstracción, de tal forma que las operaciones que realizan los programas son sobre el flujo, independientemente del dispositivo al que esté asociado.

2.1 TIPOS DE FLUJOS

2.1.1 FLUJOS DE CARACTERES

Todas las clases que gestionan los flujos de caracteres implementan las clases abstractas **Reader** y **Writer.**

2.1.2 FLUJOS DE BYTES O BINARIOS

Se usan para trabajar con datos binarios. Todas estas clases implementan las clases abstractas: **InputStream** y **OutputStream**.

2.2 FLUJOS

Todas las operaciones de entrada y salida realmente se realizan en bytes, por ello existen las clases **InputStreamReader** y **OutputStreamWriter** que hacen posible la conversión de byte a carácter.

Como ejemplo, la siguiente instrucción se puede usar cuando hay una entrada de datos por teclado, con el fin de tomar los datos que proceden del teclado en forma de byte y transformarlos en caracteres:

InputStreamReader in = new InputStreamReader(System.in)

La clase InputStreamReader requiere que el parámetro que se le pase sea de tipo InputStream, es decir, una entrada de datos en bytes, como lo es System.in

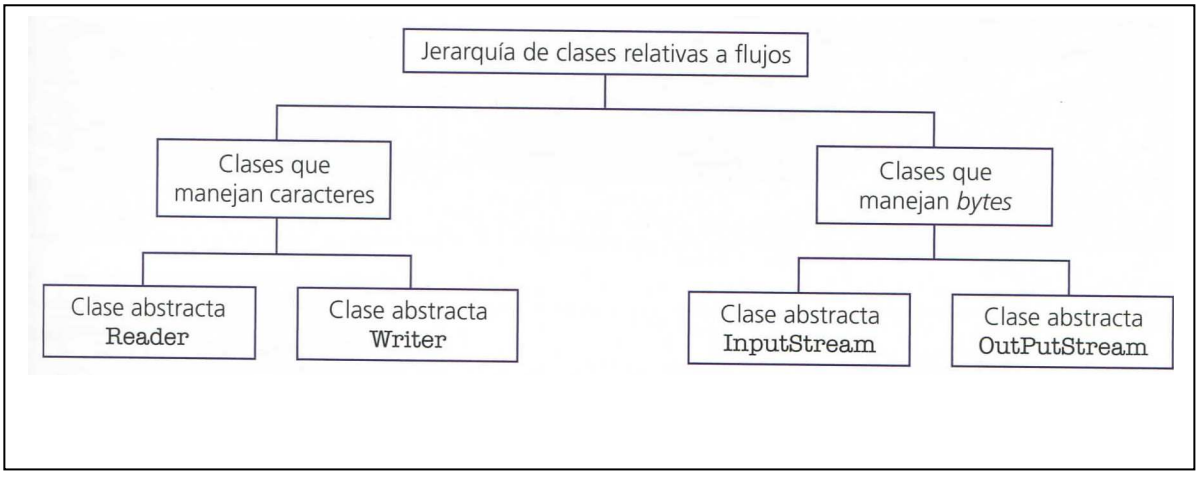
2.3 FLUJOS PREDEFINIDOS

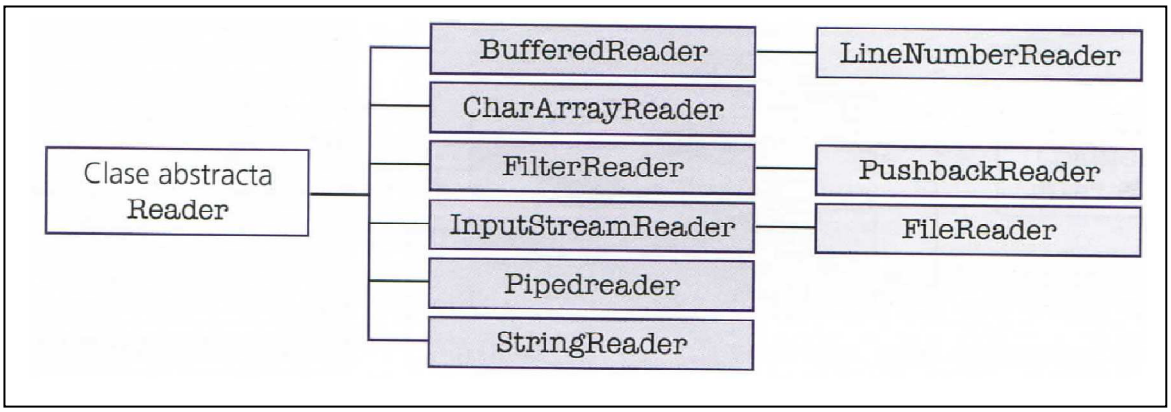
En Java, la entrada desde el teclado y la salida por pantalla están gestionadas por la clase **System**.

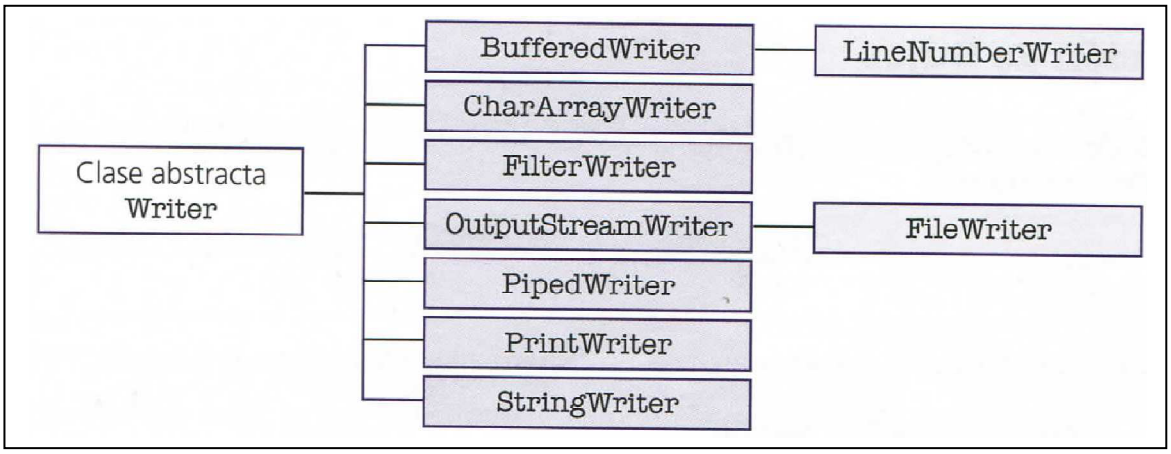
Esta clase pertenece al paquete **java.lang** y tienen tres atributos, los llamados flujos predefinidos: **in**, **out** y **err**, que son public y static, por eso podemos poner System.in, System.out y System.err.

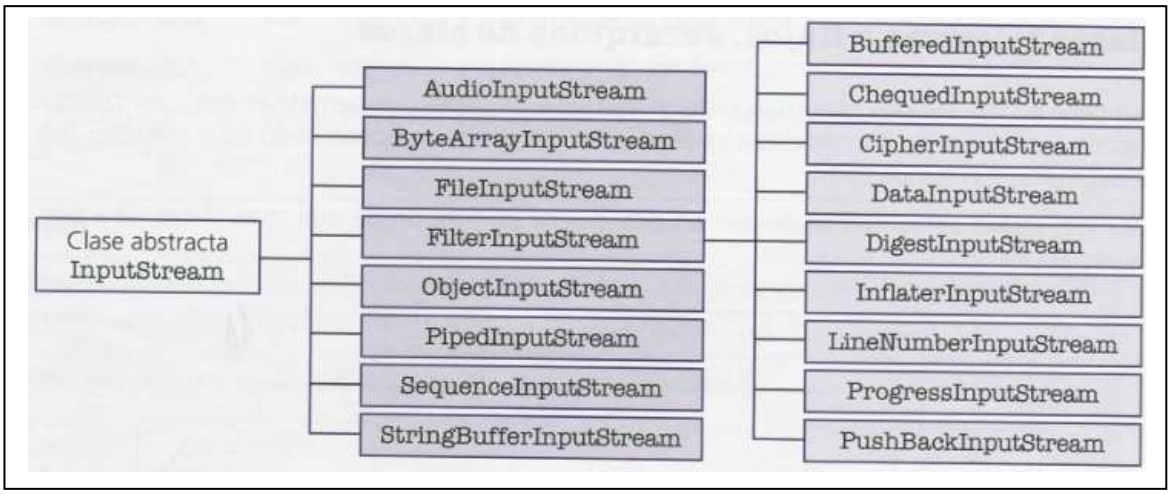
- **Sytem.in**: hace referencia a la entrada estándar de datos.
- **System.out**: hace referencia a la salida estándar de datos.
- **System.err**: hace referencia a la salida estándar de información de errores.

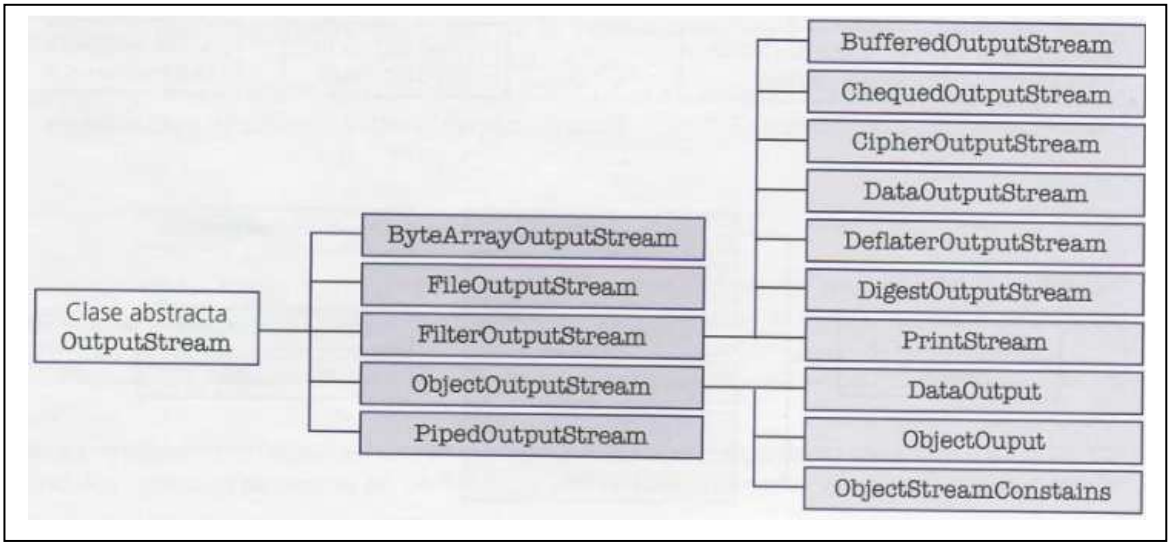
3 JERARQUÍA DE CLASES




Clase abstracta Reader
BufferedReader
CharArrayReader
FilterReader
InputStreamReader
Pipedreader
StringReader
LineNumberReader
PushbackReader
FileReader


Clase abstracta Writer
BufferedWriter
CharArrayWriter
FilterWriter
OutputStreamWriter
PipedWriter
PrintWriter
StringWriter
LineNumberWriter
FileWriter


Clase abstracta InputStream
AudioInputStream
ByteArrayInputStream
FileInputStream
FilterInputStream
ObjectInputStream
PipedInputStream
SequenceInputStream
StringBufferInputStream
BufferedInputStream
ChequedInputStream
CipherInputStream
DataInputStream
DigestInputStream
InflaterInputStream
LineNumberInputStream
ProgressInputStream
PushBackInputStream


Clase abstracta OutputStream
ByteArrayOutputStream
FileOutputStream
FilterOutputStream
ObjectOutputStream
PipedOutputStream
BufferedOutputStream
ChequedOutputStream
CipherOutputStream
DataOutputStream
DeflaterOutputStream
DigestOutputStream
PrintStream
DataOutput
ObjectOuput
ObjectStreamConstains

4 UTILIZACIÓN DE FLUJOS

- Con el uso de estas clases se puede hacer que un programa se comunique con el exterior.
- Java, independientemente de dónde procedan los datos, trabaja exactamente igual.

4.1 CLASE File

Gracias a la clase **File**, que tiene el API de Java, podemos obtener información importante sobre el fichero con el que vamos a trabajar, como su ruta, **si es un fichero o un directorio**, el tamaño, última fecha de modificación, etc.

También podemos cambiar el nombre del fichero, borrarlo, etc.

Esta clase (File) **NO** permite acceder a la información que contiene el propio fichero.

4.1.1 Constructores de File:

Constructor	Explicación
File(String nombre)	Recibe en la instanciación del objeto la ruta completa donde está el fichero junto con el nombre. Por defecto, si no se indica, lo busca en la carpeta del proyecto. **nombre** puede ser también la ruta a un directorio, sin indicar al final el nombre de ningún fichero.
File(String ruta, String nombre)	Recibe en la instanciación del objeto la ruta completa donde está el fichero, como primer parámetro, y el nombre del fichero como segundo parámetro.
File(File ruta, String nombre)	Recibe en la instanciación del objeto un objeto de tipo File, que hace referencia a un directorio, como primer parámetro y el nombre del fichero como segundo parámetro.

4.1.2 Algunos métodos de File

Para manejo de ficheros y directorios

Devuelve	Método	Explicación
boolean	**canRead()**	Informa si se puede leer la información que contiene.
boolean	**canWrite()**	Informa si se puede guardar información.
boolean	**exists()**	Informa si el fichero o el directorio existen.
boolean	**isFile()**	Informa si es un archivo.
boolean	**isDirectory()**	Devuelve true si el objeto File corresponde a un directorio.
long	**lastModified()**	Retorna la fecha de la última modificación.
String	**getName()**	Devuelve el nombre del fichero o directorio.
String	**getPath()**	Devuelve la ruta relativa.
String	**getAbsolutePath()**	Devuelve la ruta absoluta.
String	**getParent()**	Devuelve el nombre del directorio padre o null si no existe.
String[]	**list()**	Devuelve un array de Strings que contiene los nombres de los archivos y directorios que contiene el directorio.
File[]	**listFiles()**	Devuelve un array que contiene referencias a los archivos que contiene el directorio.

Sólo para manejo de ficheros

Devuelve	Método	Explicación
void	**delete()**	Borra el fichero.
long	**length()**	Devuelve el tamaño del archivo en bytes.
boolean	**renameTo(File)**	Cambia el nombre del fichero por el nombre del archivo pasado como argumento.

Sólo para manejo de directorios

Devuelve	Método	Explicación
boolean	**mkdir()**	Crea el directorio.
String[]	**list()**	Devuelve un listado de los archivos/directorios que se encuentran en el directorio.
File[]	**listFiles()**	Devuelve un array que contiene referencias a los archivos que se encuentran en el directorio.

Hay que tener en cuenta que se puede:

- indicar el nombre de un fichero sin la ruta: se buscará el fichero en el directorio actual.
- indicar el nombre de un fichero con la ruta relativa.
- indicar el nombre de un fichero con la ruta absoluta.

Ejemplo:

```
import java.io.File;
public class EjemClaseFile {
    public static void main(String[] args) {
        File fichero;
        String resp, nombre;
        resp = Leer.pedirCadena("\n¿Nombre de fichero para ver si existe?\n "
                + "Escribe 'S ' para si- y cualquier otro caracter para no\t");
        while (resp.equalsIgnoreCase("S")) {
            nombre = Leer.pedirCadena("\n\tIndica el nombre de fichero a buscar: ");
            fichero = new File(nombre);
            if (fichero.isFile()) {
                System.out.println("\t\t El fichero existe");
            } else {
                System.out.println("\t\t El fichero no existe");
            }
            resp = Leer.pedirCadena("\nNombre de fichero para ver si existe?\n "
                    + "Escribe 'S ' para si- y cualquier otro caracter para no\t");
        }
    }
}
```

5 ESCRITURA Y LECTURA DE INFORMACIÓN EN FICHEROS

Para trabajar con flujos de bytes se usan las clases:

- FileInputStream
- FileOutputStream

Para trabajar con flujos de caracteres

- FileReader
- FileWriter

5.1 FileOutputStream

Al instanciar un objeto de esta clase abre el fichero en modo escritura. Si no existe, se crea en ese momento. Ya se puede guardar información byte a byte de modo secuencial.

Constructores:

Constructor	Explicación
FileOutputStream(String nombreFich)	Recibe como parámetro el nombre del fichero a abrir y borra el contenido previo del fichero comenzando a escribir desde el principio.
FileOutputStream(File fichero)	Recibe como parámetro un objeto File que representa al fichero con el que queremos trabajar y borra el contenido previo del fichero comenzando a escribir desde el principio.
FileOutputStream(String nombreDeFich, boolean append)	Recibe como parámetro el nombre del fichero a abrir y, si **append** es **true**, se sitúa al final del fichero para añadir contenido desde el final.
FileOutputStream(File fichero, boolean append)	Recibe como parámetro un objeto File que representa al fichero con el que queremos trabajar y, si **append** es **true**, se sitúa al final del fichero para añadir contenido desde el final.

Métodos más usados:

Devuelve	Método	Explicación
int	**write(int byte)**	Escribe el byte que recibe como argumento en el fichero.
int	**write(byte cadByte[])**	Escribe todos los bytes que contiene la tabla cadByte.
void	**close()**	Cierra el fichero.

5.2 FileInputStream

Al instanciar un objeto de esta clase abre el fichero en modo lectura. Ya se puede leer byte a byte de modo secuencial.

Constructores:

Constructor	Explicación	Excepción que lanza
FileIntputStream(String nombreFich)	Recibe como parámetro el nombre del fichero a abrir.	FileNotFoundException si el fichero no existe.
FileIntputStream(File fichero)	Recibe como parámetro un objeto File que representa al fichero con el que queremos trabajar.	FileNotFoundException si el fichero no existe.

Métodos más usados:

Devuelve	Método	Explicación
int	**read()**	Devuelve el código ASCII del siguiente byte que hay después de donde está situado el puntero del fichero. Dicho puntero se va moviendo secuencialmente por el fichero, según vamos leyendo los bytes. Devuelve **-1** si no hay ningún byte más que leer.
int	**read(byte cadByte[])**	Lee hasta cadByte.length bytes guardándolos en la tabla que se envía como parámetro. Devuelve -1 si no hay ningún byte más que leer.
void	**close()**	Cierra el fichero.

5.3 DataInputStream y DataOutputStream

Con las clases **FileInputStream** y **FileOutputStream** cada byte que lee o escribe del fichero lo va cogiendo o enviando al fichero de uno en uno, lo cual hace que dicha tarea sea lenta.

DataInputStream y DataOutputStream poseen más métodos, como la posibilidad de leer y escribir datos de una sola vez, como short, int, String, etc. Sin necesidad de tener que ir byte a byte.

Constructor	Explicación
DataOutputStream(OutputStream out)	Crea un nuevo flujo de salida de datos para escribir datos en el OutputStream indicado “out”.
DataInputStream(InputStream in)	Crea un nuevo flujo de entrada de datos para leer datos en el InputStream indicado “in”.

Para abrir un fichero con estas clases:

```
FileOutputStream fichero = new FileOutputStream("fich.dat");
DataOutputStream salida = new DataOutputStream(fichero);


FileInputStream fichero = new FileInputStream("fich.dat");
DataInputStream entrada = new DataInputStream(fichero);
```

Métodos más usados:

DataInputStream	DataOutputStream
• readShort() • readInt() • readFloat() • readUTF() • readLine() • …	• writeShort(short dato) • writeInt(int dato) • writeUTF(String dato) • …

Al leer un fichero hay que utilizar el mismo formato con el que se ha escrito porque de no ser así daría errores en la ejecución al no corresponderse los tipos.

Para saber que se ha alcanzado el final del fichero, los métodos lanzan la excepción **EOFException**, así que hay que **recogerla** y **tratarla** correctamente.

5.4 FileWriter

Al instanciar un objeto de esta clase abre el fichero en modo escritura. Si no existe, se crea en ese momento. Ya se puede guardar información carácter a carácter de modo secuencial.

Constructores:

Constructor	Explicación
FileWriter(String nombreFich)	Recibe como parámetro el nombre del fichero a abrir y borra el contenido previo del fichero comenzando a escribir desde el principio.
FileWriter(File fichero)	Recibe como parámetro un objeto File que representa al fichero con el que queremos trabajar y borra el contenido previo del fichero comenzando a escribir desde el principio.
FileWriter(String nombreDeFich, boolean append)	Recibe como parámetro el nombre del fichero a abrir y, si **append** es **true**, se sitúa al final del fichero para añadir contenido desde el final.
FileWriter(File fichero, boolean append)	Recibe como parámetro un objeto File que representa al fichero con el que queremos trabajar y, si **append** es **true**, se sitúa al final del fichero para añadir contenido desde el final.

Métodos más usados:

Devuelve	Método	Explicación
void	**write(int c)**	Escribe un carácter.
void	**write(char[] buf)**	Escribe un array de caracteres.
void	**write(String str)**	Escribe una cadena de caracteres.
void	**append(char c)**	Añade un carácter.

5.5 FileReader

Al instanciar un objeto de esta clase abre el fichero en modo lectura. Ya se puede leer carácter a carácter de modo secuencial.

Constructores:

Constructor	Explicación
FileReader(File fichero)	Lanza la excepción FileNotFoundException si el fichero del parámetro no existe.
FileReader(String fichero)	Lanza la excepción FileNotFoundException si el fichero del parámetro no existe.

Métodos más usados:

Devuelve	Método	Explicación
int	**read()**	Lee un carácter y lo devuelve.
nt	**iread(char[] buf)**	Lee hasta buf.length caracteres de datos de la matriz buf pasada como parámetro.
int	**read(char[] buf, int desplazamiento, int n)**	Lee hasta n caracteres de datos de una matriz buf comenzando por buf[desplazamiento] y devuelve el número leído de caracteres.

6 BufferedReader, BufferedInputStream, BufferedWriter y BufferedOutputStream

Con las clases **FileWriter** y **FileReader**, cada carácter que lee o escribe del fichero lo va leyendo o enviando al fichero de uno en uno, lo que hace que dicha tarea sea lenta.

Gracias al uso de clases como **BufferedReader**, **BufferedInputStream**, **BufferedWriter** y **BufferedOutputStream** el proceso se hace más rápido. Esto se debe al uso de un buffer intermedio, lo cual hace que no vaya carácter a carácter o byte a byte.

7 Acceso aleatorio: RandomAccessFile

Esta clase implementa las interfaces **DataInput** y **DataOutput**. Con lo cual, puede hacer uso de los métodos `read()` y `write()` para cada tipo de dato.

Constructores:

Constructor	Explicación
RandomAccessFile(String nomFich, String oper)	**nomFich** es el nombre del fichero y **oper** es el argumento que determina si el contenido del fichero se va a poder solo leer (**r**) o leer y escribir (**rw**).
RandomAccessFile(File nomFich, String oper)	**nomFich** es el objeto fichero y **oper** es el argumento que determina si el contenido del fichero se va a poder solo leer (**r**) o leer y escribir (**rw**).

Métodos más usados:

Devuelve	Método	Explicación
long	**getFilePointer()**	Indica dónde está situado el puntero del fichero.
void	**seek(long pos)**	Desplaza el puntero 'pos' bytes desde el inicio.
long	**length()**	Devuelve la longitud del fichero.
int	**skipbytes(int d)**	Desplaza el puntero del fichero –desde su posición actual- tantos bytes como indica 'd'.
<T>	**readBoolean(), readByte(), readChar(), reading(), readDouble(), readFloat(), readUTF(), readLine ()**	Leen un dato del tipo indicado.

Devuelve	Método	Explicación
<T>	**writeBoolean, writeByte, writeBytes, writeChar, writeChars, writeInt, writeDouble, writeFloat, writeUTF, writeLine**	Todos reciben como parámetro, el dato a escribir.

8 Almacenamiento de objetos en ficheros

Serialización

Si queremos enviar un objeto a través de un flujo de datos, deberemos convertirlo en una serie de *bytes*. Esto es lo que se conoce como serialización de objetos, que nos permitirá leer y escribir objetos directamente.

Para leer o escribir objetos podemos utilizar los objetos `ObjectInputStream` y `ObjectOutputStream` que incorporan los métodos `readObject` y `writeObject` respectivamente. Los objetos que escribamos en dicho flujo deben tener la capacidad de ser *serializables*.

Interface Serializable

Serán *serializables* aquellos objetos que implementan la interfaz Serializable. Cuando queramos hacer que una clase definida por nosotros sea *serializable* deberemos implementar dicha interface, que no define ninguna función, sólo se utiliza para identificar las clases que son *serializables*. Para que nuestra clase pueda ser *serializable*, todas sus propiedades deberán ser de tipos de datos básicos o bien objetos que también sean *serializables*.

Por ejemplo, si tenemos un objeto de la siguiente clase:

```
import java.io.Serializable;
public class Punto2D implements Serializable {
	private int x;
	private int y;

	public Punto2D(int x, int y) {
		this.x = x;
		this.y = y;
	}
	public int getX() {
		return x;
	}
	public void setX(int x) {
		this.x = x;
	}
	public int getY() {
		return y;
	}
	public void setY(int y) {
		this.y = y;
	}
	public String toString(){
		return "x: " + x+ " , y: " + y;
	}
}
```

Podríamos enviarlo a través de un flujo, independientemente de su destino, de la siguiente forma:

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

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

public class Principal {

   public static void main(String[] args) {
      Punto2D p = new Punto2D(1,2);
      ObjectOutputStream fichEscribir=null;
      try{  // escribimos el objeto en el fichero
            FileOutputStream fos = new FileOutputStream("ficheroPuntos.dat");
            fichEscribir = new ObjectOutputStream(fos);
            fichEscribir.writeObject(p);
      }catch(FileNotFoundException e){
            System.out.println("Error. no se encuentra el fichero");
      }
      catch(IOException e){
            System.out.println("Error E/S: " + e.getMessage());
      } finally {
            try {
                  if (fichEscribir != null) {
                        fichEscribir.close();
                  }
            } catch (IOException ex) {
                  System.out.println("Error al cerrar fichero: " + ex.getMessage());
            }
      }//finally

      ObjectInputStream fichLeer = null;
      try { // leemos el fichero de objetos
         boolean hayaDatos = true;
         Punto2D punto;
         fichLeer = new ObjectInputStream(new FileInputStream("ficheroPuntos.dat"));
         System.out.println("\nLos datos de los alumnos recuperados del fichero son: ");
         while (hayaDatos) {
             try {
                   punto = (Punto2D) fichLeer.readObject();
                 System.out.println(punto);
             } catch (EOFException e) {
                 hayaDatos = false; //Salimos del bucle al terminar de recorrer el fichero
             }
         }
         System.out.println("Fichero recorrido completamente");
      } catch (ClassNotFoundException e) {
         System.out.println(e.getMessage());
      } catch (IOException e) {
         System.out.println(e.getMessage());
      } finally {
         try {
             fichLeer.close();
         } catch (IOException e) {
```

```
            System.out.println(e.getMessage());
        }
     }
   }// main
}//class
```

En este caso hemos utilizado como canal de datos un flujo con destino a un fichero, pero se podría haber utilizado cualquier otro tipo de canal (por ejemplo para enviar un objeto Java desde un servidor web hasta una máquina cliente). En aplicaciones distribuidas los objetos *serializables* nos permitirán mover estructuras de datos entre diferentes máquinas sin que el desarrollador tenga que preocuparse de la codificación y transmisión de los datos.

Muchas clases de la API de Java son *serializables*, como por ejemplo las colecciones. Si tenemos una serie de elementos en una lista, podríamos serializar la lista completa, y de esa forma guardar todos nuestros objetos, con una única llamada a writeObject.

Cuando una clase implemente la interfaz Serializable veremos que Eclipse nos da un *warning* si no añadimos un campo serialVersionUID. Este es un código numérico que se utiliza para asegurarnos de que al recuperar un objeto serializado éste se asocie a la misma clase con la que se creó. Así evitamos el problema que puede surgir al tener dos clases que puedan tener el mismo nombre, pero que no sean iguales (podría darse el caso que una de ellas esté en una máquina cliente, y la otra en el servidor). Si no tuviésemos ningún código para identificarlas, se podría intentar recuperar un objeto en una clase incorrecta.

Eclipse nos ofrece dos formas de generar este código pulsando sobre el icono del *warning*: con un valor por defecto, o con un valor generado automáticamente. Será recomendable utilizar esta segunda forma, que nos asegura que dos clases distintas tendrán códigos distintos.