FAMILIA PROFESIONAL:	**Informática y Comunicaciones**
CICLOS FORMATIVO:	**Desarrollo de Aplicaciones Multiplataforma**
MÓDULO:	**Acceso a Datos**

UNIDAD 1: FICHEROS SECUENCIALES Y ALEATORIOS

CONTENIDOS

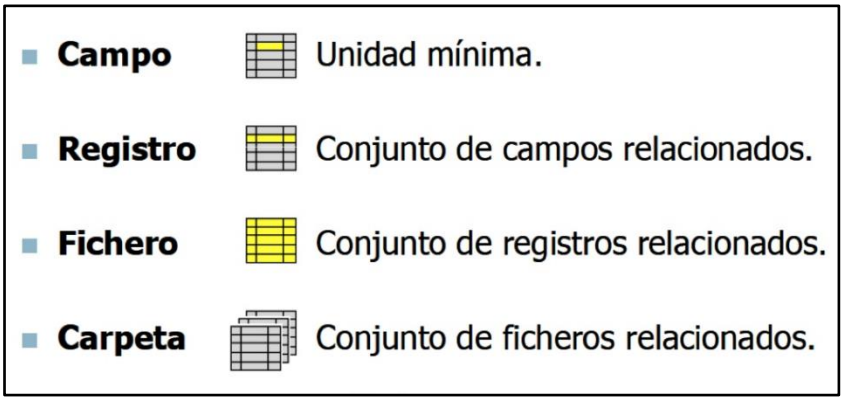
AUTORES: **Fernando Rodríguez Alonso**
Sonia Pasamar Franco

ÍNDICE DE CONTENIDOS

1. CLASIFICACIÓN DE FICHEROS

Un **fichero** o archivo es una agrupación de una serie de datos, todos del mismo tipo o con la misma estructura, almacenados en un soporte magnético para poder manipularlos en cualquier momento. Los ficheros tienen un nombre y se ubican en directorios o carpetas. Su principal ventaja es que los datos que se guardan en ellos no son volátiles.

Cada información contenida en un fichero se denomina registro. Un **registro** está formado, a su vez, por uno o varios datos que pueden ser de diferentes tipos (booleanos, números enteros o decimales, cadenas de caracteres, etc.). Cada dato incluido en un registro recibe el nombre de **campo**.



Según el **tipo de contenido**, los ficheros se clasifican:

- **Ficheros de Caracteres (o de Texto).** Son aquellos creados exclusivamente con caracteres (códigos ASCII y Unicode), por lo que pueden ser creados y visualizados por cualquier editor de texto.
- **Ficheros Binarios (o de Bytes).** Son aquellos que no contienen caracteres reconocibles, sino que los bytes que contienen representan otra información como imágenes, música o vídeo. Estos ficheros solo pueden ser abiertos por aplicaciones concretas que conozcan cómo están organizados los bytes dentro del fichero.

Según el **modo de acceso**, los ficheros se clasifican:

- **Ficheros Secuenciales.** Los registros de un fichero se crean en posiciones físicas contiguas en el soporte de almacenamiento, y se leen y se escriben en orden. Es decir, si se desea acceder a un registro concreto que está hacia la mitad del fichero, es necesario leer antes todos los anteriores. La escritura de registros se realiza a partir del último registro escrito en el fichero y no se pueden realizar inserciones entre los registros almacenados en el fichero.
- **Ficheros Directos o Aleatorios.** Permiten acceder directamente a un registro concreto del fichero, sin necesidad de leer los registros anteriores, utilizando una dirección lógica del registro. Para poder realizar esto, todos los registros almacenados en el fichero tienen un tamaño fijo y conocido. Así, se puede ir de un registro a otro de forma aleatoria para leerlos o modificarlos.
- **Ficheros Indexados o Indizados.** Aprovechan lo mejor del acceso secuencial y del acceso aleatorio. Tienen fundamentalmente dos partes:
  1) *Área de Índices.* Permite el acceso directo a los registros de datos del fichero. Este índice puede estar organizado de forma secuencial, multinivel o de árbol. Cada elemento del índice es un par formado por la clave de registro y la dirección del registro dentro del área de datos.
  2) *Área Primaria o de Datos.* Contiene los registros del fichero organizados de manera secuencial.

2. OPERACIONES DE GESTIÓN DE FICHEROS

Las operaciones básicas que se realizan sobre cualquier fichero, independientemente de la forma de acceso al mismo, son las siguientes:

- **Creación del Fichero.** El fichero se crea en el disco con un nombre que después se deberá usar para acceder a él. Este proceso solo se realiza una vez.
- **Apertura del Fichero.** Para que un programa pueda operar con un fichero, primero deberá realizar su apertura. Para ello, el programa utilizará algún método para identificar el fichero, como usar un descriptor del fichero.
- **Cierre del Fichero.** El fichero se deberá cerrar cuando el programa no lo vaya a utilizar. Suele ser la última instrucción del programa.
- **Lectura de Datos del Fichero.** Este proceso consiste en transferir información del fichero a la memoria, a través de variables del programa en las que se guardan los datos extraídos del fichero.
- **Escritura de Datos en el Fichero**. Este proceso consiste en transferir información de la memoria, por medio de las variables del programa, al fichero.

Una vez abierto, las operaciones típicas que se realizan sobre un fichero son:

- **Alta.** Consiste en añadir un nuevo registro al fichero.
- **Baja.** Consiste en eliminar del fichero un registro ya existente. La eliminación puede ser lógica (cambiando el valor de algún campo del registro que se usa para controlar esa situación), o física (eliminando físicamente el registro del fichero). A menudo, el borrado físico consiste en reescribir de nuevo el fichero en otro fichero sin los datos que se desean eliminar y renombrarlo al fichero original.
- **Modificación.** Consiste en cambiar o modificar parte del contenido de un registro. Antes de la modificación, será necesario localizar el registro a modificar dentro del fichero. Una vez localizado, se realizan los cambios y se reescribe el registro.
- **Consulta.** Consiste en buscar en el fichero un registro determinado.

2.1. OPERACIONES SOBRE FICHEROS SECUENCIALES

Las operaciones indicadas previamente se realizan de la siguiente forma en un fichero secuencial:

- **Alta.** Se realiza al final del último registro insertado, es decir, solo se permite añadir datos al final del fichero.
- **Baja.** Para eliminar un registro de un fichero, es necesario leer todos los registros uno a uno y escribirlos en un fichero auxiliar, excepto el que se desea borrar. A continuación, se borra el fichero inicial y se renombra el fichero auxiliar con el nombre del fichero original.
- **Modificación.** Consiste en localizar el registro a modificar, efectuar las modificaciones de campos pertinentes y reescribir el fichero inicial en otro fichero auxiliar incluyendo el registro modificado.
- **Consulta.** Para buscar un determinado registro, es necesario leer secuencialmente todos los registros del fichero desde el primer registro hasta localizar el registro deseado.

Los ficheros secuenciales tienen las **ventajas** de una rápida capacidad de acceso al siguiente registro (son rápidos cuando se accede a los registros de forma secuencial) y de un mejor aprovechamiento de la utilización del espacio.

Sus principales **inconvenientes** son que no se puede acceder directamente a un registro determinado (no soporta el acceso aleatorio) y que el proceso de actualización de un registro requiere una reescritura total del fichero.

2.2. OPERACIONES SOBRE FICHEROS ALETORIOS

Los ficheros aleatorios manipulan posiciones o direcciones relativas para acceder a los registros. Para localizar un registro y posicionarse sobre él, se aplica una **función de conversión**, que usualmente está relacionada con el tamaño del registro y con la clave del mismo.

Al aplicar una función de conversión a un campo clave, puede suceder que la posición devuelta esté ocupada por otro registro. En este caso, habría que buscar una nueva posición libre en el fichero para ubicar el registro o utilizar una **zona de excedentes** dentro del mismo para ubicar este registro que ha provocado la colisión.

Las operaciones indicadas previamente se realizan de la siguiente forma en un fichero aleatorio:

- **Alta.** Para insertar un registro se necesita conocer su clave, aplicar la función de conversión a la clave para obtener la dirección y escribir el registro en la posición devuelta. Si la posición está ocupada por otro registro, se insertará en la zona de excedentes.
- **Baja.** La eliminación de un registro se suele realizar de forma lógica, es decir, se utiliza un campo del registro (un booleano) para indicar si existe o no el registro. Físicamente el registro no desaparece del fichero. Se localiza el registro a borrar a partir de su campo clave y se reescriben los demás campos del registro con el valor 0.
- **Modificación.** Para actualizar un registro hay que localizarlo antes. A partir de su campo clave, se aplica la función de conversión para obtener la dirección, se modifican los campos deseados del registro y se reescribe el registro en esa posición.
- **Consulta.** Para buscar un determinado registro, se necesita conocer su campo clave, aplicar la función de conversión a la clave para obtener la dirección y leer el registro ubicado en esa dirección. Hay que comprobar si el registro buscado se encuentra en esa posición. Si no está, se deberá buscar dicho registro en la zona de excedentes.

Los ficheros aleatorios tienen la **ventaja** de un acceso rápido a una posición determinada para leer o escribir registros.

Sus principales **inconvenientes** son el establecimiento de la relación entre la posición que ocupa un registro y su contenido (ya que, al aplicar la función de conversión, a veces se obtienen posiciones ocupadas y hay que recurrir a la zona de excedentes) y que parte del espacio destinado al fichero puede quedar desaprovechado, pues pueden aparecer huecos (posiciones no ocupadas) entre los registros.

2.3. CLASE ASOCIADA CON FICHEROS

En Java, la clase **File** proporciona un conjunto de utilidades relacionadas con ficheros que proporcionan información (nombre, atributos, directorios, etc.) acerca de los mismos. Puede representar el nombre de un fichero particular o los nombres de un conjunto de ficheros de un directorio. También se puede usar para crear un nuevo directorio o una trayectoria de directorios completa si ésta no existe.

Sin embargo, esta clase no permite acceder a la información que contiene el propio fichero.

Para crear un objeto *File*, se puede usar cualquiera de los siguientes constructores:

CONSTRUCTOR	EXPLICACIÓN
`File(String directorioYFichero)`	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. directorioYFichero puede ser también la ruta a un directorio, sin indicar al final el nombre de ningún fichero.
`File(String directorio, String nombreFichero)`	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 directorio,`	Recibe en la instanciación del objeto un objeto de tipo

`String fichero)`	File, que hace referencia a un directorio, como primer parámetro, y el nombre del fichero como segundo parámetro.

La clase *File* tiene los siguientes métodos que sirven para ficheros y directorios:

MÉTODO	EXPLICACIÓN
`boolean canRead()`	Devuelve true si el fichero se puede leer.
`boolean canWrite()`	Devuelve true si el fichero se puede escribir.
`boolean exists()`	Devuelve true si el fichero/directorio existe.
`boolean isFile()`	Devuelve true si el objeto File corresponde a un fichero normal.
`boolean isDirectory()`	Devuelve true si el objeto File corresponde a un directorio.
`long lastModified()`	Devuelve la fecha de la última modificación.
`String getName()`	Devuelve el nombre del fichero/directorio.
`String getPath()`	Devuelve la ruta relativa.
`String getAbsolutePath()`	Devuelve la ruta absoluta del fichero/directorio.
`String getParent()`	Devuelve el nombre del directorio padre o null si no existe.

La clase *File* dispone de los siguientes métodos para el manejo de ficheros:

MÉTODO	EXPLICACIÓN
`boolean createNewFile()`	Crea un nuevo fichero vacío asociado al objeto File si y solo si no existe un fichero con dicho nombre.
`boolean delete()`	Borra el fichero/directorio asociado al objeto File.
`long length()`	Devuelve el tamaño del fichero en bytes.
`boolean renameTo(File nuevoNombre)`	Renombra el fichero representado por el objeto File asignándole nuevoNombre.

La clase *File* dispone de los siguientes métodos para el manejo de directorios:

MÉTODO	EXPLICACIÓN
`boolean mkdir()`	Crea un directorio con el nombre indicado en la instanciación del objeto File. Solo se crea si no existe.
`String[] list()`	Devuelve un vector de cadenas de caracteres con los nombres de ficheros y directorios asociados al objeto File.
`File[] listFiles()`	Devuelve un vector de objetos File conteniendo los ficheros que estén dentro del directorio representado por el objeto File.

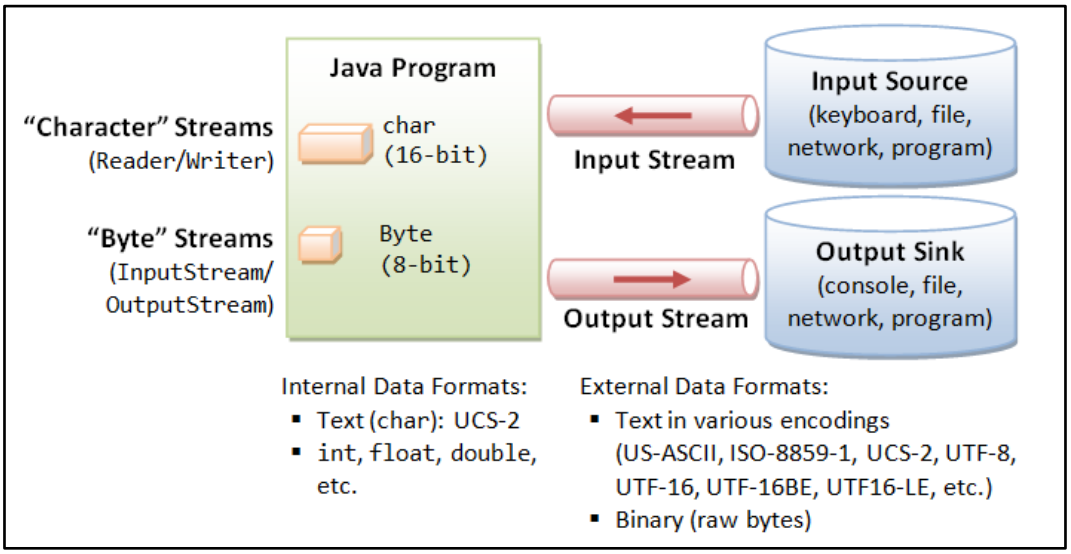
Hay que tener en cuenta que se puede:

- 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**.

3. ACCESO SECUENCIAL

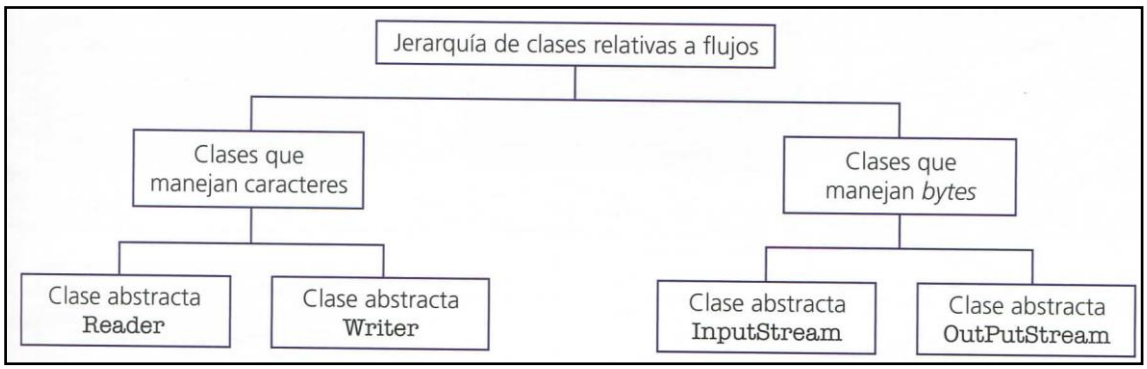
El sistema de entrada/salida en Java presenta una gran variedad de clases que se implementan en el paquete **java.io**. Utiliza la abstracción del flujo, corriente de datos o **stream** para tratar la comuni-cación de información entre una fuente y un destino. Dicha información puede estar en un fichero del disco duro, en la memoria, en algún lugar de la red, e incluso en otro programa. Por tanto, las operaciones que un programa realiza sobre un flujo son independientes del dispositivo al que esté asociado.

Así pues, un **archivo** es simplemente un flujo externo, es decir, una secuencia de bytes almacenados en un dispositivo externo. La vinculación de un flujo con una fuente o destino de datos la hace el sistema de entrada/salida de Java y los programas leen o escriben de forma secuencial en el flujo, que puede estar conectado a un dispositivo o a otro.



Hay dos tipos de flujos de datos definidos:

- **Flujo de Caracteres (16 bits).** Realiza operaciones de entrada y salida de caracteres. Todas las clases de flujo de caracteres descienden de las clases **Reader** y **Writer**. Esta jerarquía tiene su origen en la internacionalización de conjuntos de caracteres (Unicode).
- **Flujo de Bytes (8 bits).** Realiza operaciones de entrada y salida de bytes y su uso está orientado a la lectura y escritura de datos binarios. Todas las clases de flujos de bytes descienden de las clases **InputStream** y **OutputStream**.



En Java, la entrada desde el teclado y la salida por la 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*:

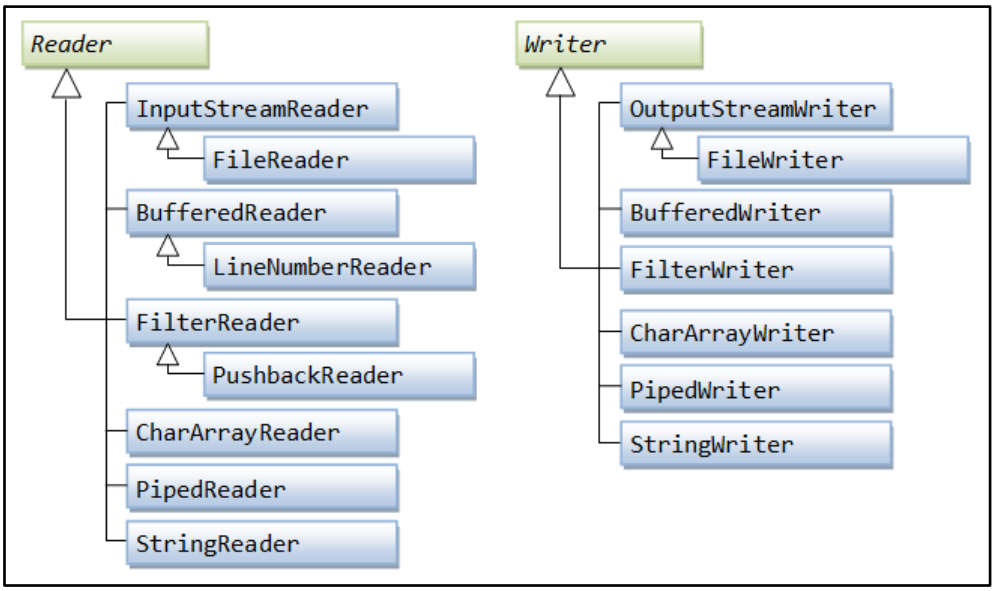
- **System.in.** Hace referencia a la entrada estándar de datos (teclado).
- **System.out.** Hace referencia a la salida estándar de datos (pantalla).
- **System.err.** Hace referencia a la salida estándar de información de errores (pantalla).

3.1. FLUJOS DE CARACTERES

Las clases abstractas **Reader** y **Writer** manejan flujos de caracteres Unicode. Hay ocasiones en las que hay que usar las clases que manejan bytes en combinación con las clases que manejan caracteres. Para lograr esto, existen varias clases "puente" (que realizan transformaciones entre flujos de bytes y flujos de caracteres):

- **InputStreamReader** convierte un *InputStream* en un *Reader* (lee bytes y los convierte a caracteres).
- **OutputStreamReader** convierte un *OutputStream* en un *Writer* (lee caracteres y los convierte a bytes).

La siguiente figura muestra la jerarquía de clases para la lectura y la escritura de flujos de caracteres.



Las clases de flujos de caracteres más importantes son:

- **FileReader / FileWriter.** Se utilizan para el acceso a ficheros de texto. Estas clases leen y escriben caracteres en ficheros.
- **CharArrayReader / CharArrayWriter.** Se utilizan para el acceso a caracteres. Estas clases leen y escriben un flujo de caracteres en un vector de caracteres.
- **BufferedReader / BufferedWriter.** Se utilizan para la buferización de datos. Estas clases usan un buffer intermedio entre la memoria y el flujo de datos, para evitar que cada lectura o escritura acceda directamente al fichero.

3.1.1. Las Clases FileReader y FileWriter

Los ficheros de texto, que normalmente se generan con un editor de textos, almacenan caracteres alfanuméricos en un formato estándar (ASCII, Unicode, UTF-8, etc.). Para trabajar con ellos en Java, se utilizan la clase **FileReader** para leer los caracteres del fichero y la clase **FileWriter** para escribir los caracteres en el fichero. Además, las lecturas o escrituras realizadas sobre un fichero deben codificarse dentro de un manejador de excepciones try-catch.

Al instanciar un objeto con la clase **FileReader**, el programa abre el fichero que se envía como argumento para lectura y su información se puede leer de forma secuencial carácter a carácter. Si el fichero no existe o no es válido, se lanzará la excepción *FileNotFoundException*.

La clase *FileReader* tiene los siguientes constructores:

CONSTRUCTOR	EXPLICACIÓN
`FileReader(File fichero)`	Crea un flujo de caracteres de lectura. Lanza la excepción FileNotFoundException si el fichero pasado no existe.
`FileReader(String fichero)`	Crea un flujo de caracteres de lectura. Lanza la excepción FileNotFoundException si el fichero pasado no existe.

Los principales métodos de la clase *FileReader* son:

MÉTODO	EXPLICACIÓN
`int read()`	Lee un carácter y lo devuelve.
`int read(char[] buffer)`	Lee hasta buffer.length caracteres de datos y los almacena en el vector de caracteres buffer. Devuelve el número de carac-teres leídos.
`int read(char[] buffer, int desplazamiento, int max)`	Lee hasta max caracteres de datos y los almacena en el vector buffer comenzando por buffer[desplazamiento]. Devuelve el número de caracteres leídos.

Estos métodos de lectura devuelven el número de caracteres leídos o -1 si se ha alcanzado el final del fichero.

Al instanciar un objeto con la clase **FileWriter**, el programa abre el fichero especificado con el fin de guardar información, pero carácter a carácter. Si el fichero no existe, el programa lo crea de forma automática. Si el disco está lleno o protegido contra escritura, se lanzará la excepción *IOException*.

La clase *FileWriter* tiene los siguientes 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 se desea trabajar y borra el contenido previo del fichero comenzando a escribir desde el principio.
`FileWriter(String nombreFich, 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 se desea trabajar y, si append es true, se sitúa al final del fichero para añadir contenido desde el final.

Los principales métodos de la clase *FileWriter* son:

MÉTODO	EXPLICACIÓN
`void write(int c)`	Escribe un carácter.
`void write(char[] buffer)`	Escribe un vector de caracteres.
`void write(char[] buffer, int desplazamiento, int max)`	Escribe max caracteres de datos del vector de caracteres buffer comenzando por buffer[desplazamiento].
`void write(String str)`	Escribe una cadena de caracteres.
`Writer append(char c)`	Añade un carácter al final de un fichero.

3.1.2. Las Clases BufferedReader y BufferedWriter

Las clases **BufferedReader** y **BufferedWriter** se pueden utilizar sobre las clases *FileReader* y *FileWriter* u otros flujos de caracteres para realizar operaciones de entrada/salida con un buffer intermedio, en lugar de carácter a carácter.

Para construir un objeto **BufferedReader**, se necesita tener instanciado previamente un objeto *FileReader*:

```
FileReader fr = new FileReader(nombreFichero);
BufferedReader br = new BufferedReader(fr);
```

La clase *BufferedReader* dispone de los siguientes métodos:

MÉTODO	EXPLICACIÓN
`String readLine()`	Lee una línea de caracteres del flujo y la devuelve. Devuelve null si no hay nada que leer o se llega al final del fichero.
`void close()`	Libera los recursos del sistema asociados al flujo y lo cierra.

Para construir un objeto **BufferedWriter**, se necesita tener instanciado previamente un objeto *FileWriter*:

```
FileWriter fw = new FileWriter(nombreFichero);
BufferedWriter bw = new BufferedWriter(fw);
```

La clase *BufferedWriter* dispone de los siguientes métodos:

MÉTODO	EXPLICACIÓN
`void write(String str)`	Escribe una línea de caracteres en el flujo.
`void newLine()`	Escribe un salto o separador de línea en el flujo.
`void close()`	Vacía el flujo y lo cierra.

3.2. FLUJOS DE BYTES

Los ficheros binarios almacenan secuencias de dígitos binarios (bytes), que no son legibles directamente por el usuario, y tienen la ventaja de que ocupan menos espacio en disco.

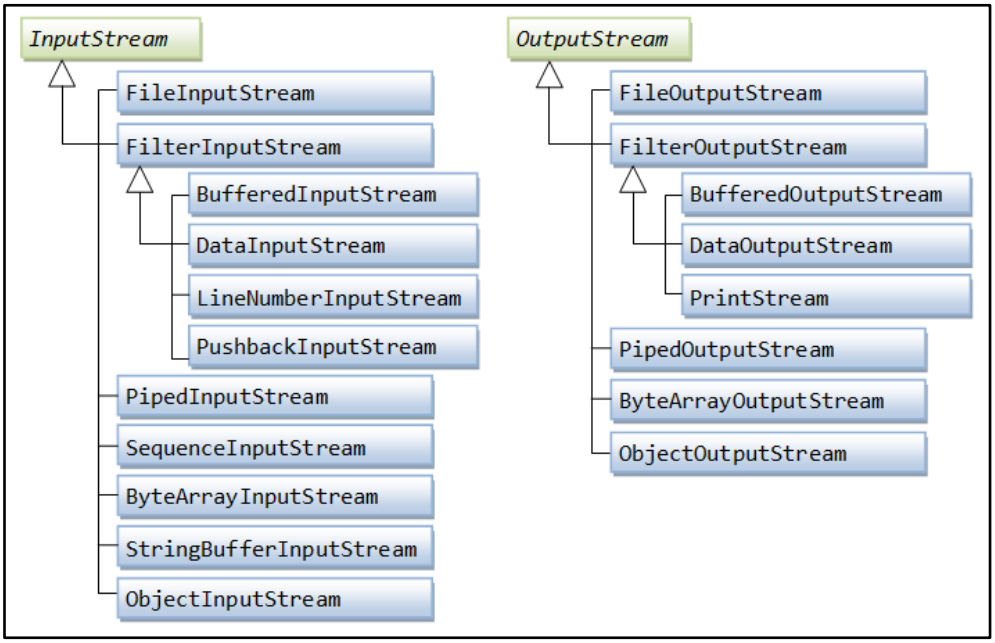
Las clases abstractas **InputStream** y **OutputStream** manejan flujos de bytes. La siguiente figura muestra la jerarquía de clases para la lectura y la escritura de flujos de bytes.

La clase abstracta **InputStream** representa un flujo de bytes asociado a una fuente de datos, como puede ser un vector de bytes, un objeto String, un fichero, una tubería (los elementos entran por un extremo y salen por el otro), una secuencia de otros flujos, otras fuentes como una conexión a internet, etc.

La clase abstracta **OutputStream** representa un flujo de bytes asociado a un destino de datos, como puede ser un vector de bytes, un fichero o una tubería.

Las clases de flujos de bytes más importantes son:

- **FileInputStream / FileOutputStream.** Se utilizan para el acceso a ficheros binarios. Estas clases leen y escriben bytes en ficheros.
- **DataInputStream / DataOutputStream.** Permiten leer y escribir datos de tipos primitivos (*boolean*, *int*, *short*, *long*, *float*, *double*, *char*) en el flujo.
- **ObjectInputStream / ObjectOutputStream.** Permiten leer y escribir objetos serializables (correspondientes a clases definidas por el programador) en el flujo.



3.2.1. Las Clases FileInputStream y FileOutputStream

Los ficheros binarios almacenan secuencias de dígitos binarios que no son legibles directamente por el usuario. Tienen la ventaja de que ocupan menos espacio en disco. En Java, las dos clases que permiten trabajar con ficheros son **FileInputStream** (para entrada) y **FileOutputStream** (para salida), que operan con flujos de bytes y crean un enlace entre el flujo de bytes y el fichero.

Cuando se instancia un objeto con la clase **FileInputStream**, el programa abre el fichero (que se debe enviar como argumento del constructor) en modo lectura. Una vez abierto, se podrá leer la información que contiene de forma secuencial byte a byte. Si el fichero no existe, se lanzará la excepción *FileNotFoundException*.

La clase *FileInputStream* tiene los siguientes constructores:

CONSTRUCTOR	EXPLICACIÓN
`FileInputStream(String nombreFich)`	Crea un flujo de bytes de lectura. Lanza la excepción FileNotFoundException si el fichero pasado no existe.
`FileInputStream(File fichero)`	Crea un flujo de bytes de lectura. Lanza la excepción FileNotFoundException si el fichero pasado no existe.

Los principales métodos de la clase *FileInputStream* son:

MÉTODO	EXPLICACIÓN
`int read()`	Lee un byte y lo devuelve.
`int read(byte[] buffer)`	Lee hasta buffer.length bytes de datos y los almacena en el vector de caracteres buffer. Devuelve el número de bytes leídos.
`int read(byte[] buffer, int desplazamiento, int max)`	Lee hasta max bytes de datos y los almacena en el vector buffer comenzando por buffer[desplazamiento]. Devuelve el número de bytes leídos.
`void close()`	Libera los recursos del sistema asociados al flujo y lo cierra.

Estos métodos de lectura devuelven el número de bytes leídos o -1 si se ha alcanzado el final del fichero.

Cuando se instancia un objeto con la clase **FileOutputStream**, el programa abre el fichero para escritura. Una vez abierto, se podrá guardar información de forma secuencial byte a byte. Si el fichero no existe, se creará en ese momento. Si el disco está lleno o protegido contra escritura, se lanzará la excepción *IOException*.

La clase *FileOutputStream* tiene los siguientes 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 se desea trabajar y borra el contenido previo del fichero comenzando a escribir desde el principio.
`FileOutputStream (String nombreFich, 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 se desea trabajar y, si append es true, se sitúa al final del fichero para añadir contenido desde el final.

Los principales métodos de la clase *FileOutputStream* son:

MÉTODO	EXPLICACIÓN
`void write(int b)`	Escribe un byte.
`void write(byte[] buffer)`	Escribe un vector de bytes.
`void write(byte[] buffer, int desplazamiento, int max)`	Escribe max bytes de datos del vector de bytes buffer comenzando por buffer[desplazamiento].
`void close()`	Libera los recursos del sistema asociados al flujo y lo cierra.

3.2.2. Las Clases DataInputStream y DataOutputStream

Para leer y escribir datos de tipos primitivos (como *boolean*, *int*, *short*, *long*, *float*, *double*, *char*, etc.), se utilizan las clases **DataInputStream** y **DataOutputStream**. Además de los métodos read() y write(), estas clases proporcionan métodos para la lectura y escritura de tipos primitivos de un modo independiente de la máquina.

Para construir un objeto **DataInputStream**, se necesita tener instanciado previamente un objeto *FileInputStream*. Por ejemplo:

```
File file = new File("C:\\directorio\\fichero.dat");
FileInputStream fis = new FileInputStream(file);
DataInputStream dis = new DataInputStream(fis);
```

Los principales métodos de la clase *DataInputStream* son:

MÉTODO	EXPLICACIÓN
`byte readByte()`	Lee un byte.
`int readUnsignedByte()`	Lee un byte sin signo.
`boolean readBoolean()`	Lee un dato de tipo booleano.
`short readShort()`	Lee un dato de tipo entero corto.
`int readUnsignedShort()`	Lee un dato de tipo entero corto sin signo.
`int readInt()`	Lee un dato de tipo entero.
`long readLong()`	Lee un dato de tipo entero largo.
`float readFloat()`	Lee un dato de tipo real con precisión simple.
`double readDouble()`	Lee un dato de tipo real con precisión doble.
`char readChar()`	Lee un dato de tipo carácter.
`String readUTF()`	Lee una cadena de caracteres en formato Unicode.

Para construir un objeto **DataOutputStream**, se necesita tener instanciado previamente un objeto *FileOutputStream*. Por ejemplo:

```
File file = new File("C:\\directorio\\fichero.dat");
FileOutputStream fos = new FileOutputStream(file);
DataOutputStream dos = new DataOutputStream(fos);
```

Los principales métodos de la clase *DataOutputStream* son:

MÉTODO	EXPLICACIÓN
`void writeByte(int i)`	Escribe un byte.
`void writeBoolean(boolean b)`	Escribe un dato de tipo booleano como 1 byte.
`void writeShort(int i)`	Escribe un dato de tipo entero corto como 2 bytes.
`void writeInt(int i)`	Escribe un dato de tipo entero como 4 bytes.
`void writeLong(long l)`	Escribe un dato de tipo entero largo como 8 bytes.
`void writeFloat(float f)`	Escribe un dato de tipo real con precisión simple como 4 bytes.
`void writeDouble(double d)`	Escribe un dato de tipo real con precisión doble como 8 bytes.
`void writeChar(int i)`	Escribe un dato de tipo carácter como 2 bytes.
`void writeChars(String s)`	Escribe una cadena de caracteres como una secuencia de caracteres.
`void writeBytes(String s)`	Escribe una cadena de caracteres como una secuencia de bytes.
`void writeUTF(String s)`	Escribe una cadena de caracteres en formato Unicode.

Hay que tener mucho cuidado con leer un fichero en el mismo formato en el que se ha escrito porque, de no ser así, se podrían producir 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 adecuadamente.

3.2.3. Las Clases ObjectInputStream y ObjectOutputStream

Se llama **persistencia** al proceso de almacenar toda la información que contiene un objeto, manteniendo su estructura, en un fichero binario. En Java, para poder guardar un objeto de una clase en un fichero binario, dicha clase tiene que implementar la interfaz **Serializable**, que dispone de métodos que permiten escribir y leer objetos en ficheros binarios:

MÉTODO PARA ESCRIBIR UN OBJETO	`void writeObject(ObjectOutputStream stream) throws IOException`
MÉTODO PARA LEER UN OBJETO	`void readObject(ObjectInputStream stream) throws IOException, ClassNotFoundException`

La **serialización** de objetos de Java permite tomar cualquier objeto que implemente la interfaz Serializable y convertirlo en una secuencia de bits, que puede ser posteriormente restaurada para regenerar el objeto original.

Para leer objetos serializables de un flujo se utiliza la clase **ObjectInputStream** y para escribir objetos serializables en un flujo se utiliza la clase **ObjectOutputStream**.

Por ejemplo, para leer un objeto "persona" de un fichero binario, se necesita crear el flujo de entrada a disco con *FileInputStream*, y a continuación, crear el flujo de entrada **ObjectInputStream**, que procesa los datos y estará vinculado al fichero. Por último, el método *readObject* leerá el objeto del flujo de entrada (consultándolo del fichero).

```
File file = new File("C:\\directorio\\fichero.dat");
FileInputStream fis = new FileInputStream(file);
ObjectInputStream ois = new ObjectInputStream(fis);
Persona persona = (Persona) ois.readObject();
```

El método *readObject* puede lanzar las excepciones *IOException* o *ClassNotFoundException*, por lo que será necesario controlarlas mediante un bloque *try-catch-finally*.

Por ejemplo, para escribir un objeto "persona" en un fichero binario, se necesita crear un flujo de salida a disco con *FileOutputStream*, y a continuación, crear el flujo de salida **ObjectOutputStream**, que procesa los datos y estará vinculado al fichero. Por último, el método *writeObject* escribirá el objeto en el flujo de salida (guardándolo en el fichero).

```
File file = new File("C:\\directorio\\fichero.dat");
FileOutputStream fos = new FileOutputStream(file);
ObjectOutputStream oos = new ObjectOutputStream(fos);
oos.writeObject(persona);
```

El método *writeObject* puede lanzar la excepción *IOException*, por lo que será necesario controlarla mediante un bloque *try-catch-finally*.

Existe un **problema** con los ficheros de objetos. Al crear un fichero de objetos, se crea una cabecera inicial con información y a continuación se añaden los objetos. Si se utiliza el mismo fichero otra vez para añadir más objetos, se creará una nueva cabecera y se añadirán estos objetos a partir de esa cabecera. El problema surge en la lectura del fichero cuando se encuentra con la segunda cabecera: aparecerá la excepción *StreamCorruptedException* y no se podrá seguir leyendo más objetos.

Para que el programa Java no añada estas cabeceras en el fichero, se puede crear una nueva clase **MyObjectOutputStream** que herede de *ObjectOutputStream*, y dentro de esta nueva clase, sobreescribir el método *writeStreamHeader* para que no realice nada.

En el programa principal de Java, antes de abrir el fichero para añadir objetos hay que comprobar si el fichero ya existe en el sistema de archivos. Si el fichero existe, se deberá usar la nueva clase redefinida *MyObjectOutputStream* para instanciar el flujo de salida, y en caso contrario, se deberá usar *ObjectOutputStream* para instanciar el flujo de salida.

4. ACCESO ALEATORIO

Hasta ahora, todas las operaciones sobre ficheros se han realizado de forma secuencial. Se empezaba la lectura en el primer carácter, el primer byte o el primer objeto, y seguidamente se leían los siguientes uno a continuación de otro hasta llegar al fin del fichero. De forma similar, para la escritura de datos en el fichero, se iban escribiendo a continuación de la última información escrita.

Java dispone de la clase **RandomAccessFile** para posicionar un cursor en una posición concreta de un fichero y acceder a su contenido de forma aleatoria o directa (no secuencial). Esta clase no es parte de la jerarquía *Reader/Writer* ni de la de *InputStream/OutputStream*, puesto que permite avanzar y retroceder dentro de un fichero.

La clase *RandomAccessFile* tiene los siguientes constructores (que pueden lanzar la excepción *FileNotFoundException*):

CONSTRUCTOR	EXPLICACIÓN
`RandomAccessFile (String nombreFich, String modoAcceso)`	Crea un flujo de acceso aleatorio asociado a un fichero a partir del nombre del fichero y con el modo de acceso indicado.
`RandomAccessFile (File fichero, String modoAcceso)`	Crea un flujo de acceso aleatorio asociado a un fichero a partir del objeto File y con el modo de acceso indicado.

El argumento `modoAcceso` especifica el **modo de acceso** con el que se abrirá el fichero:

MODO DE ACCESO	SIGNIFICADO
`"r"`	Abre el fichero en modo de solo lectura. El fichero deberá existir, en caso contrario, se lanzará la excepción *FileNotFoundException*. Cualquier operación de escritura sobre el fichero podrá lanzar la excepción *IOException*.
`"rw"`	Abre el fichero en modo de lectura y escritura. Si el fichero no existe, se realizará un intento de crearlo.

La clase *RandomAccessFile* maneja un **puntero** o **cursor** que indica la posición actual en el fichero. Cuando se abre un fichero, el puntero se coloca en 0, es decir, apuntando al principio del mismo. Además, esta clase implementa las interfaces de *DataInput* y *DataOutput*. Por tanto, una vez abierto un fichero, se pueden utilizar sus métodos *readXXX()* y *writeXXX()* para cada tipo de dato y las sucesivas llamadas a estos métodos *readXXX()* y *writeXXX()* ajustan el puntero según la cantidad de bytes leídos o escritos.

Los principales métodos de la clase *RandomAccessFile* son:

MÉTODO	EXPLICACIÓN
`long getFilePointer()`	Devuelve la posición actual del puntero del fichero.
`void seek (long posicion)`	Coloca el puntero del fichero en una posición determinada desde el comienzo del mismo.
`long length()`	Devuelve el tamaño del fichero en bytes y marca el final del fichero.
`int skipBytes (desplazamiento)`	Desplaza el puntero del fichero desde la posición actual el número de bytes indicados en desplazamiento.

Cada tipo primitivo de dato tiene un tamaño concreto en Java:

byte (1 byte) **short** (2 bytes) **int** (4 bytes) **long** (8 bytes)

float (4 bytes) **double** (8 bytes) **boolean** (1 bit) **carácter Unicode** (2 bytes)

5. DETECCIÓN Y TRATAMIENTO DE EXCEPCIONES

Una **excepción** es un evento que ocurre durante la ejecución del programa que interrumpe el flujo normal de las sentencias. Cuando no es capturada por el programa, por defecto, el gestor de excepciones la captura, retorna un mensaje y detiene el programa.

La ejecución del siguiente programa produce una excepción y visualiza un mensaje por pantalla indicando el error:

```
public class DivisionDeEnteros {

    public static void main(String[] args) {
        int dividendo = 10;
        int divisor = 0;
        int cociente = dividendo / divisor;
        System.out.println(dividendo + " / " + divisor + " = " + cociente);
    }

}
```

```
Exception in thread "main" java.lang.ArithmeticException: / by zero
        at DivisionDeEnteros.main(DivisionDeEnteros.java:8)
```

Cuando dicho error ocurre dentro de un método Java, el método crea un objeto *Exception* y lo maneja fuera, en el sistema de ejecución. El manejo de excepciones en Java está diseñado pensando en situaciones en las que el método que detecta el error no es capaz de manejarlo. En este caso, el método lanzará una excepción.

Las excepciones en Java son objetos de clases derivadas de la clase base **Exception**, que a su vez es una clase derivada de la clase base **Throwable**.

5.1. CAPTURA DE EXCEPCIONES

Para capturar una excepción se utiliza el bloque **try-catch-finally**:

- Se encierra en un bloque **try** el código que puede generar una excepción.
- Este bloque try va seguido de uno o más bloques catch. Cada bloque **catch** especifica un tipo de excepción que puede atrapar (capturar) y contiene un manejador de excepciones específico para ese tipo de excepción.
- Después del último bloque catch puede aparecer un bloque **finally** (opcional) que siempre se ejecuta, haya ocurrido o no la excepción. Este bloque finally se utiliza para cerrar ficheros o conexiones a bases de datos o liberar otros recursos del sistema después de que ocurra una excepción.

Para capturar cualquier excepción se utiliza la clase base **Exception**. Si se usa, hay que ponerla al final de la lista de manejadores para evitar que otros manejadores que vienen después queden ignorados.

```
// código que puede generar excepciones
try {

}
// manejo de la excepcion 1
catch (Excepcion1 e1) {

}
// manejo de la excepcion 2
catch (Excepcion2 e2) {

}
// etc.
// manejo de la excepcion N
catch (ExcepcionN eN) {

}
// código que se ejecuta siempre al final
finally {

}
```

Para obtener más información sobre la excepción producida, se pueden invocar los métodos de la clase base **Throwable**. Algunos de estos métodos son:

MÉTODO	EXPLICACIÓN
`String getMessage()`	Devuelve la cadena de error de este objeto.
`String getLocalizedMessage()`	Crea una descripción local de este objeto.
`String toString()`	Devuelve una descripción breve de este objeto.
`void printStackTrace()`	Imprime este objeto y la traza de la pila de llamadas realizadas en el flujo de salida estándar de errores.
`void printStackTrace (PrintStream ps)`	Imprime este objeto y la traza de la pila de llamadas realizadas en el flujo de salida especificado.
`void printStackTrace (PrintWriter pw)`	Imprime este objeto y la traza de la pila de llamadas realizadas en el escritor de impresión especificado.

Una sentencia try puede estar dentro de un bloque de otra sentencia try. Si la sentencia try interna no tiene un manejador catch, se buscará el manejador en las sentencias try más externas.

5.2. ESPECIFICACIÓN DE EXCEPCIONES

Para especificar una o varias excepciones se utiliza la palabra clave **throws**, seguida de la lista de todos los tipos de excepciones potenciales. Si un método decide no gestionar una excepción (mediante try-catch), deberá especificar que puede lanzar esa excepción.

El siguiente ejemplo indica que el método *main()* puede lanzar las excepciones *IOException* y *ClassNotFoundException*:

```
public static void main(String[] args)

throws IOException, ClassNotFoundException {}
```

Aquellos métodos que pueden lanzar excepciones deben saber cuáles son esas excepciones que se pueden producir durante su ejecución e indicarlas en su declaración. Una forma típica de conocerlas es compilando el programa.

La siguiente tabla muestra las excepciones que se pueden producir durante la ejecución de los diferentes métodos del paquete **java.io**:

EXCEPCIÓN	DESCRIPCIÓN
CharConversion Exception	Clase base para excepciones de conversión de caracteres.
EOF Exception	Indica que un fin de fichero o fin de flujo ha sido alcanzado de forma inesperada durante la entrada. Esta excepción se utiliza principalmente en flujos de entrada de datos para indicar el final del flujo. Otras operaciones de entrada devuelven un valor especial al final del flujo, en lugar de lanzar una excepción.
FileNotFound Exception	Indica que un intento de abrir el fichero denotado por una ruta especificada ha fallado. Esta excepción podrá ser lanzada en los constructores *FileInputStream*, *FileOutputStream* y *RandomAccessFile*, cuando un fichero no exista en la ruta especificada. También podrá ser lanzada en estos constructores si el fichero existe, pero es inaccesible por alguna razón (por ejemplo, intentar abrir un fichero de solo lectura para escribir).
InterruptedIO Exception	Indica que una operación de entrada/salida ha sido interrumpida. Esta excepción podrá ser lanzada para indicar que una transferencia de entrada o salida ha sido terminada debido a que el hilo que la realizaba ha sido interrumpido. El campo *bytesTransferred* indica cuántos bytes se han transferido con éxito antes de que la interrupción ocurriera.
InvalidClass Exception	Se lanza cuando en la ejecución de la serialización se detecta uno de los siguientes problemas con una clase: • La versión de serie de la clase no coincide con la del descriptor de clase leída del flujo. • La clase contiene tipos de datos desconocidos. • La clase no tiene un constructor accesible sin argumentos.
InvalidObject Exception	Indica que uno o más objetos deserializados han fallado las pruebas de validación. El argumento debería proporcionar el motivo del fallo.
IO Exception	Indica que una excepción de entrada/salida de algún tipo ha ocurrido. Esta clase es la clase general de excepciones producidas por operaciones de entrada/salida falladas o interrumpidas.
NotActive Exception	Se lanza cuando la serialización o la deserialización no está activa.
NotSerializable Exception	Se lanza cuando una instancia se requiere para tener un interfaz serializable. Esta excepción podrá ser lanzada en la ejecución de la serialización o la clase de la instancia. El argumento debería ser el nombre de la clase.
ObjectStream Exception	Superclase de todas las excepciones específicas para las clases de flujos de objetos.
OptionalData Exception	Indica el fallo de una operación de lectura de objeto debido a datos primitivos no leídos, o el final de los datos pertenecientes a un objeto serializado en el flujo. Esta excepción podrá ser lanzada en dos casos: • Cuando se intenta leer un objeto y el siguiente elemento del flujo corresponde a un dato primitivo. • Cuando se intenta leer pasado el fin de datos consumible, mediante un método *readObject* o *readExternal* definido en una clase.

StreamCorrupted Exception	Se lanza cuando la información de control que se ha leído de un flujo de objetos incumple las comprobaciones de consistencia internas.
SyncFailed Exception	Indica que una operación de sincronización ha fallado.
UncheckedIO Exception	Envuelve una excepción de entrada/salida dentro de una excepción no comprobada.
UnsupportedEncoding Exception	La codificación de caracteres no está soportada.
UTFDataFormat Exception	Indica que una cadena malformada en formato UTF-8 modificado (Unicode) ha sido leída de un flujo de entrada de datos o por cualquier clase que implemente la interfaz de entrada de datos. Véase la descripción de la clase *DataInput* para el formato en el que las cadenas UTF-8 modificadas son leídas y escritas.
WriteAborted Exception	Indica que una excepción de flujos de objetos (*ObjectStreamException*) se ha producido durante una operación de escritura. La excepción que ha terminado la escritura puede encontrarse en el campo *detail*. El flujo vuelve a su estado inicial y todas las referencias a objetos ya deserializados se descartan.