

Acceso a datos

## UD01. Acceso a ficheros

---

Desarrollo de Aplicaciones Multiplataforma

Enric Giménez Ribes

[e.gimenezribes@edu.gva.es](mailto:e.gimenezribes@edu.gva.es)

# ÍNDICE

<b>1. INTRODUCCIÓN</b>	<b>2</b>
<b>2. CLASE FILE</b>	<b>3</b>
<b>3. TIPOS DE FICHEROS</b>	<b>9</b>
3.1. Propiedades del sistema	10
3.2. Flujos o streams	11
3.3. Formas de acceso a un fichero	12
<b>4. FICHEROS DE TEXTO</b>	<b>13</b>
4.1. Ficheros de texto plano	14
4.2. Ficheros de configuración	16
4.3. Ficheros XML	18
<b>5. FICHEROS BINARIOS</b>	<b>21</b>
5.1. Lectura o escritura de ficheros binarios	22
5.2. Serialización de objetos	24
5.2.1. Problemas con la serialización de objetos	27
<b>6. RECOMENDACIONES</b>	<b>33</b>

## 1. INTRODUCCIÓN

Un **fichero** es una abstracción del sistema operativo para el almacenamiento genérico de datos. La parte del sistema operativo encargada del manejo de ficheros se denomina “**sistema de ficheros**”.

Los sistemas de ficheros tradicionales se encargan de organizar, almacenar y nombrar (identificar mediante un nombre) los ficheros almacenados en dispositivos de almacenamiento permanente, como los discos duros, las memorias USB de estado sólido, los DVDs. . .

Hoy en día, los sistemas de ficheros más conocidos son “ext3” o “ext4” para sistemas operativos Linux, “NTFS” para sistemas basados en Windows.

Aunque cada sistema de ficheros ofrece su propia visión de los datos almacenados en un disco y los gestiona y ordena a su manera, todos ellos comparten algunos aspectos generales:

- Los ficheros suelen organizarse en estructuras jerárquicas de directorios. Estos directorios son contenedores de ficheros (y de otros directorios) que permiten organizar los datos.
- El nombre de un fichero está relacionado con su posición en el árbol de directorios que lo contiene, lo que permite no sólo identificar unívocamente cada fichero, sino encontrarlo en el disco a partir de su nombre. Este nombre recibe el nombre de ruta o “path”.
  - Linux: “/home/usuario/fichero”.
  - Windows: “C:\Mis documentos\fichero.txt”.
- Los ficheros suelen tener una serie de metadatos asociados, como pueden ser su fecha de creación, la fecha de última modificación, su propietario o los permisos que tienen diferentes usuarios sobre ellos (lectura, escritura...).

Java permite:

1. **Interactuar con el sistema de archivos** independientemente de su tipo (FAT32, NTFS, EXT3, etc.) y del S.O. (Windows, Linux, MAC...).
2. **Crear y manipular ficheros binarios (byte)**: compuestos por datos de tipos primitivos.
3. **Crear y manipular ficheros de texto (char)**: compuestos tanto por las representaciones en caracteres de los tipos primitivos como por cadenas.
4. Trabajar con ficheros binarios compuestos por objetos (**serialización de objetos**).

## 2. CLASE FILE

En el paquete **java.io** se encuentra la clase **File** pensada para poder realizar operaciones de información sobre archivos. No proporciona métodos de acceso a los archivos, sino operaciones a nivel de sistema de archivos (listado de archivos, crear carpetas, borrar ficheros, cambiar nombre...).

Un objeto **File** representa un archivo o un directorio y sirve para obtener información (permisos, tamaño...). También sirve para navegar por la estructura de archivos.

### Construcción de objetos de archivo:

Utiliza como único argumento una cadena que representa una ruta en el sistema de archivo.

```
File archivo1 = new File("/datos/bd.txt");
File carpeta = new File("datos");
```

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 `/`.

Algunos de los métodos que tenemos son los siguientes:

Método / Constructor	Descripción
<code>File(String pathname)</code>	Crea un nuevo objeto de tipo <code>File</code> a partir de su ruta
<code>boolean delete()</code>	Borra el fichero/directorio
<code>boolean exists()</code>	Indica si el fichero/directorio existe
<code>String getName()</code>	Devuelve el nombre del fichero/directorio (sin la ruta)
<code>String getParent()</code>	Devuelve la ruta al fichero/directorio (sin el nombre)
<code>long length()</code>	Devuelve el tamaño del fichero. No válido para directorios
<code>File[] listFiles()</code>	Obtiene un listado de ficheros en el directorio
<code>boolean isDirectory()</code>	Indica si se trata de un directorio
...	

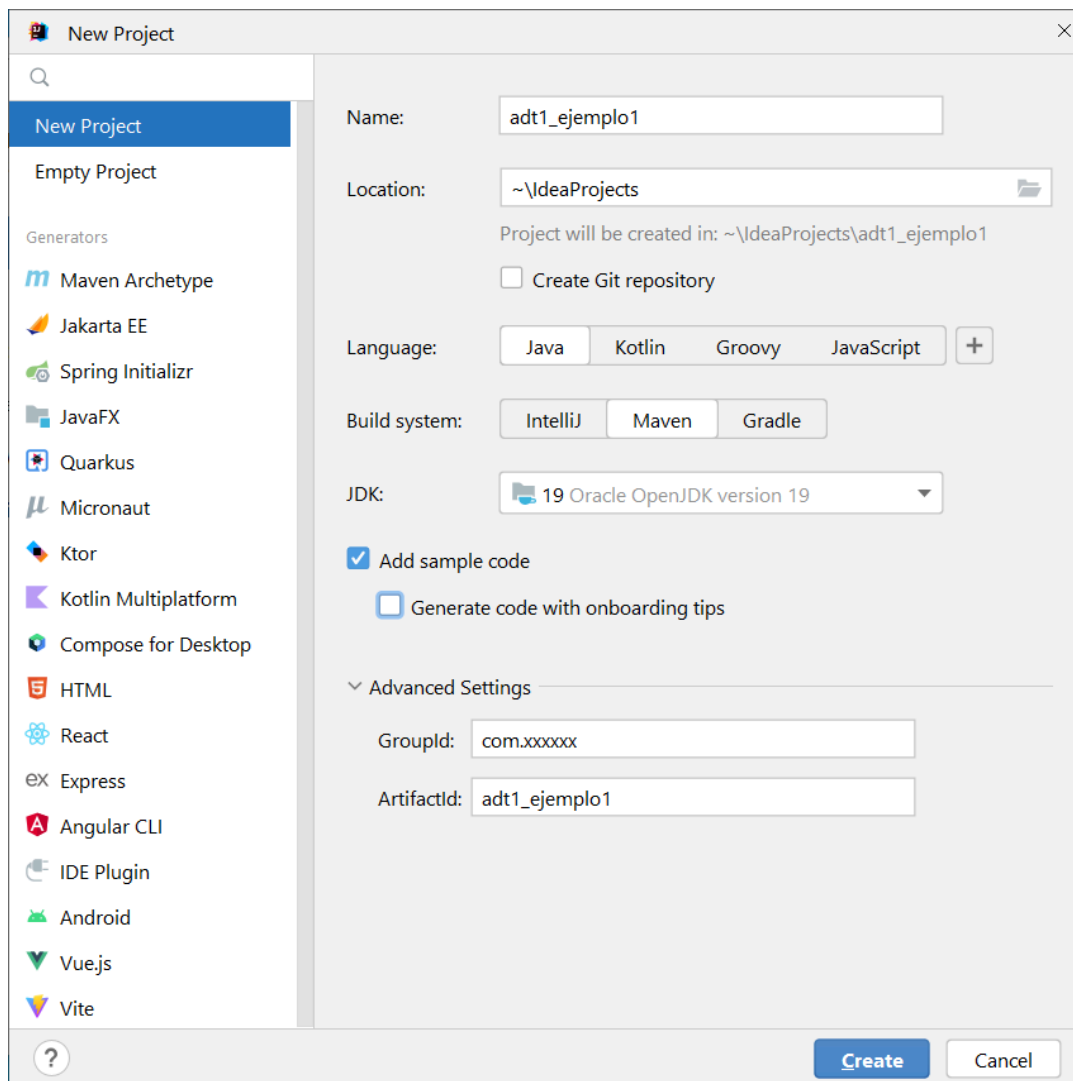


Crear un objeto `File` no afecta al sistema de archivos hasta que no se invocan métodos de la clase.

<https://docs.oracle.com/javase/7/docs/api/java/io/File.html>

**Ejemplo 1:** Realizar un programa que compruebe si existe el fichero file.txt en la carpeta resources:

En primer lugar, ejecutamos el IDE IntelliJ IDEA y creamos un nuevo proyecto:



En la siguiente pantalla indicaremos por este orden, **el nombre del proyecto, la ruta donde crear el proyecto y el Group Id**. El nombre de la aplicación será **adt1\_ejemplo1** (seguiremos esta nomenclatura durante el curso). La ruta la dejaremos por defecto. El tercero de los datos indicados tan sólo se utilizará como paquete de nuestras clases java. El Group Id utilizado será **com.xxxxxx**. El **xxxxxx** serán tus iniciales de dos en dos, esto se hace así ya que es el nombre del paquete.

Una vez creado el proyecto, renombramos la clase principal que nos ha creado por defecto (botón derecho Refactor → Rename) con el nombre de **App.java** (todas nuestras clases **Java Main Class** tendrán ese nombre).

En el método **main** que habrá creado, añadimos el siguiente código:

```
import java.io.File;


public class App {

    public static void main(String[] args) {

        File f = new File("./src/main/resources/file.txt");
        if (f.exists()) {
            System.out.println("¡El fichero existe!");
        } else {
            System.err.println("¡El fichero NO existe!");
        }

        System.out.println("getName(): " + f.getName());
        System.out.println("getParent(): " + f.getParent());
        System.out.println("length(): " + f.length());

    }
}
```

Ahora que ya tenemos configurado el proyecto, lo podemos ejecutar 

```
¡El fichero NO existe!
getName(): file.txt
getParent(): .\resources
length(): 0
```

Podemos ir a la carpeta del proyecto y crear manualmente una carpeta con nombre **/resources** y dentro un archivo llamado **file.txt** con alguna frase en su interior.

Volvemos a ejecutar el programa con el siguiente resultado:

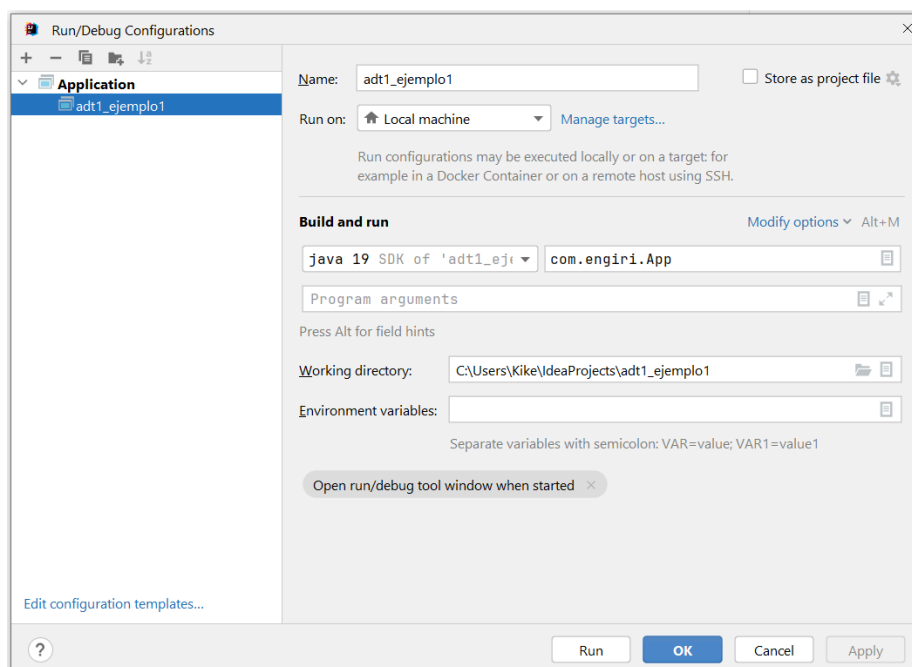
```
¡El fichero existe!
getName(): file.txt
getParent(): .\resources
length(): 4
```

## Configurar nuestro proyecto si tenemos problemas con las rutas de trabajo

Para configurar el directorio de trabajo (working directory) en IntelliJ IDEA, sigue estos pasos:

1. Ve al menú "Run" (Ejecutar) en la parte superior de la ventana.
2. Selecciona "Edit Configurations..." (Editar Configuraciones).
3. En la ventana de configuraciones de ejecución, selecciona la configuración que estás utilizando para ejecutar tu programa o crea una nueva si aún no tienes una (Application)
4. En la sección "Configuration" (Configuración), busca la opción "Working directory" (Directorio de trabajo). Debería haber un campo en blanco junto a esta opción.
5. Haz clic en el icono de carpeta o elige el directorio que deseas establecer como directorio de trabajo. Puedes navegar por el sistema de archivos para seleccionar la ubicación deseada.
6. Una vez que hayas seleccionado el directorio de trabajo, haz clic en "Apply" (Aplicar) o "OK" para guardar la configuración.

Ahora, cuando ejecutes tu programa, IntelliJ IDEA establecerá el directorio de trabajo en la ubicación que has especificado. Esto afectará cómo se resuelven las rutas relativas en tu código.



Es importante configurar el directorio de trabajo correctamente para que las rutas relativas se resuelvan de manera adecuada en tu aplicación.

**Ejemplo 2:** Realizar un programa que muestre el directorio raíz, todas las unidades del disco y todos los archivos del directorio raíz.

En este ejemplo vamos a ver la diferencia que hay entre los siguiente métodos:

- **listRoots():** muestra las diferentes unidades de disco del Sistema Operativo.
- **listFiles():** devuelve un ArrayList con los diferentes objetos de tipo File que contiene el objeto File seleccionado, con lo cual luego se pueden aplicar propiedades y métodos a cada uno de ellos.
- **list():** devuelve el nombre de las carpetas o archivos que contiene el objeto de tipo File seleccionado.

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

    String directorioRaiz = File.listRoots()[0].toString();
    File[] paths = File.listRoots();
    File direcC = new File("C:/");

    System.out.println("=====");
    System.out.println("Directorio raíz: " + directorioRaiz);

    System.out.println("=====");
    System.out.println("Unidades de disco:");
    for (File fich : paths) {
        System.out.println("\t" + fich.toString());
    }

    System.out.println("=====");
    System.out.println("Los archivos y/o carpetas que contiene " + direcC);
    for (File f : direcC.listFiles()) {
        System.out.println("\t" + f + (f.isFile() ? " - fichero" : " - directorio"));
    }

    System.out.println("=====");
    System.out.println("Los nombres de los archivos y/o carpetas que contiene " +
    direcC);
    for (String f : direcC.list()) {
        System.out.println("\t" + f);
    }
}
```



Si ejecutamos el proyecto:

```
=====
Directorio raíz: C:\
=====
Unidades de disco:
    C:\
    E:\
=====
Los archivos y/o carpetas que contiene C:\
    C:\$Recycle.Bin - directorio
    C:\$WinREAgent - directorio
    C:\android - directorio
    ...
=====
Los nombres de los archivos y/o carpetas que contiene C:\
    $Recycle.Bin
    $WinREAgent
    android
    ...
```

### Ejemplo 3: Visualiza el sistema de archivos teniendo en cuenta el directorio actual “.”

```
public static void main(String[] args) {  
  
    String dir = ".";  
    //Directorio actual  
    File f = new File(dir);  
    String[] archivos = f.list();  
  
    System.out.printf("Ficheros en el directorio actual: %d %n", archivos.length);  
  
    for (int i = 0; i < archivos.length; i++) {  
        File f2 = new File(f, archivos[i]);  
        System.out.printf("Nombre: %s, es fichero?: %b, es Directorio?: %b %n",  
            archivos[i], f2.isFile(), f2.isDirectory());  
    }  
  
}
```

Ejecutamos el proyecto:

```
Ficheros en el directorio actual: 5  
Nombre: .gitignore, es fichero?: true, es Directorio?: false  
Nombre: .idea, es fichero?: false, es Directorio?: true  
Nombre: pom.xml, es fichero?: true, es Directorio?: false  
Nombre: src, es fichero?: false, es Directorio?: true  
Nombre: target, es fichero?: false, es Directorio?: true
```

### 3. TIPOS DE FICHEROS

Desde el punto de vista de un programador solamente distinguiremos entre dos tipos de ficheros:

- **Ficheros de texto** cuando el contenido del fichero contenga exclusivamente caracteres de texto (podemos leerlo con un simple editor de texto).

Extensión	Tipo de fichero
.txt	Fichero de texto plano
.xml	Fichero XML
.json	Fichero de intercambio de información
.props	Fichero de propiedades
.conf	Fichero de configuración
.sql	Script SQL
.srt	Fichero de subtítulo

- **Ficheros binarios** cuando no están compuestos exclusivamente de texto. Pueden contener imágenes, vídeos, ficheros... aunque también podemos considerar un fichero binario a un fichero de Microsoft Word en el que sólo hayamos escrito algún texto puesto que, al almacenarse el fichero, el procesador de texto incluye alguna información binaria.

Extensión	Tipo de fichero
.pdf	Fichero PDF
.jpg	Fichero de imagen
.doc, .docx	Fichero de Microsoft Word
.avi	Fichero de video
.ppt, .pptx	Fichero de PowerPoint

A veces, en ficheros binarios, podremos encontrarnos con las extensiones *.bin* o *.dat* para hacer referencia a ficheros que contienen información binaria en un formato que no está ampliamente difundido. Serán simplemente ficheros que una aplicación determinada es capaz de leer/escribir de una forma específica sólo definida para dicha aplicación.

### 3.1. Propiedades del sistema

En cualquier caso, para el acceso a ficheros independientemente del tipo de los mismos, conviene conocer el funcionamiento de las [System Properties](#) de las que dispone Java en su API. Éstas permiten acceder a propiedades de la configuración del sistema y, entre otras, podemos encontrarnos con algunas muy interesantes relacionadas con este tema:

- **"file.separator"** → Obtiene el carácter, según el Sistema Operativo, para la separación de las rutas (/ ó \). También se puede utilizar la constante **File.separator**.
- **"user.home"** → Obtiene la ruta de la carpeta personal del usuario (que dependerá del Sistema Operativo en cada caso).
- **"user.dir"** → Obtiene la ruta en la que se encuentra actualmente el usuario.
- **"line.separator"** → Obtiene el carácter que separa las líneas de un fichero de texto (difiere entre Windows/Linux).

En el caso de que se quiera acceder al valor de alguna de estas propiedades debe hacerse utilizando la llamada al método **System.getProperty(String)**.

```
System.out.println("La carpeta de mi usuario es " + System.getProperty("user.home"));
```

### 3.2. Flujos o streams

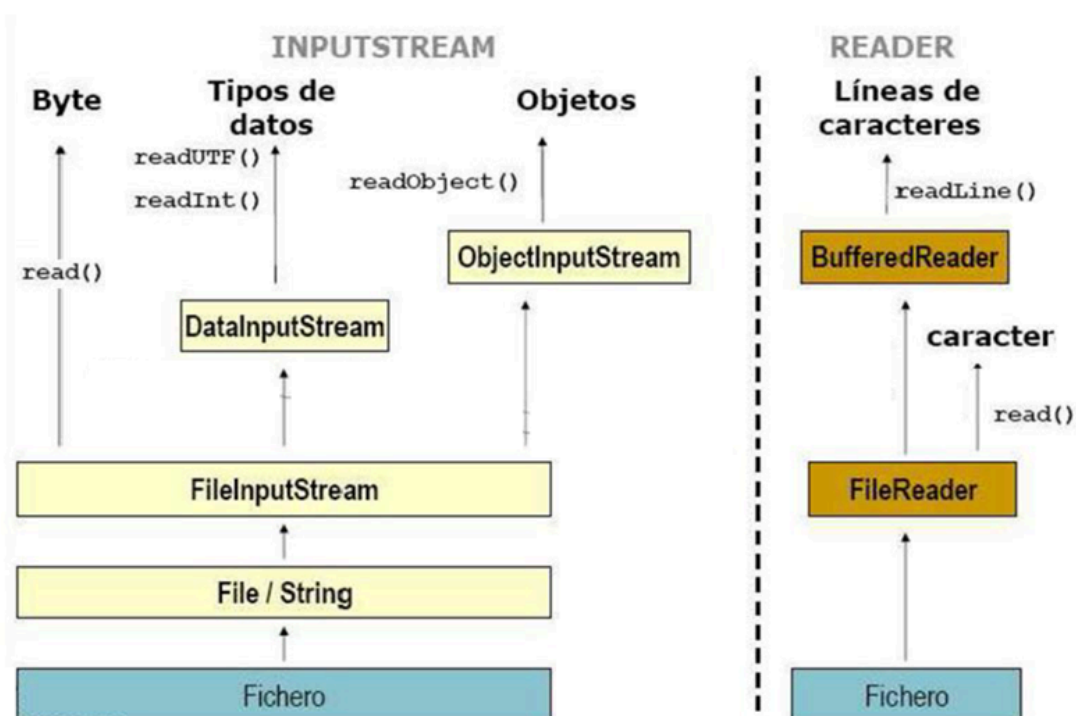
En Java la entrada/salida se realiza utilizando flujos (o streams), que son secuencias de información que tiene una fuente (flujos de entrada) o un destino (flujos de salida).



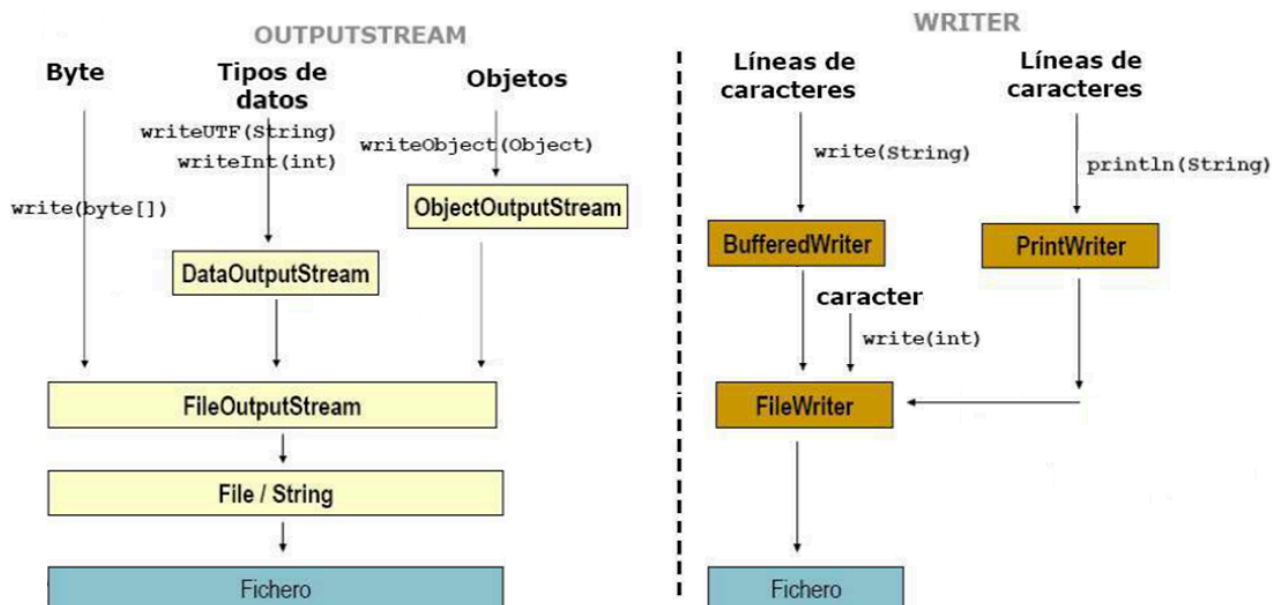
Existen 2 tipos de flujos de datos:

- **Flujos de bytes (8 bits o 1 byte)**
  - Realizan operaciones de entrada/salida de bytes.
  - Uso orientado a la lectura/escritura de ficheros binarios.
  - Clases abstractas: INPUTSTREAM y OUTPUTSTREAM.
- **Flujos de caracteres (16 bits o 2 bytes)**
  - Realizan operaciones de entrada/salida de caracteres.
  - Uso orientado a la lectura/escritura de ficheros de texto.
  - Clases abstractas: READER y WRITER.

### ESQUEMA GENERAL FLUJOS DE ENTRADA



## ESQUEMA GENERAL FLUJOS DE SALIDA



BufferedReader y BufferedWriter se utilizan para evitar que cada lectura o escritura acceda directamente al fichero, ya que utilizan un buffer intermedio entre la memoria y el fichero.

### 3.3. Formas de acceso a un fichero

Existen dos formas de acceso a la información:

- **Acceso secuencial:** Los datos se leen y se escriben en orden. Si se quiere acceder a un dato que está hacia la mitad del fichero es necesario leer antes todos los anteriores. La escritura de datos se hará a partir del último dato escrito, no es posible realizar inserciones entre los datos ya escritos.
  - Para el acceso binario: FileInputStream y FileOutputStream.
  - Para el acceso a caracteres: FileReader y FileWriter.
- **Acceso aleatorio:** Permite acceder directamente a un dato sin necesidad de leer los anteriores. Los datos están almacenados en registros de tamaño conocido, por lo que nos podemos mover de uno a otro para leerlos o modificarlos.
  - Se utiliza la clase RandomAccessFile.

## 4. FICHEROS DE TEXTO

En esta parte vamos a trabajar con 3 tipos de ficheros de texto:

- **Ficheros de texto plano** que contendrán texto *libre* y donde podremos escribir sin respetar ningún tipo de formato.
- **Ficheros de configuración** que contendrán información de configuración para una aplicación. Tienen un formato específico y Java además proporciona una API para trabajar más cómodamente con ellos.
- **Ficheros XML** que contienen información y están acompañados de etiquetas que le dan significado. Tienen unas reglas y formato más o menos definido y Java proporciona una API para trabajar con ellos.

Existen más tipos de ficheros de texto también muy extendidos como *.json* y *.csv* pero no serán estudiados en esta parte.

### 4.1. Ficheros de texto plano

Los ficheros de texto son aquellos que únicamente contienen texto, por lo que pueden ser editados directamente con cualquier editor de texto plano (Bloc de Notas, notepad++, . . .). Se podría decir que son aquellos ficheros que podrían ser *leídos* por cualquier persona. Son aquellos que normalmente se almacenan con la extensión *.txt* pero también podríamos incluir los scripts SQL (*.sql*), ficheros de código Java (*.java*), ficheros de configuración (*.ini*, *.props*, *.conf*, . . .), . . .

También se incluyen en la categoría de ficheros de texto los que además incluyen información adicional (siempre en forma de texto) que permiten interpretar los datos del fichero de una manera u otra, añadiendo más información al mismo. Estos formatos son HTML, XML, JSON, . . .

En cualquier caso, desde Java siempre se podrán leer/escribir de la misma manera, según veremos a continuación. También veremos cómo pueden leerse/escribirse utilizando librerías aquellos ficheros de texto plano que contienen formato, como hemos comentado anteriormente, haciendo de esa forma mucho más fácil el trabajo.

## Escribir ficheros de texto plano

Como hemos comentado anteriormente, la forma más cómoda de generar y leer ficheros de texto es mediante las clases **FileReader** y **FileWriter**.

Podemos utilizar la clase **PrintWriter**: escribe representaciones formateadas de tipos primitivos a un fichero de texto.

Método / Constructor	Descripción
<code>PrintWriter(File file)</code>	Crea un nuevo <code>PrintWriter</code> a partir de un <code>File</code>
<code>PrintWriter(OutputStream out)</code>	Crea un nuevo <code>PrintWriter</code> a partir de un stream
<code>void print(float f)</code>	Escribe un float (y no termina la línea)
<code>void println(float f)</code>	Escribe un float (y añade un salto de línea)
<code>void print(boolean b)</code>	Escribe un boolean (y no termina la línea)
...	...
<code>boolean checkError()</code>	Comprueba si ha habido error en la escritura anterior

Por defecto, la creación de un objeto de la clase `PrintWriter` provoca que:

- Si el fichero existe entonces se trunca su tamaño a cero.
- Si no, se crea un nuevo fichero.

Para añadir datos al final del fichero de texto (append):

```
PrintWriter pw = new PrintWriter(new FileWriter("/tmp/datos.txt", true));  
pw.println("Frase al final del fichero");
```

No olvidar cerrar el fichero con:

```
pw.close();
```



**Ejemplo 4:** Realiza un programa que abra un fichero de texto para escribir una frase.

```
FileWriter fichero = null;
PrintWriter escritor = null;

try {
    fichero = new FileWriter("./src/main/resources/archivo.txt");
    escritor = new PrintWriter(fichero) ;
    escritor.println("Esto es una línea del fichero");
} catch (IOException ioe) {
    ioe.printStackTrace() ;
} finally {
    if (fichero != null)
        try {
            fichero.close();
        } catch (IOException ioe) { . . . }
}
```

Si modificamos: `fichero = new FileWriter("archivo.txt", true);`, añadiría al final del fichero y no lo borraría cada vez que ejecutemos el programa.

### Leer ficheros de texto plano

**Ejemplo 5:** La clase `FileReader` permite procesar ficheros de texto. Algunos de sus métodos:

- `int read()` → lee un caracter y lo devuelve. Si llega al final del fichero devuelve -1.
- `String readLine()` → lee una línea y la devuelve. Si llega al final del fichero devuelve null.

```
File fichero = null;
BufferedReader buffer = null;

try {
    fichero = new File("./src/main/resources/archivo.txt");
    buffer = new BufferedReader(new FileReader(fichero));
    String linea = null;
    while ((linea = buffer.readLine()) != null) {
        System.out.println(linea);
    }
} catch (FileNotFoundException fnfe) {
    fnfe.printStackTrace();
} catch (IOException ioe) {
    ioe.printStackTrace();
} finally {
    if (buffer != null)
        try {
            buffer.close();
        } catch (IOException ioe) { }
}
```

## 4.2. Ficheros de configuración

En la API de Java se incluyen librerías para trabajar con los ficheros de configuración. Puesto que todos siguen un mismo patrón, es la librería la que se encarga de acceder al fichero a bajo nivel y el programador sólo tiene que indicar a qué propiedad quiere acceder o que propiedad quiere modificar, sin tener que añadir nada de código para leer o escribir el fichero tal y como hemos visto en el punto anterior.

Un ejemplo de fichero de configuración de esta librería de java sería el fichero que sigue:

### configuracion.props

```
# Fichero de configuración
# Thu Nov 14 10:49:39 CET 2013

user=usuario
password=micontrasena
server=localhost
port=3306
```

### Escribir ficheros de configuración

Así, si queremos generar, desde Java, un fichero de configuración como el anterior:

```
. . .
Properties configuracion = new Properties();
configuracion.setProperty("user", miUsuario);
configuracion.setProperty("password", miContrasena);
configuracion.setProperty("server", elServidor);
configuracion.setProperty("port", elPuerto);
try {
    configuracion.store(new FileOutputStream("configuracion.props"),
                        "Fichero de configuracion");
} catch (FileNotFoundException fnfe) {
    fnfe.printStackTrace();
} catch (IOException ioe) {
    ioe.printStackTrace();
}
. . .
```

## Leer ficheros de configuración

A la hora de leerlo, en vez de tener que recorrer todo el fichero como suele ocurrir con los ficheros de texto, simplemente tendremos que cargarlo e indicar de qué propiedad queremos obtener su valor (`getProperty(String)`).

```
. . .
Properties configuracion = new Properties();
try {
    configuracion.load(new FileInputStream("configuracion.props"));
    usuario = configuracion.getProperty("user");
    password = configuracion.getProperty("password");
    servidor = configuracion.getProperty("server");
    puerto = Integer.valueOf(configuracion.getProperty("port"));
} catch (FileNotFoundException fnfe) {
    fnfe.printStackTrace();
} catch (IOException ioe) {
    ioe.printStackTrace();
}
. . .
```

Para ambos casos, escribir y leer este tipo de ficheros, hay que tener en cuenta que, al tratarse de ficheros de texto, toda la información se almacena como si de un String se tratara. Por tanto, todos aquellos tipos Date, boolean o incluso cualquier tipo numérico serán almacenados en formato texto. Así, habrá que tener en cuenta las siguientes consideraciones:

- Para el caso de las fechas, deberán ser convertidas a texto cuando se quieran escribir y nuevamente convertidas a Date cuando se lea el fichero y queramos trabajar con ellas.
- Para el caso de los tipos boolean, podemos usar el método `String.valueOf(boolean)` para pasarlos a String cuando queramos escribirlos. En caso de que queramos leer el fichero y pasar el valor a tipo boolean podremos usar el método `Boolean.parseBoolean(String)`.
- Para el caso de los tipos numéricos (integer, float, double) es muy sencillo ya que Java los convertirá a String cuando sea necesario al escribir el fichero. En el caso de que queramos leerlo y convertirlos a su tipo concreto, podremos usar los métodos `Integer.parseInt(String)`, `Float.parseFloat(String)` y `Double.parseDouble()`, según proceda.

Se puede comprobar en el ejemplo 9 del repositorio `adt1_ejemplo9`.

### 4.3. Ficheros XML

Los ficheros XML permiten el intercambio de información entre aplicaciones utilizando para ello un fichero de texto plano al que se le pueden añadir etiquetas para darle significado a cada uno de los valores que se almacenan.

Por ejemplo, el siguiente fichero XML podría ser el resultado de volcar una Base de Datos sobre productos de una compañía, de forma que dicha información podría ahora leerse desde otra aplicación e incorporarla. Se puede ver como a parte de los datos de dichos productos (sus nombres, precios, . . .) aparece otra información en forma de etiquetas (entre los caracteres < y >) que permite dar significado a cada dato almacenado en el fichero. Así, podemos saber de qué estamos hablando y a qué corresponde cada valor.

Obviamente, tal y como ocurría con los ficheros de configuración, toda la información se tiene que pasar a texto para poder crear el fichero y reconvertida a su tipo original cuando se cargue de nuevo. Las mismas consideraciones que hemos tenido en cuenta en los ficheros de configuración nos servirán ahora.

#### productos.xml

```
<?xml version="1.0" encoding="UTF-8" standalone="no">
<xml>
<productos>
  <producto>
    <nombre>Cereales</nombre>
    <precio>3.45</precio>
  </producto>
  <producto>
    <nombre>Colacao</nombre>
    <precio>1.45</precio>
  </producto>
  <producto>
    <nombre>Agua mineral</nombre>
    <precio>1.00</precio>
  </producto>
</productos>
</xml>
```

La clase Java que definiría cada uno de los objetos que se representan por el fichero XML es:

### Producto.java

```
public class Producto {  
    private String nombre;  
    private float precio;  
  
    public Producto(String nombre, float precio) {  
        this.nombre = nombre;  
        this.precio = precio;  
    }  
  
    public String getNombre () { return nombre; }  
    public void setNombre (String nombre) { this.nombre = nombre; }  
  
    public float getPrecio () { return precio; }  
    public void setPrecio (float precio) { this.precio = precio; }  
}
```

### Escribir ficheros XML

Si ahora queremos generar el fichero XML con toda la información de una colección de objetos **Producto**, podemos utilizar el siguiente código:

```
. . .  
documento = dom.createDocument(null, "xml", null);  
Element raiz = document.createElement("productos");  
documento.getDocumentElement().appendChild(raiz);  
Element nodoProducto = null , nodoDatos = null ;  
Text texto = null;  
  
for (Producto producto : listaProductos) {  
    nodoProducto = documento.createElement("producto");  
    raiz.appendChild(nodoProducto);  
    nodoDatos = documento.createElement("nombre");  
    nodoProducto.appendChild(nodoDatos);  
    texto = documento.createTextNode(producto.getNombre());  
    nodoDatos.appendChild(texto);  
    nodoDatos = documento.createElement("precio");  
    nodoProducto.appendChild(nodoDatos);  
    texto = documento.createTextNode(producto.getPrecio());  
    nodoDatos.appendChild(texto);  
}  
. . .
```

## **Leer ficheros XML**

Y si lo que queremos es leer un fichero XML y cargarlo como una colección Java de objetos Producto, el ejemplo siguiente muestra un breve ejemplo acerca de cómo acceder a la información de dicho fichero XML.

```
. . .
NodeList productos = documento.getElementyByTagName("producto");
for (int i = 0; i < productos.getLength(); i++) {
    Node producto = productos . item ( i ) ;
    Element elemento = ( Element ) producto ;
    System.out.println(elemento.getElementsByTagName("nombre").item(0)
        .getChildNodes().item(0).getNodeValue());
    System.out.println(elemento.getElementsByTagName("precio").item(0)
        .gethildNodes().item(0).getNodeValue());
}
. . .
```

## 5. FICHEROS BINARIOS

Un fichero binario o de datos está formado por secuencias de bytes. Estos archivos pueden contener datos de tipo básico (int, float, char, etc) y objetos.

Para poder leer el contenido de un fichero binario debemos conocer la estructura interna del fichero, es decir, debemos saber cómo se han escrito: si hay enteros, long, etc. y en qué orden están escritos en el fichero. Si no se conoce su estructura podemos leerlo byte a byte.

### 5.1. Lectura o escritura de ficheros binarios

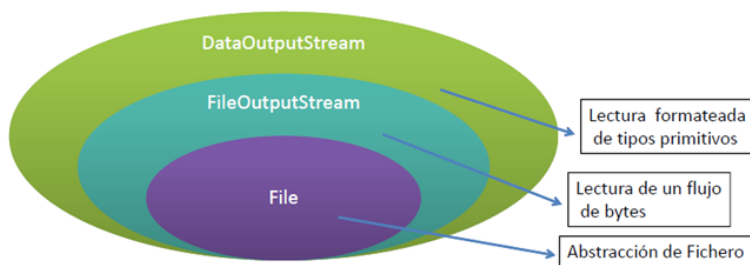
La lectura/escritura de tipos primitivos (boolean, int, float, String, etc.) en formato binario sobre ficheros se realiza usando las clases **DataInputStream** (lectura) y **DataOutputStream** (escritura) del paquete java.io.

Para **lectura** o **escritura** de fichero binario:

1. Crear un File con el **origen** / **destino** de datos.
2. Envolverlo en un **FileInputStream** / **FileOutputStream** para crear un flujo de datos **desde** / **hacia** el fichero.
3. Envolver el objeto anterior en un **DataInputStream** / **DataOutputStream** para poder **leer** / **escribir** tipos de datos primitivos del flujo de datos.
4. Usar métodos del estilo **readInt**, **readDouble**, **writeInt**, **writeDouble**, etc.

Método / Constructor	Descripción
<code>DataOutputStream(OutputStream out)</code>	Crea un nuevo objeto a partir del stream de salida
<code>void writeInt(int v)</code>	Escribe el entero al stream de salida como 4 bytes
<code>void writeLong(long v)</code>	Escribe el long al stream de salida como 8 bytes
<code>void writeUTF(String str)</code>	Escribe el String en formato portable (UTF8 mod.)
<code>void writeDouble(double v)</code>	Escribe el double al stream de salida como 8 bytes
...	

## Uso de DataOutputStream



El constructor de FileOutputStream puede lanzar la excepción FileNotFoundException:

- Si el fichero especificado no existe en el sistema de archivos.

Los métodos de escritura de DataOutputStream y los métodos de lectura de DataInputStream pueden lanzar la excepción IOException:

- Si ocurre algún problema al escribir los bytes en el disco (fallo en el dispositivo, etc.).

Estas excepciones deben ser gestionadas convenientemente mediante el uso de un bloque try - catch.

### Escribir y leer ficheros binarios

#### **Ejemplo 6:** Escritura y lectura de datos primitivos en un fichero binario

```

String fichero = "./src/main/resources/ejemplo6.data";
String nombre = "IIP";
int conv = 1;
double nota = 7.8;

try {
    // Escritura de datos primitivos en un fichero binario
    DataOutputStream out = new DataOutputStream(new FileOutputStream(fichero));
    out.writeUTF(nombre);
    out.writeInt(conv);
    out.writeDouble(nota);
    out.close();

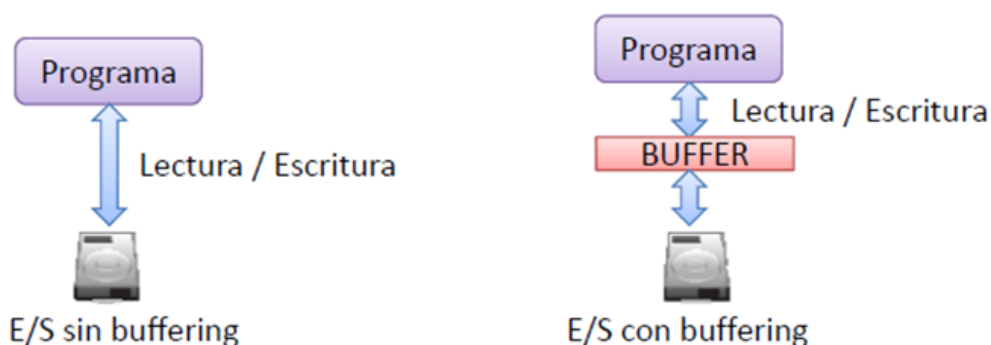
    // Lectura de datos primitivos en un fichero binario
    DataInputStream in = new DataInputStream(new FileInputStream(fichero));
    System.out.println("Valor leído de nombre: " + in.readUTF());
    System.out.println("Valor leído de convocatoria: " + in.readInt());
    System.out.println("Valor leído de nota: " + in.readDouble());
    in.close();
}
    
```



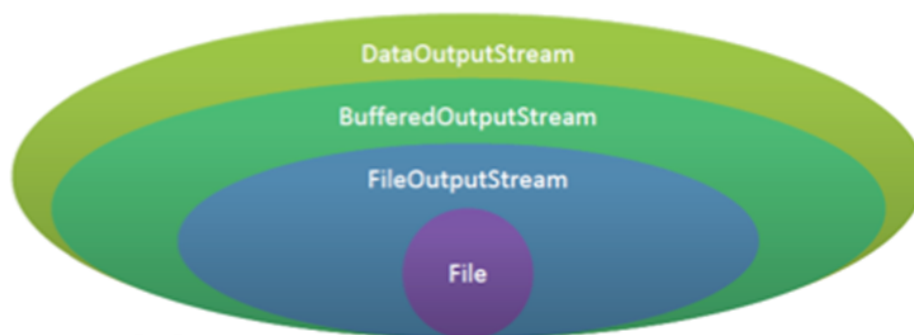
```
} catch (FileNotFoundException e) {
    System.err.println("Problemas con el fichero " + fichero + "." +
        e.getMessage());
} catch (IOException e) {
    System.err.println("Problemas al escribir en el fichero " + fichero);
}
```

En principio, las escrituras de datos mediante `DataOutputStream` desencadenan escrituras inmediatas sobre el disco, un proceso costoso.

- Solución eficiente: Uso de un buffer que almacena en memoria temporalmente los datos a guardar y, cuando hay suficientes datos, se desencadena la escritura en disco.
  - Es más eficiente realizar 1 escritura en disco de 100Kbytes que 100 escrituras de 1Kbyte.



Clases `BufferedInputStream` / `BufferedOutputStream`



## 5.2. Serialización de objetos

**Serializar** es el proceso por el cual un objeto en memoria pasa a transformarse en una estructura que pueda ser almacenada en un fichero (persistencia). Al proceso contrario le llamaremos **deserializar**.

La serialización de objetos permite:

- Convertir cualquier objeto que implemente la interfaz **Serializable** en una secuencia de bits que puede ser utilizada posteriormente para reconstruir el objeto original.
- Esta secuencia de bits puede guardarse en un fichero o puede enviarse a otra máquina virtual (que puede estar ejecutándose en otro SO) para reconstruir el objeto en otro instante o en otra máquina virtual.

Para leer y escribir objetos serializables a un fichero se utilizan las claves Java **ObjectInputStream** y **ObjectOutputStream** respectivamente.

Las consideraciones que tenemos que tener en cuenta para serializar objetos son las siguientes:

- Cualquier clase que desee poder serializar sus objetos debe implementar la interfaz **Serializable**.
- En esta implementación el objeto define cómo debe almacenarse o recuperarse en un fichero.
- Si se trata de serializar un objeto que no se implementa la interfaz **Serializable**, se obtiene la excepción **NotSerializableException**.
- No se serializan las variables **static** o **transient**.
  - <http://lineadecodigo.com/java/variables-java-transient/>
- Si los objetos mantienen referencias a otros objetos. Estos otros objetos deben ser también almacenados y recuperados con el fin de mantener las relaciones originales.
- La interfaz **Serializable** no define ningún método, sirve como un indicador.

## Ejemplo 7: Serialización de objetos en ficheros binarios

### Punto.java

```
public class Punto implements Serializable {  
  
    int x;  
    int y;  
  
    public Punto(int x, int y) {  
        this.x = x;  
        this.y = y;  
    }  
  
    public String toString() {  
        return "Punto[" + x + ", " + y + "];"  
    }  
  
}
```

```
public static void main(String[] args) {  
    String nombre = "./src/main/resources/ejemploObjeto1.data";  
    Punto punto = new Punto(52, 5);  
  
    // Escritura del objeto  
    ObjectOutputStream oos = null;  
  
    try {  
        oos = new ObjectOutputStream(new FileOutputStream(nombre));  
        oos.writeObject(punto);  
    } catch (IOException ex) {  
        System.out.println("Error escribiendo objeto");  
    } finally {  
        if (oos != null) {  
            try {  
                oos.close();  
            } catch (IOException ex) {  
                System.out.println("Error cerrando escritura");  
            }  
        }  
    }  
}
```

```
// Lectura del objeto
ObjectInputStream ois = null;

try {
    ois = new ObjectInputStream(new FileInputStream(nombre));
    Punto lecturaPunto = (Punto) ois.readObject();
    System.out.println(lecturaPunto);

} catch (FileNotFoundException ex) {
    System.out.println("Error en la lectura: no existe el fichero.");
} catch (IOException ex) {
    System.out.println("Error en la lectura del objeto");
} catch (ClassNotFoundException ex) {
    System.out.println("Error en la lectura del objeto: clase no existe");
} finally {
    if (ois != null) {
        try {
            ois.close();
        } catch (IOException ex) {
            System.out.println("Error cerrando lectura");
        }
    }
}
}
```

### 5.2.1. Problemas con la serialización de objetos

#### **Problema 1**

Al instanciar el `ObjectOutputStream` escribe unos bytes de cabecera en el fichero, antes incluso de que escribamos nada. Como el `ObjectInputStream` lee correctamente estos bytes de cabecera, aparentemente no pasa nada y ni siquiera nos enteramos de que existen.

El problema se presenta si escribimos unos datos en el fichero y lo cerramos. Luego volvemos a abrirlo para añadir datos, creando un nuevo `ObjectOutputStream` mediante el siguiente código:

```
/* El true indica que se abre el fichero para añadir datos al final del fichero.*/
ObjectOutputStream oos = new ObjectOutputStream(new FileOutputStream(fichero, true));
```

Esto escribe una nueva cabecera justo al final del fichero. Luego se irán añadiendo los objetos que vayamos escribiendo. El fichero contendrá lo que vemos en la siguiente ilustración, dos cabeceras:

Primera sesión con el fichero					Segunda sesión con el fichero. Le añadimos datos			
cabecera	Persona	Persona	Persona	Persona	cabecera	Persona	Persona	Persona

¿Qué pasa cuando leemos el fichero? Al crear el `ObjectInputStream`, este lee la cabecera del principio y luego se pone a leer objetos. Cuando llegamos a la segunda cabecera que se añadió al abrir por segunda vez el fichero para añadir datos, obtendremos un error `StreamCorruptedException` y no podremos leer más objetos.

Una solución es evidente, no usar más que un solo `ObjectOutputStream` para escribir todo. Sin embargo, esto no es siempre posible. Por ejemplo, si nuestro programa es una agenda, un día escribimos tres amigos, cerramos la agenda, apagamos el ordenador y nos vamos de juerga. Al día siguiente, queremos meter a otros dos amigos, arrancamos la agenda y por más que busquemos, de nuestro antiguo `ObjectOutputStream` no queda ni rastro. Hay que abrir uno nuevo. No se puede pretender en una agenda que metamos a todos nuestros amigos de una sola vez.

Una buena solución al problema es hacernos nuestro propio `ObjectOutputStream`, heredando del original (o sea creamos una nueva clase que herede de la otra) y redefiniendo el método `writeStreamHeader()` vacío, o sea que no haga nada. Esto solo se utilizará cuando el fichero ya existe y tiene una cabecera principal y queremos volver a escribir en el mismo.

La nueva clase obtenida será la siguiente:

```

public class AppendableObjectOutputStream extends ObjectOutputStream {

    public AppendableObjectOutputStream(OutputStream out) throws IOException {
        super(out);
    }

    @Override
    protected void writeStreamHeader() throws IOException {}
}

```

Cuando escribamos la segunda vez utilizaremos esta clase y no la `ObjectOutputStream`.

**Ejemplo 8:** Crea una aplicación que serializa Personas y sus Mascotas, de forma que se puedan añadir datos al fichero mascotas.dat, una vez creado.

Hemos simplificado la creación de las clases Mascota y Persona para centrarnos en la serialización de objetos que es el estudio de este ejemplo.

```
public class Mascota implements Serializable
{
    public String nombre;
    public int numeroPatas;
}
```

```
public class Persona implements Serializable {

    public String nombre;
    public String apellido;
    public Mascota mascota = new Mascota();
    public int edad;

    /**
     * Rellena los campos añadiendo i a unos nombres por defecto
     * @param i Un valor para añadir al final de los nombres
     */
    public Persona(int i) {
        setPersona(i);
    }

    /**
     * Rellena todos los campos de la clase con nombres por defecto a los que
     * añade el número i.
     */
    public void setPersona(int i) {
        this.nombre = "nombre" + i;
        this.apellido = "apellido" + i;
        this.mascota.nombre = "Fido" + i;
        this.mascota.numeroPatas = i;
        this.edad = i;
    }

    /**
     * Método para que al meter esta clase en un System.out.println() salga algo
     * legible.
     */
}
```

```
public String toString() {  
    return nombre + " "  
        + apellido + " de "  
        + edad + " años tiene como mascota a "  
        + mascota.nombre + " de "  
        + mascota.numeroPatas + " patas.";  
}
```

La aplicación principal tendrá en el método main() las llamadas a las funciones para la creación del fichero para escritura, la de añadir datos al fichero y la de lectura del fichero.

```
public static void main(String[] args) {  
    escribeFichero("./src/main/resources/mascotas.dat");  
    anyadeFichero("./src/main/resources/mascotas.dat");  
    leeFichero("./src/main/resources/mascotas.dat");  
}
```

Vamos a detallar cada uno de los métodos. Primero abrimos el fichero para escritura y escribimos a 5 personas. En este caso utilizamos la clase **ObjectOutputStream**. Fíjate cómo el fichero se cierra.

```
/**  
 * Escribe en el fichero que se le pasa y empezando desde cero, 5 objetos  
 * de la clase Persona.  
 *  
 * @param fichero Path completo del fichero que se quiere escribir  
 */  
public static void escribeFichero(String fichero) {  
    try {  
        ObjectOutputStream oos = new ObjectOutputStream(  
            new FileOutputStream(fichero));  
        for (int i = 0; i < 5; i++) {  
            // ojo, se hace un new por cada Persona. El new dentro  
            // del bucle.  
            Persona p = new Persona(i);  
            oos.writeObject(p);  
        }  
        oos.close();  
        System.out.println("cierro fichero");  
    } catch (Exception e) {  
        e.printStackTrace();  
    }  
}
```

A continuació, añadiremos más personas al fichero, para ello necesitamos la clase **MiObjectOutputStream.java**. Esta clase es igual que **ObjectOutputStream** (que existe por defecto), pero se le ha sobrescrito el método que escribe las cabeceras (**writeStreamHeader()**) para que nos las añada.

```
public class MiObjectOutputStream extends ObjectOutputStream {
    /**
     * Constructor que recibe OutputStream
     */
    public MiObjectOutputStream(OutputStream out) throws IOException {
        super(out);
    }
    /**
     * Constructor sin parámetros
     */
    protected MiObjectOutputStream() throws IOException, SecurityException {
        super();
    }
    /**
     * Redefinición del método de escribir la cabecera para que no haga nada.
     */
    protected void writeStreamHeader() throws IOException {
    }
}
```

Ahora ya podemos añadir objetos al fichero:

```
/**
 * Añade al final del fichero que se le pasan 5 objetos de la clase Persona.
 *
 * @param fichero Path completo del fichero
 */
public static void anyadeFichero(String fichero) {
    try {
        // Se usa un ObjectOutputStream que no escriba una cabecera en
        // el stream.
        MiObjectOutputStream oos = new MiObjectOutputStream(
            new FileOutputStream(fichero, true));
        // Se hace el new fuera del bucle, sólo hay una instancia de persona.
        // Se debe usar entonces writeUnshared().
        Persona p = new Persona(5);
        for (int i = 5; i < 10; i++) {
            p.setPersona(i); // Se rellenan los datos de Persona.
            oos.writeUnshared(p);
        }
        oos.close();
    }
}
```



```
        System.out.println("cierro fichero por segunda vez");
    } catch (Exception e) {
        e.printStackTrace();
    }
}
```

La función leeFichero() muestra el contenido del fichero:

```
/**
 * Se leen todas las Persona en el fichero y se escriben por pantalla.
 *
 * @param fichero El path completo del fichero que contiene las Persona.
 */
public static void leeFichero(String fichero) {
    try {
        // Se crea un ObjectInputStream
        ObjectInputStream ois = new ObjectInputStream(
            new FileInputStream(fichero));

        // Se lee el primer objeto
        Object aux = ois.readObject();

        // Mientras haya objetos
        while (aux != null) {
            if (aux instanceof Persona) {
                System.out.println(aux);
            }
            aux = ois.readObject();
        }
        ois.close();
    } catch (EOFException e1) {
        System.out.println("Fin de fichero");
    } catch (Exception e2) {
        e2.printStackTrace();
    }
}
```

Ejecutamos el programa para ver el resultado:

```
cierro fichero
cierro fichero por segunda vez
nombre0 apellido0 de 0 años tiene como mascota a Fido0 de 0 patas.
nombre1 apellido1 de 1 años tiene como mascota a Fido1 de 1 patas.
...
Fin de fichero
```

## **Problema 2**

Este problema aparece cuando serializamos un objeto a un fichero, y al leer el fichero y deserializarlo se nos produce el siguiente error:

```
run:
El proceso ha terminado
java.io.InvalidClassException: POJO.Persona; local class incompatible: stream classdesc serialVersionUID = -71170895735541816
49, local class serialVersionUID = -412656312954111922
|   at java.io.ObjectStreamClass.initNonProxy(ObjectStreamClass.java:617)
|   at java.io.ObjectInputStream.readNonProxyDesc(ObjectInputStream.java:1622)
|   at java.io.ObjectInputStream.readClassDesc(ObjectInputStream.java:1517)
|   at java.io.ObjectInputStream.readOrdinaryObject(ObjectInputStream.java:1771)
|   at java.io.ObjectInputStream.readObject0(ObjectInputStream.java:1350)
|   at java.io.ObjectInputStream.readObject(ObjectInputStream.java:370)
|   at leerfichobject.LeerFichObject.main(LeerFichObject.java:29)
BUILD SUCCESSFUL (total time: 0 seconds)
```

Lo que nos está indicando es que el objeto Persona que está almacenado en el fichero y la clase Persona que estoy intentando utilizar para leerlo no es la misma.

Para solucionar este problema utilizamos el serialVersionUID, que es un número de versión que posee cada clase Serializable, el cual es usado en la deserialización para verificar que el emisor y el receptor de un objeto serializado mantienen una compatibilidad en lo que a serialización se refiere con respecto a la clases que tienen cargadas (el emisor y el receptor).

Es importante ponerlo ya si la clase no especifica un serialVersionUID el proceso de serialización calculará un serialVersionUID por defecto, basándose en varios aspectos de la clase, de forma que si la clase sufre algún cambio se volverá a calcular el serialVersionUID. Es muy recomendable que se declare un serialVersionUID en las clases serializables, ya que el cálculo del serialVersionUID es muy sensible a detalles de la clase, los cuales pueden variar entre compiladores, es decir, si trabajamos serializando/deserializando objetos y trabajamos con distintos compiladores Java podemos llegar a obtener una InvalidClassException durante el proceso de deserialización debido a discrepancias entre los serialVersionUID calculados por cada compilador. Por eso para garantizar un serialVersionUID que sea indiferente a la implementación del compilador es altamente recomendado declarar un valor explícito del serialVersionUID (de tipo long) y de ser posible que tenga el modificador de acceso private para que afecte únicamente a la clase que lo ha declarado y no a las clases hijas (subclases) que hereden de ella, forzando de alguna manera a cada clase hija a declarar su propio serialVersionUID.

## 6. RECOMENDACIONES

Algunas recomendaciones para trabajar con ficheros en Java:

- Los ficheros (flujos) siempre han de ser explícitamente cerrados (método `close()`) después de ser utilizados.
- En particular, un fichero sobre el que se ha escrito, debe ser cerrado antes de poder ser abierto para lectura (para garantizar la escritura de datos).
- Si el programador no cierra de forma explícita el fichero, Java lo hará, pero:
  - Si el programa termina anormalmente (se va la luz!) y el flujo de salida usaba buffering, el fichero puede estar incompleto o corrupto.
- Es importante gestionar las excepciones en los procesos de E/S:
  - Ficheros inexistentes (`FileNotFoundException`).
  - Fallos de E/S (`IOException`).