**Ficheros:**

El paquete java.io contiene todas las clases para trabajar con ficheros.

En java existen muchas clases para trabajar con ficheros:

**FileOutputStream**, **FileWriter**, **PrintStream**, etc. Todas hacen lo mismo, escribir en un fichero. La diferencia entre ellas suelen ser los métodos que tienen para escribir en el fichero o algunos aspectos internos, como si usan buffer o no. Algunas de ellas sólo tienen métodos para escribir bytes, otras permiten escribir String, etc.

Para leer un fichero nos pasa algo parecido. Tenemos un montón de clases que leen ficheros **FileInputStream**, **FileReader**, **InputStreamReader**, **BufferedReader.**

Las clases **FileReader y FileWriter** definen flujos para leer y escribir caracteres en un flujo sin ningún proceso más. Igualmente las clases

**FileOutputStream y FileInputStream** permiten leer y escribir bytes en un flujo.

Las clases **DataInputStream y DataOutputStream,**

**ObjectOutputStream y ObjectInputStream** nos sirven de filtros para convertir bytes y hacer una lectura o escritura más fácil.

Las clases [BufferedWriter](http://w3api.com/wiki/Java:BufferedWriter) **y BufferedReader** nos permiten crear un buffer intermedio para agrupar la información leída o escrita en un flujo.

**clase FileInputStream y FileOutputStream:**

# Heredan de la clase InputStream.

**FileOutputStream:** permite escribir bytes en un fichero.

**FileIntputStream**: permite leer bytes de un fichero.

**FileOutputStream**(String nombrefichero)

**FileInputStream** (String nombrefichero)

Crean un flujo para leer o escribir bytes a bytes de un fichero. Va leyendo 8 bits, luego otros 8 etc. Con lo cual si queremos leer un entero, tendremos que leer 4 veces (4 bytes), para un long 8 veces, etc.

De esta forma tendremos que controlar el número de veces que leemos para cada dato.

Existen unos filtros que se encargan de transformar cada grupo de bytes, en el tipo de dato deseado:

**Clase ObjectOutputStream y ObjectInputStream :**

Estos filtros me permiten convertir esos bytes en objetos y al revés, objetos en una serie de bytes. Pues de esa forma podremos recuperar el objeto entero no cada uno de sus componentes.

**ObjectOutputStream**: proporciona un filtro que permite convertir un objeto (atributos de dicho objeto, menos los estáticos) en una secuencia de bytes. Esta secuencia de bytes las escribiremos a continuación en el fichero por medio de los flujos **FileOutputStream**.

**ObjectInputStream** convierte una secuencia de bytes en un objeto de una clase.

La secuencia de bytes cogida con el flujo **FileInputStream**, mediante el filtro **ObjectInputStream** la convertiremos en los atributos de un objeto de una clase determinada.

Debemos indicar en la clase que sus objetos los vamos a serializar, implementando la interface Serializable.

Java ha añadido una interesante faceta al lenguaje denominada serialización de objetos que permite convertir cualquier objeto cuya clase implemente el interface Serializable en una secuencia de bytes que pueden ser posteriormente leídos para restaurar el objeto original. Esta característica se mantiene incluso a través de la red, por lo que podemos crear un objeto en un ordenador que corra bajo Windows, serializarlo y enviarlo a través de la red a una estación de trabajo que corra bajo UNIX donde será correctamente reconstruido. No tenemos que preocuparnos, en absoluto, de las diferentes representaciones de datos en los distintos ordenadores.

Serializable es una interface vacía, solo la utilizamos para identificar clases cuyo objeto se pueden serializar, no hay que redefinir ningún método. No tenemos que escribir ningún otro método. El método *defaultWriteObject* de la clase *ObjectOutputStream* realiza la serialización de los objetos de una clase. Este método escribe en el flujo de salida todo lo necesario para reconstruir dichos objetos:

Public class Cpersona implements Serializable {

Cuerpo de la clase

}

Constructores:

**FileOutputStream** flujofich = **new FileOutputStream** (“datos.dat”);

**ObjectOutputStream** filtro= **new ObjectOutputStream** (flujofich);

Métodos:

Grabar en fichero un objeto: Flujofich.**writeObject**(objeto);

Eje.

flujofich.**writeObject** (new Cpersona(nom, dir, tel));

Leer del fichero un objeto:

objeto =(ClaseObjeto)flujoobj.**readObject**();

Eje.

Cpersona persona;

Persona = (Cpersona) flujoobj.**readObject**();

**Seriar objetos que referencian a objetos:**

Cuando en un fichero se escriben objetos que hacen referencia a otros objetos, todos los objetos accesibles desde el primero, deben ser escritos en el mismo proceso para mantener así la relación existente. Este proceso es llevado a cabo por el método **writeObject**() automáticamente.

Para guardar un objeto que tenga como variable de instancia un array de objetos (array de referencia a objetos) debemos seguir el mismo proceso anterior.

Creamos el objeto:

**ArrayList** lista

Definimos una variable para ver si la lista ha sido modificada o no:

**boolean** modifica= false;

Definimos una variable File:

**File** fichero = new **File**(“ficheroDatos.dat”);

Comprobamos si existe el fichero:

 if (fichero.exists()) {

//Si existe creamos un objeto con los datos guardados en el fichero:

**ObjectInputStream** flujoobj = new **ObjectInputStream** (new **FileInputStream** (fichero)); o “ficheroDatos.dat”

lista = (ArrayList) flujoobj.**readObject**();

flujoobj.**close**();

}

else

//Si no existe creamos un objeto Vector nuevo y vacío:

lista= new **ArrayList**();

A continuación se trabaja con el array igual que en los otros ejercicios.

Cuando termine el programa tendremos que volver a copiar el array en el disco para salvarlo:

Si se ha modificado la lista:

**ObjectOutputStream** flujoobjE = new **ObjectOutputStream** ( new **FileOutputStream** (“ficheroDatos.dat”));

flujoobjE.**writeObject**(lista);

flujoobj.**close**();

Si lo que queremos guardar son los **datos campo a campo**, utilizaremos los filtros: **DataInputStream y DataOutputStream**

Por lo tanto:

* Creamos el flujo para el filtro:

**FileOutputStream** flujofich = new **FileOutputStream**  ( “prueba.dat”)

* Creamos el filtro a utilizar y lo asociamos con el flujo:

**DataOutputStream** filtro = new **DataOutputStream** (flujofich);

* Escribimos los datos: leemos los datos

filtro.**writeInt**(x); x=filtro.**readInt**();

filtro.**writeFloat**(y); y=filtro.**readFloat**();

filtro.**writeDouble**(d); d=filtro.**readDouble**();

filtro.**writeUtf**(cadena); cadena=filtro**.readUTF**();

* Cerramos el fichero y el flujo por este orden**:**

Filtro.close();

Flujo.close();

**Excepciones:**

Throws FileNotFounException, IOException:

Antes de escribir o leer:

Try {

Leer o escribir bloque de código a examinar y que

Puede producir excepciones o errores.

} catch ( FileNotFoundException excfnf) tipo de error

{ System.out.println (“ fichero no encontrado” + excfnf); }

catch (IOException excio);

{ System.out.println (error de entrada/salida” + excio); }

finally { se ejecuta siempre al finalizar

filtro.close();

flujo.close(); }

}

Dependiendo del error que se produzca escogerá siempre el catch asociado a ese error.

De esta forma vamos guardando en el fichero cada uno de los datos con su tipo, tenemos que recuperar esos datos en el mismo orden que los guardamos y con el mismo formato.

Si queremos abrir el flujo para escribir a continuación de lo que ya tenemos en el fichero pondremos:

**FileOutputStream** flujofich = new **FileOutputStream**  ( “prueba.dat”, true);

**Clase File:**

Si queremos realizar programas que sirvan para cualquier fichero, podemos definir un objeto de la clase File. Este objeto representa el nombre de un fichero o de un directorio.

Sus métodos permiten interrogar al sistema sobre todas las características de ese fichero o ese directorio.

**Constructores:**

public **File** (String ruta-completa);

public **File** (String ruta, String nombre\_fichero);

**Métodos:**

**File** fichero = new **File**(“proyecto\\texto.dat”);

fichero.**getName**(); devuelve nombre fichero, texto.dat

fichero.**getParent**(); devuelve directorio padre proyecto

fichero.**getPath**(); devuelve ruta relativa proyecto\texto.dat

fichero.**getAbsolutePath**() devuelve ruta absoluta c:\java\proyecto\texto.dat

fichero.**exists**(); devuelve true si el fichero existe.

fichero.**canWriter**(); devuelve true si se puede escribir en el fichero

fichero.**canRead**(); devuelve true si se puede leer del fichero

fichero.**isFile**(); true si se trata de un fichero válido.

fichero.**length**(); tamaño del fichero.

fichero.**mkdir**(); crea el directorio especificado por el objeto File.

fichero.**delete**(); borra el fichero o el directorio especificado por el

objeto File.

fichero.**renameTo**(objeto); renombra el fichero especificado por el objeto File que recibe este mensaje, con el nombre especificado por el objeto File pasado como parámetro.

Ejemplo:

String nombrefich;

**File** fichero = null;

Try {

Do {

**System.out.println**(“nombre del fichero”);

nombrefich = flujo.**readLine**();

fichero = new **File** (nombrefich);

}while(!fichero.**exists**());

**FileOutputStram** flujofich = new **FileOutputStream** (“fichero”);

**DataOutputStream** filtro = new **DataOutputStream**(flujofich);

filtro.**writeInt**(x);

filtro.**writeUTF**(caden);

**Ficheros de texto**

Si usamos sólo **FileInputStream, FileOuputStream, FileReader o FileWriter,** cada vez que hagamos una lectura o escritura, se hará físicamente en el disco duro. Si escribimos o leemos pocos caracteres cada vez, el proceso se hace costoso y lento, con muchos accesos a disco duro.

Los **BufferedReader, BufferedInputStream, BufferedWriter , PrintWriter y BufferedOutputStream** añaden un buffer intermedio. Cuando leamos o escribamos, esta clase controlará los accesos a disco.

**Writer** : escritura de caracteres. Es una clase abstracta, por eso usamos la clase **FileWriter** que heredan de ella.

Para elegir una de ellas, simplemente hay que ver qué tenemos, un **File** en nuestro caso y qué necesitamos, algún método que nos permita escribir **String**, esto lo podemos hacer con los métodos de la clase **PrintWriter**.

**File** fichDatos= new **File**("c:/prueba.txt");

**PrintWriter** escribir= new **PrintWriter**(new **FileWriter**(fichDatos));

escribir.**print**(areaTexto.getText()); o escribir.**print(“**Hola que tal”);  
escribir.**close();**

Abrimos el fichero haciendo un new **PrintWriter(),** con **print()** o con **println**() escribimos el texto obtenido de un JTextArea o el String y cerramos el fichero.

**PrintWriter** tiene además muchos métodos que permiten escribir en formato texto plano, números, cadenas, etc.

Otra forma sería usando las **clases** [**BufferedWriter**](http://w3api.com/wiki/Java:BufferedWriter) **y** [**FileWriter**](http://w3api.com/wiki/Java:FileWriter):

[**BufferedWriter**](http://w3api.com/wiki/Java:BufferedWriter) escribir = new [**BufferedWriter**](http://w3api.com/wiki/Java:BufferedWriter)(new [**FileWriter**](http://w3api.com/wiki/Java:FileWriter)(fichero));

for (int x=0;x<10;x++)

escribir.write("Fila numero " + x + "\n");

**Leer un fichero**

Leer un archivo de texto en java no es muy complicado, en resumen es un proceso muy similar a leer texto escrito del teclado, y para hacerlo necesitaremos hacer uso de las siguientes clases, la clase **File**, **FileReader y BufferedReader.**

Igual que al escribir, tenemos que ver que es lo que tenemos (un Fichero) y que es lo que necesitamos, algún método que nos permita leer **String** y elegir la clase que nos interese.

No hay una clase que admita File y tenga métodos para leer String. Mirando la [API](http://java.sun.com/j2se/1.5.0/docs/api), vemos que el método **readLine**() que nos interesa y que es capaz de leer una línea y dárnosla como String está en la clase **BufferedReader**. Desgraciadamente esta clase admite en su constructor un **Reader** y no un **File** que es lo que tenemos.

Debemos buscar una clase que herede de **Reader** y que admita un File. Buscando, buscando, encontramos que **FileReader** cumple con estos requisitos.

**BufferedReader** leer = new **BufferedReader**(new **FileReader**(fichero));

**BufferedReader** va leyendo de línea en línea. No lee los retornos de carro. Debemos hacer un bucle para ir leyendo. Puede ser similar a este

String linea = leer.**readLine**();  
**while** (linea != null)  
{  
   // Aquí lo que tengamos que hacer con la línea puede ser esto  
   areaTexto.append(linea);  
   linea = leer**.readLine**();  
}

leer.**close();**

Por supuesto, no se nos debe olvidar cerrar el fichero cuando terminemos

De esta misma forma podemos leer ficheros escritos desde el bloc de notas o desde Word pero guardado como texto sin formato .txt.

**Ejemplo escribir líneas en un fichero y luego leerlas**

package ficherotexto;

import java.io.\*;

public class Main {

public static void main(String[] args) throws IOException{

File fichero=new File("texto.txt");

PrintWriter escribir = new PrintWriter(new FileWriter(fichero));

for(int i=0;i<10;i++)

escribir.print("linea"+i+"\n");

escribir.close();

BufferedReader leer = new BufferedReader(new

FileReader(fichero));

String linea = leer.readLine();

while (linea != null)

{

System.out.println(linea);

linea = leer.readLine();

}

leer.close();

}

**Ejemplo de lectura de un fichero con *Scanner***

Supongamos que tenemos un fichero en el que en cada línea hay los datos de una persona. Pueden ser un id, un nombre y una edad, separados por comas y quizás espacios. Algo como lo siguiente

1 , Pedro , 33

2, Juan, 44

4, Antonio, 55

Vamos a hacer y explicar un pequeño programa en java usando *Scanner* que nos permita leer estos datos.

En primer lugar, creamos un [File](http://chuwiki.chuidiang.org/index.php?title=La_clase_File) con el contenido del fichero y después una instancia de *Scanner* pasándole ese File. Por supuesto, al terminar de leer nuestro fichero, debemos cerrarlo

File f = new File("fichero.txt");

Scanner s;

try {

s = new Scanner(f);

//

// Aquí la lectura del fichero

//

s.close();

} catch (FileNotFoundException e) {

e.printStackTrace();

}

Para la lectura del fichero, bastará de momento con un bucle para ir leyendo línea a línea. Para ello, podemos usar el método hasNextLine() que nos indica si hay o no una línea más que leer, y el método nextLine() que nos la devuelve

while (s.hasNextLine()) {

String linea = s.nextLine();

//

// Aquí el tratamiento de la línea

//

}

Para tratar la línea y sacar los tres campos que hay en ella, podemos usar nuevamente otra instancia de la clase *Scanner*. El delimitador para los campos será una coma, precedida o no de uno o más espacios y seguida o no de uno o más espacios. Eso, usando [Expresiones Regulares en Java](http://chuwiki.chuidiang.org/index.php?title=Expresiones_Regulares_en_Java) se expresa así "\\s\*,\\s\*", donde \\s indica un espacio blanco y con asterisco detrás \\s\* indica cero o más espacios en blanco. Por tanto, el código para recoger los tres campos, puede ser como este

Scanner sl = new Scanner(linea);

sl.useDelimiter("\\s\*,\\s\*");

System.out.println(sl.next());

System.out.println(sl.next());

System.out.println(sl.next());

Y si ahora ponemos el programa completo

package com.chuidiang.ejemplos.file\_scanner;

import java.io.File;

import java.io.FileNotFoundException;

import java.util.Scanner;

public class FileScanner {

public static void main(String[] args) {

File f = new File("fichero.txt");

Scanner s;

try {

s = new Scanner(f);

while (s.hasNextLine()) {

String linea = s.nextLine();

Scanner sl = new Scanner(linea);

sl.useDelimiter("\\s\*,\\s\*");

System.out.println(sl.next());

System.out.println(sl.next());

System.out.println(sl.next());

}

s.close();

} catch (FileNotFoundException e) {

e.printStackTrace();

}

}

}

**Si utilizamos interface gráfica y JTextArea**

void botoEscribir\_actionPerformed(ActionEvent e) {

fichero=new File("fichArea.txt");

try{

PrintWriter escribir = new PrintWriter(new FileWriter(fichero));

escribir.print(txtArea1.getText());

escribir.close();

}catch (IOException ee){}

}

void botonLeerArea\_actionPerformed(ActionEvent e) {

fichero=new File("fichArea.txt");

try{

BufferedReader reader = new BufferedReader(new FileReader(fichero));

String dato="";

String linea= reader.readLine();

while(linea!=null){

dato=dato + linea+"\n";

System.out.println(linea);

linea=reader.readLine();

}

txtArea1.setText(dato);

}catch (IOException ee){}

}

void botonBorrar\_actionPerformed(ActionEvent e) {

txtArea1.setText("");

}

**Otra forma de escribir en el JTextArea:**

void botonLeerArea\_actionPerformed(ActionEvent e) {

fichero=new File("fichArea.txt");

try{

BufferedReader reader = new BufferedReader(new FileReader(fichero));

String linea= reader.readLine();

while(linea!=null){

txtArea1.append(linea+"\n");

linea=reader.readLine();

}

}catch (IOException ee){}

}

#### La clase JFileChooser:

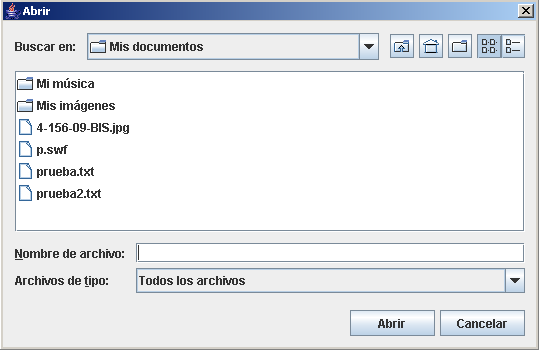
Vamos a ver un ejemplo de un editor de texto sencillo, dos de las acciones que debemos implementar en nuestro menú son las de salvar el texto en fichero y leer el texto de un fichero. En ambos casos hay que darle oportunidad al usuario de elegir el fichero bien para salvar, bien para leer.

La clase **JFileChooser** muestra directamente una ventana que permite navegar por los directorios y elegir un fichero. Un uso simple de esta clase es realmente sencillo.

Para pedir un fichero para abrirlo y leerlo basta hacer dos líneas de código

JFileChooser fileChooser = new JFileChooser();  
int seleccion = fileChooser.showOpenDialog(areaTexto);

Veremos el parámetro **areaTexto** un poco más abajo. Por lo demás, listo, esto abre una ventana para navegar por los directorios y elegir un fichero similar a la de la figura



Cuando el usuario termine, pulsará "Abrir" o "Cancelar". En el entero **seleccion** se nos guardará el botón que ha apretado. Sólo debemos comprobar si es el botón de "Abrir" para pedirle al **JFileChooser** qué fichero ha elegido el usuario y empezar a trabajar con él.

if (seleccion == JFileChooser.APPROVE\_OPTION)  
{  
   File fichero = fileChooser.getSelectedFile();  
   // y a trabajar con fichero ....  
}

Si **seleccion** vale **JFileChooser.APPROVE\_OPTION** es que el usuario ha pulsado "Abrir", así que pedimos el fichero seleccionado con **getSelectedFile()** y ... ¡¡¡ a trabajar !!!

¿qué es eso de pasarle el areaTexto en el parámetro?. La explicación es simple. **JFileChooser** abre una ventana que se mostrará junto a la ventana de nuestro editor. Nos gustaría que esta ventana de selección de fichero no se vaya detrás y quede oculta por la ventana del editor. La ventana de elegir fichero debería estar siempre delante y visible sobre el editor. La forma de conseguir esto es que **JFileChooser** sepa cual es esta ventana delante de la cual debe estar siempre. Para saber cual es esa ventana, basta con pasarle en el parámetro cualquier componente de dicha ventana. En nuestro ejemplo hemos decidido pasarle el **JTextArea** de nuestro editor, pero podíamos pasarle perfectamente el **JMenuBar** o el **JFrame** principal.

El **JTextArea** es una buena elección por un segundo motivo. Este código estará en las acciones de salvar y cargar texto de un fichero. El texto que se va a guardar o el que se lea se cojera o meterá en el **JTextArea**.

La clase **AccionSalvar** y **AccionCargar** deben tener accesible el **JTextArea** para leer de él el texto que tienen que guardar en fichero o para meter en él el texto que han leído del fichero. Puesto que tienen que tener el **JTextArea** de todas maneras, lo usaremos como parámetro del **JFileChooser**.

Para elegir un fichero para salvar, el código es exactamente igual, pero cambia el método para visualizar el diálogo.

JFileChooser fileChooser = new JFileChooser();  
int seleccion = fileChooser.showSaveDialog(areaTexto);

La única diferencia es que cambia la etiqueta del botón "Abrir" por "Guardar".