TEMA 6

INDICE

1.- Introducción. 4
1.1.- Excepciones. 4
2.- Concepto de flujo. 6
Sockets 6
Qué es un socket? 6
Tipos de socket 6
Clase Socket 7
Servicios de la clase Socket 7
Clase ServerSocket 7
Servicios de la clase ServerSocket 7
Comunicación entre cliente-servidor 8
Clase DatagramSocket 8
Servicios de la clase DatagramSocket 8
Clase DatagramPacket 8
Threads 9
¿Qué es un thread? 9
¿Qué ganamos? 9
Atributos de un thread en JAVA 9
Clase Thread 9
Servicios de la clase Thread 10
Estados de un Thread 10
Sincronización 10
Ejemplo:Un chat 11
Servidor.java 11
Flujo.java 11
Cliente.java 12
3.- Clases relativas a flujos. 14
3.1.- Ejemplo comentado de una clase con flujos. 15
4.- Flujos. 16
4.1.- Flujos predefinidos. Entrada y salida estándar. 16
4.2.- Flujos predefinidos. Entrada y salida estándar. Ejemplo. 17
4.3.- Flujos basados en bytes. 18
4.4.- Flujos basados en caracteres. 19
4.5.- Rutas de los ficheros. 20
5.- Trabajando con ficheros. 21
5.1.- Escritura y lectura de información en ficheros. 21
5.2.- Ficheros binarios y ficheros de texto (I). 22
5.2.1.- Ficheros binarios y ficheros de texto (II). 23
5.3.- Modos de acceso. Registros. 25
5.4.- Acceso secuencial. 25
5.5.- Acceso aleatorio. 27
6.- Aplicaciones del almacenamiento de información en ficheros. 29
7.- Utilización de los sistemas de ficheros. 30
7.1.- Clase File. 30
7.2.- Interface FilenameFilter. 31
7.3.- Creación y eliminación de ficheros y directorios. 32
8.- Almacenamiento de objetos en ficheros. Persistencia. Serialización. 34
8.1.- Serialización: utilidad. 34

Almacenando datos.

Caso práctico

Ada está repasando los requisitos de la aplicación informática que están desarrollando para la clínica veterinaria.

En particular, ahora mismo se está centrando en estudiar las necesidades respecto al almacenamiento de datos. Ada piensa que hay ciertas partes de la aplicación que no necesitan una base de datos para guardar los datos, y sería suficiente con emplear ficheros. Por ejemplo, para guardar datos de configuración de la aplicación.

Tras repasar, se reúne con María y Juan para planificar adecuadamente el tema de los ficheros que van a usar en la aplicación, ya que es un asunto muy importante, que no deben dejar aparcado por más tiempo.

Precisamente Antonio, que cada vez está más entusiasmado con la idea de estudiar algún ciclo, de momento, está matriculado y cursando el módulo de Programación, y está repasando para el examen que tiene la semana que viene, uno de los temas que le "cae" es precisamente el de almacenamiento de información en ficheros.

1.- Introducción.

Cuando desarrollas programas, en la mayoría de ellos los usuarios pueden pedirle a la aplicación que realice cosas y pueda suministrarle datos con los que se quiere hacer algo. Una vez introducidos los datos y las órdenes, se espera que el programa manipule de alguna forma esos datos, para proporcionar una respuesta a lo solicitado.

Además, normalmente interesa que el programa guarde los datos que se le han introducido, de forma que si el programa termina, los datos no se pierdan y puedan ser recuperados en una sesión posterior. La forma más normal de hacer esto es mediante la utilización de ficheros, que se guardarán en un dispositivo de memoria no volátil (normalmente un disco).

Por tanto, sabemos que el almacenamiento en variables o vectores (arrays) es temporal, los datos se pierden en las variables cuando están fuera de su ámbito o cuando el programa termina. Las computadoras utilizan ficheros para guardar los datos, incluso después de que el programa termine su ejecución. Se suele denominar a los datos que se guardan en ficheros datos persistentes, porque existen, persisten más allá de la ejecución de la aplicación. Los ordenadores almacenan los ficheros en unidades de almacenamiento secundario como discos duros, discos ópticos, etc. En esta unidad veremos cómo hacer con Java estas operaciones de crear, actualizar y procesar ficheros.

A todas estas operaciones, que constituyen un flujo de información del programa con el exterior, se les conoce como Entrada/Salida (E/S).

Distinguimos dos tipos de E/S: la E/S estándar que se realiza con el terminal del usuario y la E/S a través de ficheros, en la que se trabaja con ficheros de disco.

Todas las operaciones de E/S en Java vienen proporcionadas por el paquete estándar del API de Java denominado java.io que incorpora interfaces, clases y excepciones para acceder a todo tipo de ficheros.

El contenido de un archivo puede interpretarse como campos y registros (grupos de campos), dándole un significado al conjunto de bits que en realidad posee.

Para saber más

A continuación puedes ampliar tus conocimientos sobre Entrada y Salida en general, en el mundo de la informática. Verás que es un basto tema lo que abarca.

Entrada y Salida. http://exa.unne.edu.ar/depar/areas/informatica/SistemasOperativos/SO5.htm

1.1.- Excepciones.

Cuando se trabaja con archivos, es normal que pueda haber errores, por ejemplo: podríamos intentar leer un archivo que no existe, o podríamos intentar escribir en un archivo para el que no tenemos permisos de escritura. Para manejar todos estos errores debemos utilizar excepciones. Las dos excepciones más comunes al manejar archivos son:

- ✓ `FileNotFoundException`: Si no se puede encontrar el archivo.
- ✓ `IOException`: Si no se tienen permisos de lectura o escritura o si el archivo está dañado.

Un esquema básico de uso de la captura y tratamiento de excepciones en un programa, podría ser este, importando el paquete `java.io.IOException`:

```
importjava.io.IOException;

// ...

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

```
try {
  // Se ejecuta algo que puede producir una excepción
} catch (FileNotFoundException e) {
// manejo de una excepción por no encontrar un archivo
    } catch (IOException e) {
  // manejo de una excepción de entrada/salida
    } catch (Exception e) {
  // manejo de una excepción cualquiera
    } finally {
  // código a ejecutar haya o no excepción
}
}
```

Señala la opción correcta:

- ○ Java no ofrece soporte para excepciones.
- ○ Un campo y un archivo es lo mismo.
- ● Si se intenta abrir un archivo que no existe, entonces saltará una excepción.
- ○ Ninguna es correcta.

En el siguiente enlace hay un manual muy interesante de Java. Puedes consultar desde el índice, más información sobre las excepciones en Java.

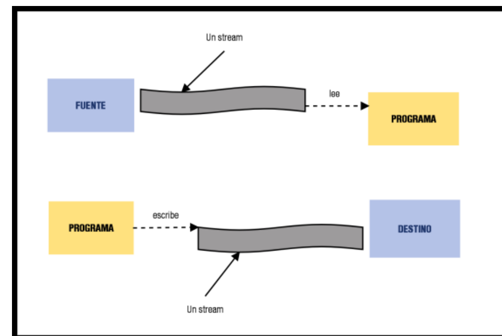
Excepciones en Java.http://www.webtaller.com/manual-java/indice_manual_java.php

2.- Concepto de flujo.

Caso práctico

Antonio *está estudiando un poco antes de irse a dormir. Se ha tomado un vaso de leche con cacao y está repasando el concepto de flujo. Entenderlo al principio, cuando lo estudió por primera vez, le costó un poco, pero ya lo entiende a la perfección y piensa que si le sale alguna pregunta en el examen de la semana que viene, sobre esto, seguro que la va a acertar.*

La clase `Stream` representa un flujo o corriente de datos, es decir, un conjunto secuencial de bytes, como puede ser un archivo, un dispositivo de entrada/salida (en adelante E/S), memoria, un conector TCP/IP (Protocolo de Control de Transmisión/Protocolo de Internet), etc. Cualquier programa realizado en Java que necesite llevar a cabo una operación de entrada salida lo hará a través de un **stream**.



Un flujo es una abstracción de aquello que produzca o consuma información. Es una entidad lógica.

Las clases y métodos de E/S que necesitamos emplear son las mismas **independientemente del dispositivo con el que estemos actuando**, luego, el núcleo de Java, sabrá si tiene que tratar con el teclado, el monitor, un sistema de archivos o un socket de red liberando al programador de tener que saber con quién está interactuando.

La vinculación de un flujo al dispositivo físico la hace el sistema de entrada y salida de Java.

En resumen, será el flujo el que tenga que comunicarse con el sistema operativo concreto y "entendérselas" con él. De esta manera**, no tenemos que cambiar absolutamente nada en nuestra aplicación**, que va a ser independiente tanto de los dispositivos físicos de almacenamiento como del sistema operativo sobre el que se ejecuta. Esto es primordial en un lenguaje multiplataforma y tan altamente portable como Java**.**

Señala la opción correcta:

- (x) La clase `Stream` puede representar, al instanciarse, a un archivo.
- ( ) Si programamos en Java, hay que tener en cuenta el sistema operativo cuando tratemos con flujos, pues varía su tratamiento debido a la diferencia de plataformas.
- ( ) La clase keyboard es la clase a utilizar al leer flujos de teclado.
- ( ) La vinculación de un flujo al dispositivo físico la hace el hardware de la máquina.

Para saber más

En la siguiente presentación puedes aprender más sobre sockets en Java.

Sockets

Qué es un socket?

- ✓ **Definición**: Un socket es un punto final de un enlace de comunicación de dos vías entre dos programas que se ejecutan a través de la red.
- ✓ El cliente y el servidor deben ponerse de acuerdo sobre el protocolo que utilizarán.

Tipos de socket

Hay dos tipos de socket:

- ✓ Orientado a conexión:

- → Establece un camino virtual entre servidor y cliente, fiable, sin pérdidas de información ni duplicados, la información llega en el mismo orden que se envía.
- → El cliente abre una sesión en el servidor y este guarda un estado del cliente.
  - El cliente utiliza la clase Socket
  - El servidor utiliza la clase ServerSocket

✓ No orientado a conexión:

- → Envío de datagramas de tamaño fijo. No es fiable, puede haber pérdidas de información y duplicados, y la información puede llegar en distinto orden del que se envía.
- → No se guarda ningún estado del cliente en el servidor, por ello, es más tolerante a fallos del sistema.
  - Tanto el cliente como el servidor utilizan la clase DatagramSocket.

Todas estas clases (Socket, ServerSocket y DatagramSocket) se encuentran en el paquete java.net

Clase Socket

✓ Constructores:

- → public Socket ()
- → public Socket (InetAddress address, int port)
- → public Socket (String host, int port)
- → public Socket (InetAddress address, int port, InetAddresslocalAddr, intlocalPort)
- → public Socket (String host, int port, InetAddresslocalAddr, intlocalPort)
  - address / localAddr: dirección IP de la máquina remota / local.
  - port / localPort: puerto de la máquina remota / local.
  - host: nombre de la máquina remota

En el caso del primer constructor crea un socket sin conexión.

Servicios de la clase Socket

✓ publicInetAddressgetInetAddress():

- → Devuelve la dirección IP de la máquina en la que estamos conectados.

✓ publicintgetPort():

- → Devuelve el puerto de la máquina remota.

✓ publicvoidclose()

- → Cierra el canal de comunicación.

✓ publicInputStreamgetInputStream():

- → Devuelve el canal de lectura del socket.

✓ publicOutputStreamgetOutputStream():

- → Devuelve el canal de escritura del socket.

Ademas JAVA proporciona dos llamadas para saber la @IP y puerto local (getLocalAddress() y getLocalPort())

Clase ServerSocket

✓ Constructores:

- → publicServerSocket (intport)
- → public ServerSocket (int port, int backlog)
- → public ServerSocket (int port, int backlog, InetAddressbindAddr)
  - port: puerto de la máquina servidora.
  - backlog: tamaño de la cola de espera, en el primero es 50.
  - bindAddr: dirección IP local que se hará pública mediante el bind.

✓ El constructor ServerSocket se encarga de hacer el bind y el listen.

Servicios de la clase ServerSocket

✓ public Socket accept():

  - → Devuelve el socket resultado de aceptar una petición, para llevar a cabo la comunicación con el cliente.
- ✓ publicvoidclose()
  - → Cierra el canal de comunicación.
- ✓ publicInetAddressgetInetAddress():
  - → Devuelve la dirección IP de la máquina local.
- ✓ publicintgetLocalPort():
  - → Devuelve el puerto de la máquina local.

Comunicación entre cliente-servidor

- ✓ Para la transmisión de datos entre cliente y servidor se utilizarán las clases DataInputStream (recibir datos) y DataOutputStream (enviar datos)
- ✓ Estas clases disponen de métodos para leer y escribir datos en el socket:
  - → read/writeBoolean
  - → read/writeChar
  - → read/writeDouble, read/writeFloat, read/writeInt, read/writeLong, read/writeShort
  - → read/writeUTF (leer/escribir cadenas de caracteres)
- ✓ Para enviar los datos se utiliza el método flush() de la clase DataOutputStream.

Clase DatagramSocket

- ✓ Constructores:
  - → publicDatagramSocket ()
  - → publicDatagramSocket (intport)
  - → public DatagramSocket (int port, InetAddressladdr)
    - port: puerto de la máquina.
    - laddr: dirección IP local que se hará pública mediante el bind.
- ✓ El constructor DatagramSocket se encarga de hacer el bind.
- ✓ El primer constructor coge un puerto libre.

Servicios de la clase DatagramSocket

  - → public void connect(InetAddress address, int port):
    - Conecta el socket a la máquina remota con la @IP address y puerto port.
  - → publicvoidclose()
    - Cierra el canal de comunicación.
  - → publicvoiddisconnect():
    - Desconecta el socket.
  - → publicInetAddressgetInetAddress():
    - Devuelve la @IP de la máquina remota.
  - → publicintgetPort():
    - Devuelve el puerto de la máquina remota.

También hay dos llamadas para saber la @IP y puerto local (getLocalAddress() y getLocalPort()).

  - → public void send(DatagramPacket p):
    - Envía un datagrama a la máquina remota, por el socket asociado.
  - → public void receive(DatagramPacket p):
    - Recibe un datagrama de otra máquina, por el socket asociado.

Clase DatagramPacket

- ✓ Constructores:
  - → public DatagramPacket (byte[] buff, int length)
    - Construye un DatagramPacket para recibir paquetes en el buffer buff, de longitud length

- public DatagramPacket (byte[] buff, int length, InetAddress address, int port)
  - Construye un DatagramPacket para envíar paquetes con datos del buffer buff, de longitud length, a la @IP address y el puerto port.
- Servicios:
  - Para la actualización y consulta de los diferentes campos de un DatagramPacket disponemos de los siguientes métodos: set/getAddress, set/getData, set/getLength, set/getPort

Threads

¿Qué es un thread?

- **Definición**: Un thread es un flujo secuencial de control dentro de un programa.
- Está definido por un contador de programa (PC) y un puntero a la pila (SP)
- Un thread no puede ejecutarse por sí mismo, ha de hacerlo dentro de un programa.
- Todos los threads de un proceso comparten los recursos (Ej: canales E/S, memoria, ...)

¿Qué ganamos?

- Conseguimos concurrencia entre procesos y también dentro de un mismo proceso.
- También podemos conseguir paralelismo si disponemos de una arquitectura multiprocesador.
- El cambio de contexto entre threads no es tan costoso al sistema como el cambio de contexto entre procesos.

Atributos de un thread en JAVA

- Un thread tiene:
  - Nombre
  - Prioridad
    - Ha de seguir la siguiente relación:
      - MIN_PRIORITY <= Prioridad <= MAX_PRIORITY
      - La prioridad por defecto es NORM_PRIORITY
    - Valores:
      - MIN_PRIORITY = 1
      - NORM_PRIORITY = 5
      - MAX_PRIORITY = 10

Clase Thread

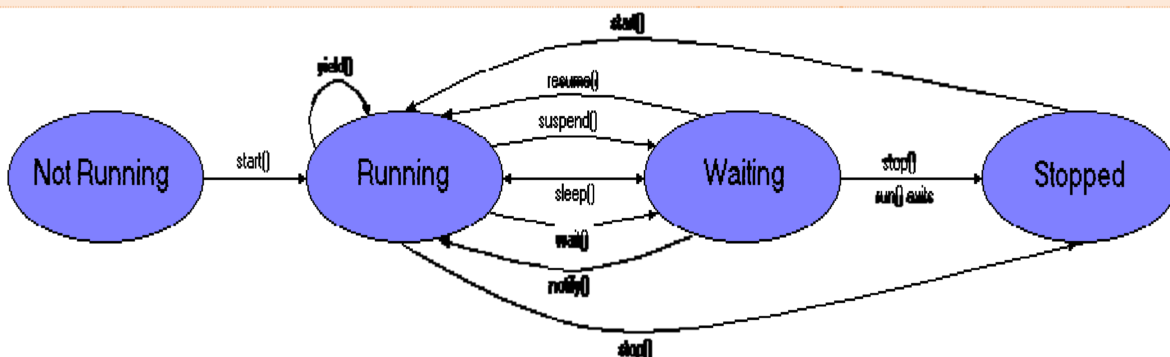
- Constructores:
  - publicThread ()
    - Crea un thread estándar.
  - publicThread (Runnable target)
    - Crea un thread en base a una clase que implementa la interfaz Runnable.
  - public Thread (Runnable target, String name)
    - Equivalente al anterior constructor dándole el nombre name al thread.
  - publicThread (Stringname)
    - Crea un thread estándar con el nombre name.
  - public Thread (ThreadGroup group, Runnable target)
    - Crea un thread en base a una clase que implementa la interfaz Runnable dentro del ThreadGroup.
  - public Thread (ThreadGroup group, Runnable target, String name)
    - Equivalente al anterior constructor dándole el nombre name al thread.
  - public Thread (ThreadGroup group, String name)
    - Crea un thread estándar con el nombre name dentro del ThreadGroup

- Cuando se crea un thread no se pone en marcha, se lo hemos de indicar explícitamente.
- Un ThreadGroup es un conjunto de threads, los cuales tienen acceso a información entre ellos.

Servicios de la clase Thread

- publicStringgetName ():
  - Devuelve el nombre que identifica al thread.
- public String setName (String name):
  - Cambia el nombre del thread.
- publicintgetPriority()
  - Devuelve la prioridad del thread.
- public void setPriority(intnewPriority)
  - Cambia la prioridad del thread
- publicThreadGroupgetThreadGroup():
  - Devuelve el grupo de donde procede el thread.
- voidrun()
  - Es el método que ejecutará un thread cuando este se ponga en marcha.
- voidstart()
  - Inicia la ejecución del thread.
- voidjoin()
  - Espera a que un thread termine su ejecución.
  - Podemos especificar el tiempo de espera máximo.
- voidsleep(long milis)
  - Bloquea el thread milis milisegundos.
  - Hay otro parámetro que permite afinar más el tiempo de bloqueo en nanosegundos.
- void stop/destroy()
  - Destruye el thread sin eliminar estructuras, esto es tarea del método join.
- staticThreadcurrentThread()
  - Devuelve el thread que se está ejecutando actualmente.
- booleanisAlive()
  - Mira si un thread no ha terminado su ejecución
- staticvoidyield()
  - Provoca el cambio de contexto entre flujos.

Estados de un Thread



Sincronización

Aquí se presenta el problema de que varios threads intenten acceder a los mismos datos y que el resultado final sea incorrecto.

- Por ejemplo: Acceso a una variable global x con valor inicial 0, y dos threads que intentan incrementar su valor en una unidad.
- Si los threads no están sincronizados un resultado posible sería que la variable x tenga el valor 1 (incorrecto)

- Solución: Para resolver el problema anterior se utiliza la etiqueta synchronized junto con el objeto que se quiere sincronizar.
  - En el caso anterior se solucionaría de la siguiente forma:
    - synchronized (this) { x++; }
      - this es la clase que contiene la variable global
  - Con esto se garantiza el acceso en exclusión mutua.
- Si queremos acceder a un método en exclusión mutua tenemos las siguientes opciones:
  - Igual que el ejemplo anterior:
    - synchronized (this) { this.incrementar() }
  - Declarando el método como sincronizado:
    - synchronizedvoid incrementar ()
    - También se pueden sincronizar varios threads mediante los métodos: wait(), notify() y notifyAll()

Ejemplo:Un chat

Servidor.java

```
import java.io.*;
import java.net.*;
importjava.util.*;
public class Servidor extends Thread{
   public static Vector usuarios = new Vector();
   public static void main (String args[]){
      ServerSocket sfd = null;
      try{
         sfd = new ServerSocket(7000);
      }catch (IOException ioe){
         System.out.println("Comunicación rechazada."+ioe);
         System.exit(1);
      }while (true){
         try{
            Socket nsfd = sfd.accept();
            System.out.println("Conexion aceptada de: "+nsfd.getInetAddress());
            Flujoflujo = new Flujo(nsfd);
            Thread t = new Thread(flujo);
            t.start();
         }catch(IOException ioe){
            System.out.println("Error: "+ioe);
         }
      }
   }
}
```

Flujo.java

```
import java.net.*;
import java.io.*;
importjava.util.*;
public class Flujo extends Thread{
   Socket nsfd;
   DataInputStreamFlujoLectura;
   DataOutputStreamFlujoEscritura;
   publicFlujo (Socket sfd){
      nsfd = sfd;
      try{
         Flujo Lectura = new DataInputStream(new BufferedInputStream(sfd.getInputStream()));
         Flujo          Escritura          =          new          DataOutputStream(new
BufferedOutputStream(sfd.getOutputStream()));
      }catch(IOExceptionioe){
         System.out.println("IOException(Flujo): "+ioe);
      }
   }
   publicvoidrun(){
      broadcast(nsfd.getInetAddress()+"> se ha conectado");
      Servidor.usuarios.add ((Object) this);
      while(true){
         try{
            String linea = FlujoLectura.readUTF();
```

```
            if (!linea.equals("")){
               linea = nsfd.getInetAddress() +”> “+ linea;
               broadcast(linea);
            }
         }
         catch(IOException ioe){
            Servidor.usuarios.removeElement(this);
            broadcast(nsfd.getInetAddress()+“> se ha desconectado");
            break;
         }
      }
   }

   public void broadcast(String mensaje){
      synchronized (Servidor.usuarios){
         Enumeration e = Servidor.usuarios.elements();
         while (e.hasMoreElements()){
            Flujo f = (Flujo) e.nextElement();
            try{
               synchronized(f.FlujoEscritura){
                  f.FlujoEscritura.writeUTF(mensaje);
                  f.FlujoEscritura.flush();
               }
            }
            catch(IOException ioe){
               System.out.println("Error: "+ioe);
            }
         }
      }
   }
}
```

Cliente.java

```
import java.io.*;
import java.net.*;
importjava.awt.*;
importjava.awt.event.*;
public class Cliente extends Frame implements ActionListener
{
static Socket sfd = null;
staticDataInputStream EntradaSocket;
staticDataOutputStream SalidaSocket;
staticTextField salida;
staticTextArea entrada;
  String texto;
publicCliente()
  {
setTitle("Chat");
setSize(350,200);
salida = new TextField(30);
salida.addActionListener(this);

entrada = new TextArea();
entrada.setEditable(false);

add("South",salida);
add("Center", entrada);
setVisible(true);
  }
public static void main(String[] args){
Cliente cliente = new Cliente();
try
    {
sfd = new Socket("192.168.0.1",7000);
EntradaSocket = new DataInputStream(new BufferedInputStream(sfd.getInputStream()));
SalidaSocket = new DataOutputStream(new BufferedOutputStream(sfd.getOutputStream()));
}
catch (UnknownHostExceptionuhe)
    {
System.out.println("No se puede acceder al servidor.");
System.exit(1);
    }
catch (IOException ioe)
{
System.out.println("Comunicación rechazada.");
```

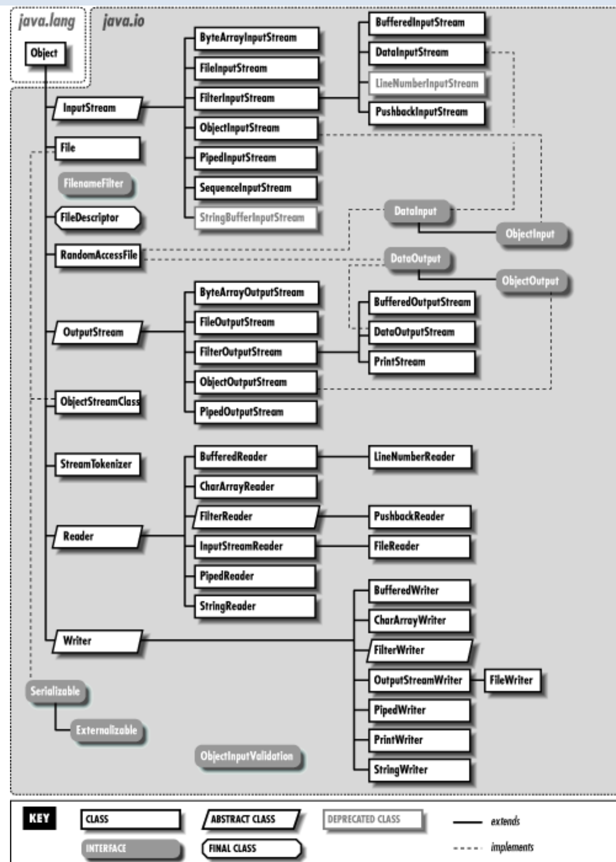
```
System.exit(1);
    }
while (true)
    {
try
      {
        String linea = EntradaSocket.readUTF();
entrada.append(linea+"\n");
      }
catch(IOException ioe)
      {
   System.exit(1);
      }
    }
  }

public void actionPerformed (ActionEvent e)
  {
texto = salida.getText();
salida.setText("");
try
{
SalidaSocket.writeUTF(texto);
SalidaSocket.flush();
}
catch (IOException ioe)
    {
System.out.println("Error: "+ioe);
    }
  }

publicbooleanhandleEvent(Event e)
  {
if ((e.target == this) && (e.id == Event.WINDOW_DESTROY)){
   if (sfd != null){
      try{
         sfd.close();
      }catch (IOException ioe)       {
         System.out.println("Error: "+ioe);
      }
      this.dispose();
      }
    }
return true;
  }
}
```

3.- Clases relativas a flujos.

Caso práctico

Otro aspecto importante que ***Ada*** *trata con* ***María*** *y* ***Juan****, acerca de los ficheros para la aplicación de la clínica, es el tipo de ficheros a usar. Es decir, deben estudiar si es conveniente utilizar ficheros para almacenar datos en ficheros de texto, o si deben utilizar ficheros binarios. María comenta -Quizás debemos usar los dos tipos de ficheros, dependerá de qué se vaya a guardar, -Juan le contesta -tienes razón María, pero debemos pensar entonces cómo va el programa a leer y a escribir la información, tendremos que utilizar las clases Java adecuadas según los ficheros que decidamos usar.*

Existen dos tipos de flujos, flujos de bytes (byte streams) y flujos de caracteres (characterstreams).



- Los **flujos de caracteres** (16 bits) se usan para manipular datos legibles por humanos (por ejemplo un fichero de texto). Vienen determinados por dos clases abstractas: `Reader` y `Writer`. Dichas clases manejan flujos de caracteres [Unicode](). De ellas derivan subclases concretas que implementan los métodos definidos destacados los métodos `read()` y `write()` que, en este caso, leen y escriben **caracteres** de datos respectivamente.
- Los **flujos de bytes** (8 bits) se usan para manipular datos binarios, legibles solo por la maquina (por ejemplo un fichero de programa). Su uso está orientado a la lectura y escritura de datos binarios. El tratamiento del flujo de bytes viene determinado por dos clases abstractas que son `InputStream` y `OutputStream`. Estas dos clases son las que definen los métodos que sus subclases tendrán implementados y, de entre todos, destacan `read()` y `write()` que leen y escriben bytes de datos respectivamente.

Las clases del paquete `java.io` se pueden ver en la ilustración. Destacamos las clases relativas a flujos:

- `BufferedInputStream`: permite leer datos a través de un flujo con un buffer intermedio.
- `BufferedOutputStream`: implementa los métodos para escribir en un flujo a través de un buffer.
- `FileInputStream`: permite leer bytes de un fichero.
- `FileOutputStream`: permite escribir bytes en un fichero o descriptor.
- `StreamTokenizer`: esta clase recibe un flujo de entrada, lo analiza (parse) y divide en diversos pedazos (tokens), permitiendo leer uno en cada momento.
- `StringReader`: es un flujo de caracteres cuya fuente es una cadena de caracteres o string.
- `StringWriter`: es un flujo de caracteres cuya salida es un buffer de cadena de caracteres, que puede utilizarse para construir un string.

Destacar que hay clases que se **"montan" sobre otros flujos para modificar la forma de trabajar con ellos.** Por ejemplo, con `BufferedInputStream` podemos añadir un buffer a un flujo `FileInputStream`, de manera que se mejore la eficiencia de los accesos a los dispositivos en los que se almacena el fichero con el que conecta el flujo.

3.1.- Ejemplo comentado de una clase con flujos.

Vamos a ver un ejemplo con una de las clases comentadas, en concreto, con `StreamTokenizer`.

La clase `StreamTokenizer` obtiene un flujo de entrada y lo divide en "tokens". El flujo tokenizerpuede reconocer identificadores, números y otras cadenas.

El ejemplo que puedes descargarte en el siguiente recurso, muestra cómo utilizar la clase `StreamTokenizer` para contar números y palabras de un fichero de texto. Se abre el flujo con ayuda de la clase `FileReader`, y puedes ver cómo se "monta" el flujo `StreamTokenizer` sobre el `FileReader`, es decir, que se construye el objeto **StreamTokenizer** con el flujo FileReader como argumento, y entonces se empieza a iterar sobre él.

```
packagetokenizer;

importjava.io.FileNotFoundException;
importjava.io.FileReader;
importjava.io.IOException;
importjava.io.StreamTokenizer;


public class token {

public void contarPalabrasyNumeros(String nombre_fichero) {

StreamTokenizersTokenizer = null;
intcontapal = 0, numNumeros = 0;

try {

sTokenizer = new StreamTokenizer(new FileReader(nombre_fichero));

while (sTokenizer.nextToken() != StreamTokenizer.TT_EOF) {

if (sTokenizer.ttype == StreamTokenizer.TT_WORD)
contapal++;
else if (sTokenizer.ttype == StreamTokenizer.TT_NUMBER)
numNumeros++;
            }

System.out.println("Número de palabras en el fichero: " + contapal);
System.out.println("Número de números en el fichero: " + numNumeros);

        } catch (FileNotFoundException ex) {
System.out.println(ex.getMessage()) ;
        } catch (IOException ex) {
System.out.println(ex.getMessage()) ;
        }
    }

    /**
     * @paramargs the command line arguments
     */
public static void main(String[] args) {
new Main().countWordsAndNumbers("c:\\datos.txt");
    }
}
```

El método `nextToken` devuelve un int que indica el tipo de token leído. Hay una serie de constantes definidas para determinar el tipo de token:

- ✓ `TT_WORD` indica que el token es una palabra.
- ✓ `TT_NUMBER` indica que el token es un número.
- ✓ `TT_EOL` indica que se ha leído el fin de línea.
- ✓ `TT_EOF` indica que se ha llegado al fin del flujo de entrada.

En el código de la clase, apreciamos que se iterará hasta llegar al fin del fichero. Para cada token, se mira su tipo, y según el tipo se incrementa el contador de palabras o de números.

Indica si la siguiente afirmación es verdadera o falsa.

Según el sistema operativo que utilicemos, habrá que utilizar un flujo u otro.

Verdadero. Falso.

4.- Flujos.

Caso práctico

Ana *y* ***Antonio*** *salen de clase. Antonio ha quedado con una amiga y Ana va camino de casa pensando en lo que le explicaron en clase hace unos días. Como se quedó con dudas, también le consultó a María. En concreto, le asaltaban dudas sobre cómo leer y escribir datos por teclado en un programa, y también varias dudas sobre lectura y escritura de información en ficheros.* ***María*** *le solventó las dudas hablándole sobre el tema, pero aún así, tenía que probarlo tranquilamente en casa, haciéndose unos pequeños ejemplos, para comprobar toda la nueva información aprendida.*

-Antes de irte, -dice Antonio a Ana, -siéntate a hablar con nosotros un rato.

-Bueno, pero me voy a ir enseguida, -contesta Ana-.

Hemos visto qué es un flujo y que existe un árbol de clases amplio para su manejo. Ahora vamos a ver en primer lugar los **flujos predefinidos**, también conocidos como de entrada y salida, y después veremos los **flujos basados en bytes** y los **flujos basados en carácter**.

"Lo escuché y lo olvidé, lo vi y lo entendí, lo hice y lo aprendí".Confucio.

Para saber más

Antes, hemos mencionado Unicode. Puedes consultar el origen y más cosas sobre Unicode, en el enlace de la Wikipedia:

http://www.juntadeandalucia.es/educacion/adistancia/cursos/mod/scorm/player.php?id=74601¤torg=eXeContenidos4df9bd3a1eeeacaecb3a&mode=normal&scoid=145047

4.1.- Flujos predefinidos. Entrada y salida estándar.

Tradicionalmente, los usuarios del sistema operativo Unix, Linux y también MS-DOS, han utilizado un tipo de entrada/salida conocida comúnmente por entrada/salida estándar. El fichero de entrada estándar (`stdin`) es típicamente el teclado. El fichero de salida estándar (`stdout`) es típicamente la pantalla (o la ventana del terminal). El fichero de salida de error estándar (`stderr`) también se dirige normalmente a la pantalla, pero se implementa como otro fichero de forma que se pueda distinguir entre la salida normal y (si es necesario) los mensajes de error.

Java tiene acceso a la entrada/salida estándar a través de la clase `System`. En concreto, los tres ficheros que se implementan son:

- `Stdin`. Es un objeto de tipo `InputStream`, y está definido en la clase `System` como flujo de entrada estándar. Por defecto es el teclado, pero puede redirigirse para cada host o cada usuario, de forma que se corresponda con cualquier otro dispositivo de entrada.
- `Stdout. System.out` implementa `stdout` como una instancia de la clase `PrintStream`. Se pueden utilizar los métodos `print()` y `println()` con cualquier tipo básico Java como argumento.
- `Stderr`. Es un objeto de tipo `PrintStream`. Es un flujo de salida definido en la clase `System` y representa la salida de error estándar. Por defecto, es el monitor, aunque es posible redireccionarlo a otro dispositivo de salida.

Para la entrada, se usa el método read para leer de la entrada estándar:

- `intSystem.in.read();`
  - → Lee el siguiente `byte (char)` de la entrada estándar.

- ✓ `intSystem.in.read(byte[] b);`
  - → Leer un conjunto de bytes de la entrada estándar y lo almacena en el vector b.

Para la salida, se usa el método print para escribir en la salida estándar:

- ✓ `System.out.print(String);`
  - → Muestra el texto en la consola.
- ✓ `System.out.println(String);`
  - → Muestra el texto en la consola y seguidamente efectúa un salto de línea.

Normalmente, para **leer valores numéricos**, lo que se hace es tomar el valor de la entrada estándar en forma de cadena y entonces usar métodos que permiten transformar el texto a números (int, float, double, etc.) según se requiera.

Funciones de conversión.	
Método	**Funcionamiento**
byte Byte.parseByte(String)	Convierte una cadena en un número entero de un byte
short Short.parseShort(String)	Convierte una cadena en un número entero corto
intInteger.parseInt(String)	Convierte una cadena en un número entero
longLong.parseLong(String)	Convierte una cadena en un número entero largo
floatFloat.parseInt(String)	Convierte una cadena en un número real simple
doubleDouble.parseInt(String)	Convierte una cadena en un número real doble
booleanBoolean.parseBoolean(String)	Convierte una cadena en un valor lógico

4.2.- Flujos predefinidos. Entrada y salida estándar. Ejemplo.

Veamos un ejemplo en el que se lee por teclado hasta pulsar la tecla de retorno, en ese momento el programa acabará imprimiendo por la salida estándar la cadena leída.

Para ir construyendo la cadena con los caracteres leídos podríamos usar la clase `StringBuffer` o la `StringBuilder`. La clase `StringBuffer` permite almacenar cadenas que cambiarán en la ejecución del programa. `StringBuilder` es similar, pero no es síncrona. De este modo, para la mayoría de las aplicaciones, donde se ejecuta un solo hilo, supone una mejora de rendimiento sobre `StringBuffer`.

El proceso de lectura ha de estar en un bloque `try..catch`.

```
Import java.io.IOException;

public class leeEestandar {
   public static void main(String[] args) {
      // Cadena donde iremos almacenando los caracteres que se escriban
      StringBuilder str = new StringBuilder();
      char c;
      // Por si ocurre una excepción ponemos el bloque try-cath
      try{
         // Mientras la entrada de terclado no sea Intro
         while ((c=(char)System.in.read())!='\n'){
            // Añadir el character leído a la cadena str
            str.append(c);
         }
      }catch(IOException ex){
         System.out.println(ex.getMessage()); }

      // Escribir la cadena que se ha ido tecleando
      System.out.println("Cadena introducida: " + str);
   }
}
```

Señala la opción correcta:

- [ ] Read es una clase de System que permite leer caracteres.
- [ ] `StringBuffer` permite leer y `StringBuilder` escribir en la salida estándar.
- [ ] La clase `keyboard` también permite leer flujos de teclado.

Stderr por defecto dirige al monitor pero se puede direccionar a otro dispositivo.

Debes conocer

Debemos procesar las excepciones que la propia máquina virtual va a lanzar. Existen dos mecanismos básicos para gestionar los errores que se pueden producir en una aplicación: el primero consiste en utilizar la estructura `try-catch`, que permite intentar capturar un error que se pueda producir. Se pone el `try`, el código en cuestión y se finaliza con el `catch`. En esa sección del `catch` es donde pondremos las operaciones a realizar cuando se detecte un error. Se pueden poner varios bloques `catch`: una genérica y otras más detalladas.

El otro mecanismo consiste en declarar que nuestro método puede lanzar una excepción, pero no gestionarla en ese método, sino tratarla, por ejemplo, en el método `main()` en el que se está llamando a ese método.

4.3.- Flujos basados en bytes.

Este tipo de flujos es el idóneo para el manejo de entradas y salidas de bytes, y su uso por tanto está orientado a la lectura y escritura de datos binarios.

Para el tratamiento de los flujos de bytes, Java tiene dos clases abstractas que son `InputStream`y `OutputStream`. Cada una de estas clases abstractas tiene varias subclases concretas, que controlan las diferencias entre los distintos dispositivos de E/S que se pueden utilizar.

```
classFileInputStreamextandsInputStream {
FileInputStream (String fichero) throws FileNotFoundException;
FileInputStream (File fichero) throws FileNotFoundException;
    ... ... ...
  }

classFileOutputStream extends OutputStream {
FileOutputStream (String fichero) throws FileNotFoundException;
FileOutputStream (File fichero) throws FileNotFoundException;
... ... ...
          }
```

`OutputStream` y el `InputStream` y todas sus subclases, reciben en el constructor el objeto que representa el flujo de datos para el dispositivo de entrada o salida.

Por ejemplo, podemos copiar el contenido de un fichero en otro:

```
voidcopia (String origen, String destino) throws IOException {
try{
            // Obtener los nombres de los ficheros de origen y destino
            // y abrir la conexión a los ficheros.
InputStreamfentrada = new FileInputStream(origen);
OutputStreamfsalida = new FileOutputStream(destino);
            // Crear una variable para leer el flujo de bytes del origen
byte[] buffer= new byte[256];
while (true) {
                 // Leer el flujo de bytes
int n = fentrada.read(buffer);
                 // Si no queda nada por leer, salir delwhile
if (n < 0)
break;
                 // Escribir el flujo de bytes leídos al fichero destino
fsalida.write(buffer, 0, n);
            }
            // Cerrar los ficheros
fentrada.close();
fsalida.close();
}catch(IOException ex){
System.out.println(ex.getMessage()); }
```

Recomendación

En los enlaces siguientes puedes ver la documentación oficial de Oracle sobre `FileInputStream` y sobre `FileOutputStream`.

FileInputStream.	http://download.oracle.com/javase/1.4.2/docs/api/java/io/FileInputStream.html
FileOutputStream.	http://download.oracle.com/javase/1.4.2/docs/api/java/io/FileOutputStream.html

4.4.- Flujos basados en caracteres.

Las clases orientadas al flujo de bytes nos proporcionan la suficiente funcionalidad para realizar cualquier tipo de operación de entrada o salida, pero no pueden trabajar directamente con **caracteres Unicode**, los cuales están **representados por dos bytes**. Por eso, se consideró necesaria la creación de las clases orientadas al flujo de caracteres para ofrecernos el soporte necesario para el tratamiento de caracteres.
Para los flujos de caracteres, Java dispone de dos clases abstractas: `Reader`y `Writer`.
`Reader`, `Writer`, y todas sus subclases, reciben en el constructor el objeto que representa el flujo de datos para el dispositivo de entrada o salida.
Hay que recordar que **cada vez que se llama a un constructor se abre el flujo de datos y es necesario cerrarlo** cuando no lo necesitemos.
Existen muchos tipos de flujos dependiendo de la utilidad que le vayamos a dar a los datos que extraemos de los dispositivos.
Un flujo puede ser envuelto por otro flujo para tratar el flujo de datos de forma cómoda. Así, un `bufferWriter` nos permite manipular el flujo de datos como un buffer, pero si lo envolvemos en un `PrintWriter` lo podemos escribir con muchas más funcionalidades adicionales para diferentes tipos de datos.
En este ejemplo de código, se ve cómo podemos escribir la salida estándar a un fichero. Cuando se teclee la palabra "salir", se dejará de leer y entonces se saldrá del bucle de lectura.
Podemos ver cómo se usa `InputStreamReader` que es un puente de flujos de bytes a flujos de caracteres: lee bytes y los decodifica a caracteres. `BufferedReader` lee texto de un flujo de entrada de caracteres, permitiendo efectuar una lectura eficiente de caracteres, vectores y líneas.
Como vemos en el código, usamos `FileWriter` para flujos de caracteres, pues para datos binarios se utiliza `FileOutputStream`.

```
try{
PrintWriter out = null;
out = new PrintWriter(new FileWriter("c:\\salida.txt", true));

BufferedReaderbr = new BufferedReader(
newInputStreamReader(System.in));
      String s;
while (!(s = br.readLine()).equals("salir")){
out.println(s);
      }
out.close();
     }
catch(IOException ex){
System.out.println(ex.getMessage()); }
```

Indica si es verdadera o falsa la siguiente afirmación:
Para flujos de caracteres es mejor usar las clases `Reader` y `Writer` en vez de `InputStream` y `OutputStream`.
Verdadero. Falso.

4.5.- Rutas de los ficheros.

En los ejemplos que vemos en el tema estamos usando la ruta de los ficheros tal y como se usan en MS-DOS, o Windows, es decir, por ejemplo:

```
c:\\datos\Programacion\fichero.txt
```

Cuando operamos con rutas de ficheros, el carácter separador entre directorios o carpetas suele cambiar dependiendo del sistema operativo en el que se esté ejecutando el programa.

Para evitar problemas en la ejecución de los programas cuando se ejecuten en uno u otro sistema operativo, y por tanto persiguiendo que nuestras aplicaciones sean lo más portables posibles, se recomienda usar en Java: `File.separator`.

Podríamos hacer una función que al pasarle una ruta nos devolviera la adecuada según el separador del sistema actual, del siguiente modo:

```
String substFileSeparator(String ruta){
 String separador = "\\";
try{
     // Si estamos en Windows
if ( File.separator.equals(separador) )
separador = "/" ;
        // Reemplaza todas las cadenas que coinciden con la expresión
        // regular dada oldSep por la cadena File.separator
returnruta.replaceAll(separador, File.separator);
}catch(Exception e){
    // Por si ocurre una java.util.regex.PatternSyntaxException
returnruta.replaceAll(separador + separador, File.separator);
}
       }
```

Indica si es verdadera o falsa la siguiente afirmación.

Cuando trabajamos con fichero en Java, no es necesario capturar las excepciones, el sistema se ocupa automáticamente de ellas. ¿Verdadero o Falso?

Verdadero. ○ Falso. ◉

5.- Trabajando con ficheros.

Caso práctico

***Juan** le comenta a **María** -Tenemos que programar una copia de seguridad diaria de los datos del ficheros de texto plano que utiliza el programa para guardar la información. -Mientras María escucha a Juan, recuerda que para copias de seguridad, siempre ha comprobado que la mejor opción es utilizar ficheros secuenciales.*

¿Crees que es una buena opción la que piensa María o utilizarías otra en su lugar?

En este apartado vas a ver muchas cosas sobre los ficheros: cómo leer y escribir en ellos, aunque ya hemos visto algo sobre eso, hablaremos de las formas de acceso a los ficheros: secuencial o de manera aleatoria.

Siempre hemos de tener en cuenta que la manera de proceder con ficheros debe ser:

Abrir o bien **crear** si no existe el fichero.

Hacer las operaciones que necesitemos.

Cerrar el fichero, para no perder la información que se haya modificado o añadido.

También es muy importante el **control de las excepciones**, para evitar que se produzcan fallos en tiempo de ejecución. Si intentamos abrir sin más un fichero, sin comprobar si existe o no, y no existe, saltará una excepción.

Para saber más

En el siguiente enlace a la wikipedia podrás ver la descripción de varias extensiones que pueden presentar los archivos.

Extensión de un archivo. http://es.wikipedia.org/wiki/Extensi%C3%B3n_de_archivo

5.1.- Escritura y lectura de información en ficheros.

Acabamos de mencionar los pasos fundamentales para proceder con ficheros: abrir, operar, cerrar.

Además de esas consideraciones, debemos tener en cuenta también las clases Java a emplear, es decir, recuerda que hemos comentado que si vamos a tratar con ficheros de texto, es más eficiente emplear las clases de `ReaderWriter`, frente a las clases de `InputStream`y `OutputStream` que están indicadas para flujos de bytes.

Otra cosa a considerar, cuando se va a hacer uso de ficheros, es la forma de acceso al fichero que se va a utilizar, si va a ser de manera secuencial o bien aleatoria. En un fichero secuencial, **para acceder a un dato debemos recorrer todo el fichero desde el principio hasta llegar a su posición**. Sin embargo, en un fichero de acceso aleatorio*(Es la posibilidad de acceder a un elemento arbitrario de una secuencia de datos en el mismo tiempo*) podemos posicionarnos directamente en una posición del fichero, y ahí leer o escribir.

Aunque ya has visto un ejemplo que usa `BufferedReader`, insistimos aquí sobre la filosofía de estas clases, que usan la idea de un buffer.

La idea es que cuando una aplicación necesita leer datos de un fichero, tiene que estar esperando a que el disco en el que está el fichero le proporcione la información.

Un dispositivo cualquiera de memoria masiva (*Consiste en un tipo de almacenamiento permanente, no volátil, a diferencia de la memoria principal del ordenador que sí es volátil, pero posee mayor capacidad de almacenamiento que la memoria principal, aunque es más lenta que ésta*), por muy rápido que sea, es mucho más lento que la CPU del ordenador.

Así que, es fundamental **reducir el número de accesos al fichero** a fin de mejorar la eficiencia de la aplicación, y para ello se asocia al fichero una memoria intermedia, el buffer, de modo que cuando se necesita leer un byte del archivo, en realidad se traen hasta el buffer asociado al flujo, ya que es una memoria mucho más rápida que cualquier otro dispositivo de memoria masiva.

Cualquier operación de Entrada/Salida a ficheros puede generar una `IOException`, es decir, un error de Entrada/Salida. Puede ser por ejemplo, que el fichero no exista, o que el dispositivo no funcione correctamente, o que nuestra aplicación no tenga permisos de lectura o escritura sobre el

fichero en cuestión. Por eso, las sentencias que involucran operaciones sobre ficheros, deben ir en un bloque `try`.

Señala si es verdadera o es falsa la siguiente afirmación:
La idea de usar buffers con los ficheros es incrementar los accesos físicos a disco.
Verdadero. Falso.

5.2.- Ficheros binarios y ficheros de texto (I).

Ya comentamos anteriormente que los ficheros se utilizan para guardar la información en un soporte: disco duro, disquetes, memorias usb, dvd, etc., y posteriormente poder recuperarla. También distinguimos dos tipos de ficheros: los de **texto** y los **binarios**.
En los **ficheros de texto** la información se guarda como caracteres. Esos caracteres están codificados en **Unicode**, o en **ASCII** u otras codificaciones de texto.
En la siguiente porción de código puedes ver cómo para un fichero existente, que en este caso es texto.txt, averiguamos la codificación que posee, usando el método `getEncoding()`

```
FileInputStream fichero;
try {
     // Elegimos fichero para leer flujos de bytes "crudos"
fichero = new FileInputStream("c:\\texto.txt");
     // InputStreamReader sirve de puente de flujos de byte a caracteres
InputStreamReaderunReader = new InputStreamReader(fichero);
     // Vemos la codificaciÓn actual
System.out.println(unReader.getEncoding());
} catch (FileNotFoundException ex) {
Logger.getLogger(textos.class.getName()).log(Level.SEVERE, null, ex);
}
```

Para **archivos de texto**, se puede abrir el fichero para leer usando la clase `FileReader`. Esta clase nos proporciona métodos para **leer caracteres**. Cuando nos interese no leer carácter a carácter, sino **leer líneas completas**, podemos usar la clase `BufferedReader` a partir de `FileReader`. Lo podemos hacer de la siguiente forma:

```
File arch = new File ("C:\\fich.txt");
FileReaderfr = new FileReader (arch);
BufferedReaderbr = new BufferedReader(fr);
...
String linea = br.readLine();
```

Para **escribir en archivos de texto** lo podríamos hacer, teniendo en cuenta:

```
FileWriterfich = null;
PrintWriter pw = null;
fich = new FileWriter("c:/fich2.txt");
pw = new PrintWriter(fichero);
pw.println("Linea de texto");
...
```

Si el fichero al que queremos escribir existe y lo que queremos es añadir información, entonces pasaremos el segundo parámetro como true:

```
FileWriter("c:/fich2.txt",true);
```

Para saber más

En el siguiente enlace a wikipedia puedes ver el código ASCII.

Binario	Dec	Hex	Representa
0010 0000	32	20	espacio ( )
0010 0001	33	21	!
0010 0010	34	22	"
0010 0011	35	23	#
0010 0100	36	24	$
0010 0101	37	25	%
0010 0110	38	26	&
0010 0111	39	27	'
0010 1000	40	28	(
0010 1001	41	29	)
0010 1010	42	2A	*
0010 1011	43	2B	+
0010 1100	44	2C	,
0010 1101	45	2D	-
0010 1110	46	2E	.
0010 1111	47	2F	/
0011 0000	48	30	0
0011 0001	49	31	1
0011 0010	50	32	2
0011 0011	51	33	3
0011 0100	52	34	4
0011 0101	53	35	5
0011 0110	54	36	6
0011 0111	55	37	7
0011 1000	56	38	8
0011 1001	57	39	9
0011 1010	58	3A	:
0011 1011	59	3B	;
0011 1100	60	3C	<
0011 1101	61	3D	=
0011 1110	62	3E	>
0011 1111	63	3F	?

Binario	Dec	Hex	Representa
0100 0000	64	40	@
0100 0001	65	41	A
0100 0010	66	42	B
0100 0011	67	43	C
0100 0100	68	44	D
0100 0101	69	45	E
0100 0110	70	46	F
0100 0111	71	47	G
0100 1000	72	48	H
0100 1001	73	49	I
0100 1010	74	4A	J
0100 1011	75	4B	K
0100 1100	76	4C	L
0100 1101	77	4D	M
0100 1110	78	4E	N
0100 1111	79	4F	O
0101 0000	80	50	P
0101 0001	81	51	Q
0101 0010	82	52	R
0101 0011	83	53	S
0101 0100	84	54	T
0101 0101	85	55	U
0101 0110	86	56	V
0101 0111	87	57	W
0101 1000	88	58	X
0101 1001	89	59	Y
0101 1010	90	5A	Z
0101 1011	91	5B	[
0101 1100	92	5C	\
0101 1101	93	5D	]
0101 1110	94	5E	^
0101 1111	95	5F	_

Binario	Dec	Hex	Representa
0110 0000	96	60	`
0110 0001	97	61	a
0110 0010	98	62	b
0110 0011	99	63	c
0110 0100	100	64	d
0110 0101	101	65	e
0110 0110	102	66	f
0110 0111	103	67	g
0110 1000	104	68	h
0110 1001	105	69	i
0110 1010	106	6A	j
0110 1011	107	6B	k
0110 1100	108	6C	l
0110 1101	109	6D	m
0110 1110	110	6E	n
0110 1111	111	6F	o
0111 0000	112	70	p
0111 0001	113	71	q
0111 0010	114	72	r
0111 0011	115	73	s
0111 0100	116	74	t
0111 0101	117	75	u
0111 0110	118	76	v
0111 0111	119	77	w
0111 1000	120	78	x
0111 1001	121	79	y
0111 1010	122	7A	z
0111 1011	123	7B	{
0111 1100	124	7C	|
0111 1101	125	7D	}
0111 1110	126	7E	~

5.2.1.- Ficheros binarios y ficheros de texto (II).

Los **ficheros binarios** almacenan la información en bytes, codificada en binario, pudiendo ser de cualquier tipo: fotografías, números, letras, archivos ejecutables, etc.

Los archivos binarios guardan una representación de los datos en el fichero. O sea que, cuando se guarda texto no se guarda el texto en sí, sino que se guarda su representación en código UTF-8 (*Es un formato de codificación de caracteres Unicode usando símbolos de longitud variable*).

Para **leer datos de un fichero binario**, Java proporciona la clase **FileInputStream**. Dicha clase trabaja con bytes que se leen desde el flujo asociado a un fichero. Aquí puedes ver un ejemplo comentado.

```
packagefileinputconbuffer;

importjava.io.BufferedInputStream;
importjava.io.File;
importjava.io.FileInputStream;
importjava.io.IOException;

public class leerConBuffer {

public static void main(String[] args) {
int tama ;

try{
      // Creamos un nuevo objeto File, que es la ruta hasta el fichero desde
      File f = new File("C:\\apuntes\\test.bin");

      // Construimosunflujo de tipo FileInputStream (un flujo de entrada desde
```

```
      // fichero) sobre el objeto File. Estamos conectando nuestra aplicación
      // a un extremo del flujo, por donde van a salir los datos, y "pidiendo"
      // al Sistema Operativo que conecte el otro extremo al fichero que indica
      // la ruta establecida por el objeto File f que habíamos creado antes. De
FileInputStreamflujoEntrada = new FileInputStream(f);
BufferedInputStreamfEntradaConBuffer = new BufferedInputStream(flujoEntrada);

      // Escribimos el tamañodel fichero en bytes.
tama = fEntradaConBuffer.available();
System.out.println("Bytes disponibles: " + tama);

      // Indicamos que vamos a intentar leer 50 bytes del fichero.
System.out.println("Leyendo 50 bytes....");

      // Creamos unarray de 50 bytes para llenarlo con los 50 bytes
      // que leamos del flujo (realmente del fichero)*/
bytebytearray[] = new byte[50];

      // El método read() de la clase FileInputStream recibe como parámetro un
      // array de byte, y lo llena leyendo bytes desde el flujo.
      // Devuelve un número entero, que es el número de bytes que realmente se
      // han leído desde el flujo. Si el fichero tiene menos de 50 bytes, no
      // podrá leer los 50 bytes, y escribirá un mensaje indicándolo.
if (fEntradaConBuffer.read(bytearray) != 50)
System.out.println("No se pudieron leer 50 bytes");

      // Usamos un constructor adecuado de la clase String, que crea un nuevo
      // String a partir de los bytes leídos desde el flujo, que se almacenaron
      // en el arraybytearray, y escribimos eseString.
System.out.println(new String(bytearray, 0, 50));

      // Finalmentecerramos el flujo.Es importante cerrar los flujos
      // para liberar ese recurso. Al cerrar el flujo, se comprueba que no
      // haya quedado ningún dato en el flujo sin que se haya leído por la aplicación. */
fEntradaConBuffer.close();

      // Capturamos la excepción de Entrada/Salida. El error que puede
      // producirse en este caso es que el fichero no esté accesible, y
      // es el mensaje que enviamos en tal caso.
}catch (IOException e){
System.err.println("No se encuentra el fichero");
    }
    }

}
```

Para **escribir datos a un fichero binario**, la clase nos permite usar un fichero para escritura de bytes en él, es la clase`FileOutputStream`. La filosofía es la misma que para la lectura de datos, pero ahora el flujo es en dirección contraria, desde la aplicación que hace de fuente de datos hasta el fichero, que los consume.

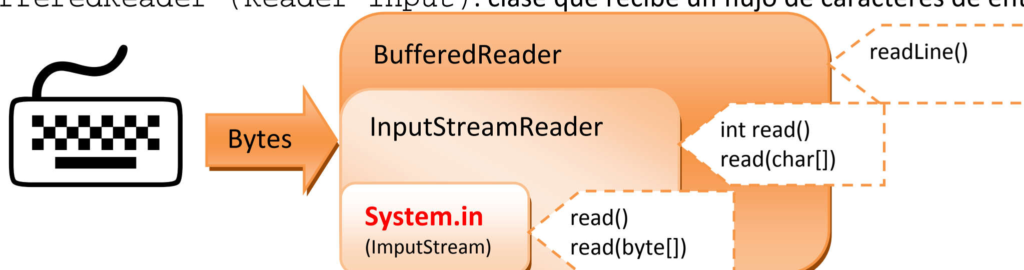
En la siguiente presentación puedes ver un esquema de cómo utilizar buffer para optimizar la lectura de teclado desde consola, por medio de las envolturas, podemos usar métodos como `readline()`, de la clase `BufferedReader`, que envuelve a un objeto de la clase `InputStreamReader`.

Envolturas o Wrappers

Leer desde consola

```
BufferedReader buf = new BufferedReader(new InputStreamReader(System.in);
```

- `Buf` es un flujo de caracteres que se enlaza a la consola a través de la clase `System.in`. Esta se envuelve para pasar de byte a char
- `InputStreamReader (InputStream inp)`: clase que convierte de byte a carácter.
- `BufferedReader (Reader input)`: clase que recibe un flujo de caracteres de entrada.



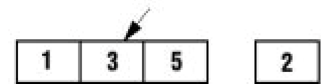
Resumen textual alternativo

Señala si es verdadera o falsa la siguiente afirmación:

Para leer datos desde un fichero codificados en binario empleamos la clase `FileOutputStream`.

Verdadero. Falso.

5.3.- Modos de acceso. Registros.



En Java no se impone una estructura en un fichero, por lo que conceptos como el de registro que si existen en otros lenguajes, en principio no existen en los archivos que se crean con Java. Por tanto, los programadores deben estructurar los ficheros de modo que cumplan con los requerimientos de sus aplicaciones.

Así, el programador definirá su registro con el número de bytes que le interesen, moviéndose luego por el fichero teniendo en cuenta ese tamaño que ha definido.

Se dice que un fichero es de acceso directo o de organización directa cuando para acceder a un registro **n** cualquiera, no se tiene que pasar por los **n-1** registros anteriores. En caso contrario, estamos hablando de ficheros secuenciales.

Con Java se puede trabajar con **ficheros secuenciales** y con **ficheros de acceso aleatorio**.

En los **ficheros secuenciales**, la información se almacena de manera secuencial, de manera que para recuperarla se debe hacer en el mismo orden en que la información se ha introducido en el archivo. Si por ejemplo queremos leer el registro del fichero que ocupa la posición tres (en la ilustración sería el número 5), tendremos que abrir el fichero y leer los primeros tres registros, hasta que finalmente leamos el registro número tres.

Por el contrario, si se tratara de un **fichero de acceso aleatorio**, podríamos acceder directamente a la posición tres del fichero, o a la que nos interesara.

Señala la opción correcta:

- Java sólo admite el uso de ficheros aleatorios.
- Con los ficheros de acceso aleatorio se puede acceder a un registro determinado directamente.
- Los ficheros secuenciales se deben leer de tres en tres registros.
- Todas son falsas.

5.4.- Acceso secuencial.

En el siguiente ejemplo vemos cómo se **escriben datos en un fichero secuencial**: el nombre y apellidos de una persona utilizando el método `writeUTF()` que proporciona `DataOutputStream`, seguido de su edad que la escribimos con el método `writeInt()` de la misma clase. A continuación escribimos lo mismo para una segunda persona y de nuevo para una tercera. Después cerramos el fichero. Y ahora lo abrimos de nuevo para ir **leyendo de manera secuencial** los datos almacenados en el fichero, y escribiéndolos a consola.

```
/*
 * To change this template, choose Tools | Templates
 * and open the template in the editor.
 */

package escylecsecuen;

import java.io.DataInputStream;
import java.io.DataOutputStream;
```

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

/**
 *
 * @author JJBH
 */
public class escylee {

    /**
     * @param args the command line arguments
     */
    public static void main(String[] args) {
        // Declarar un objeto de tipo archivo
        DataOutputStream archivo = null ;
        DataInputStream fich = null ;
        String nombre = null ;
        int edad = 0 ;
        try {
            // Creando o abriendo para añadir el archivo
            archivo = new DataOutputStream( new FileOutputStream("c:\\secuencial.dat",true) );

            // Escribir el nombre y los apellidos
            archivo.writeUTF( "Antonio López Pérez     " );
            archivo.writeInt(33) ;
            archivo.writeUTF( "Pedro Piqueras Peñaranda" );
            archivo.writeInt(45) ;
            archivo.writeUTF( "José Antonio Ruiz Pérez " ) ;
            archivo.writeInt(51) ;
            // Cerrar fichero
            archivo.close();

            // Abrir para leer
            fich = new DataInputStream( new FileInputStream("c:\\secuencial.dat") );
            nombre = fich.readUTF() ;
            System.out.println(nombre) ;
            edad = fich.readInt() ;
            System.out.println(edad) ;
            nombre = fich.readUTF() ;
            System.out.println(nombre) ;
            edad = fich.readInt() ;
            System.out.println(edad) ;
            nombre = fich.readUTF() ;
            System.out.println(nombre) ;
            edad = fich.readInt() ;
            System.out.println(edad) ;
            fich.close();


      } catch(FileNotFoundException fnfe) { /* Archivo no encontrado */ }
        catch (IOException ioe) { /* Error al escribir */ }
        catch (Exception e) { /* Error de otro tipo*/
          System.out.println(e.getMessage());}

    }

}
```

Fíjate al ver el código, que hemos tenido la precaución de ir escribiendo las cadenas de caracteres con el mismo tamaño, de manera que sepamos luego el tamaño del registro que tenemos que leer.
Por tanto para **buscar información en un fichero secuencial**, tendremos que abrir el fichero e ir leyendo registros hasta encontrar el registro que buscamos.

¿Y si queremos **eliminar un registro en un fichero secuencial**, qué hacemos? Esta operación es un problema, puesto que no podemos quitar el registro y reordenar el resto. Una opción, aunque costosa, sería crear un nuevo fichero. Recorremos el fichero original y vamos copiando registros en el nuevo hasta llegar al registro que queremos borrar. Ese no lo copiamos al nuevo, y seguimos copiando hasta el final, el resto de registros al nuevo fichero. De este modo, obtendríamos un nuevo fichero que sería el mismo que teníamos pero sin el registro que queríamos borrar. Por tanto, si se

prevé que se va a borrar en el fichero, no es recomendable usar un fichero de este tipo, o sea, secuencial.

Señala si es verdadera o es falsa la siguiente afirmación:
Para encontrar una información almacenada en la mitad de un fichero secuencial, podemos acceder directamente a esa posición sin pasar por los datos anteriores a esa información.
Verdadero. Falso.

5.5.- Acceso aleatorio.

A veces no necesitamos leer un fichero de principio a fin, sino acceder al fichero como si fuera una base de datos, donde se accede a un registro concreto del fichero. Java proporciona la clase `RandomAccessFile` para este tipo de entrada/salida.
La clase `RandomAccessFile` permite utilizar un fichero de **acceso aleatorio** en el que el programador define el formato de los registros.

```
RandomAccessFile objFile = new RandomAccessFile( ruta, modo );
```

Donde ruta es la dirección física en el sistema de archivos y modo puede ser:

- ✓ "r" para sólo lectura.
- ✓ "rw" para lectura y escritura.

La clase `RandomAccessFile` implementa los interfaces `DataInput` y `DataOutput`. Para abrir un archivo en modo lectura haríamos:

```
RandomAccessFile in = new RandomAccessFile("input.dat", "r");
```

Para abrirlo en modo lectura y escritura:

```
RandomAccessFile inOut = new RandomAccessFile("input.dat", "rw");
```

Esta clase permite leer y escribir sobre el fichero, no se necesitan dos clases diferentes.

Hay que especificar el modo de acceso al construir un objeto de esta clase: sólo lectura o lectura/escritura.

Dispone de métodos específicos de desplazamiento como `seek` y `skipBytes` para poder moverse de un registro a otro del fichero, o posicionarse directamente en una posición concreta del fichero.
No está basada en el concepto de flujos o streams.

En la siguiente presentación vemos cómo crear un fichero, cómo podemos acceder, y actualizar información en él.

Ficheros de acceso aleatorio

Crear fichero:
Debido a la existencia de dos constructores, tenemos dos posibilidades de abrir un fichero de acceso aleatorio:

- ✓ Mediante el nombre del fichero:

```
miFichero = new RandomAccessFile(String nombre, String modo);
```

- ✓ Mediante un objeto file:

```
miFichero = new RandomAccessFile(File fichero, String modo);
```

El parámetro **modo** determina si se tiene acceso de sólo lectura (r) o bien de lectura/escritura (rw). Por ejemplo, se puede abrir un fichero de sólo lectura:

```
RandomAccessFile miRAFile;
miRAFile = new RandomAccessFile("/usr/bin/pepe.txt","rw");
```

Acceder a la información:

- ✓ Con un objeto RandomAccessFile se tiene acceso a todas las operaciones read() y write() de las clases DataInputStream y DataOutputStream
- ✓ Se dispone de muchos métodos para ubicarsedentro de un fichero:

```
long getFilePointer()
```

Devuelve la posición actual del punter del fichero

```
void seek(long pos)
```

Sitúa el puntero del fichero en una posición determinada. La posición se da como un desplazamiento en bytes desde el comienzo del fichero. La posición 0 marca el comienzo de ese fichero

```
long length()
```

Devuelve la longitud del fichero. La posición length() marca el final de ese fichero.

```
int skipBytes(int desplazamiento)
```

Desplaza el puntero desde la posición actual, el número de bytes indicado por desplazamiento

Actualizar la información:

Con los ficheros de acceso aleatorio se puede añadir información a un fichero aleatorio existente. Aquí nos iríamos al final del fichero:

```
miFR = new RandomAccessFile("c:\\aleatorios\prueba.dat","rw");
miFR.seek(miFR.length());
```

Cualquier write() que hagamos a partir de este punto del código añadirá información al fichero.

Indica si es verdadera o es falsa la siguiente afirmación:

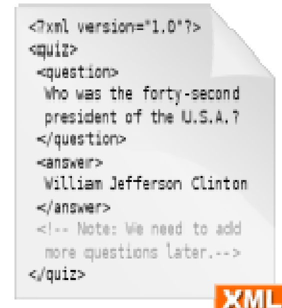
Para decirle el modo de lectura y escritura a un objeto `RandomAccessFile` debemos pasar como parámetro "rw".

Verdadero. ◉ Falso. ○

6.- Aplicaciones del almacenamiento de información en ficheros.

Caso práctico

Antonio *ha quedado con* ***Ana*** *para estudiar sobre el tema de ficheros. De camino a la biblioteca, Ana le pregunta a Antonio -¿Crees que los ficheros se utilizan realmente, o ya están desfasados y sólo se utilizan las bases de datos? -Antonio tras pensarlo un momento le dice a Ana -Yo creo que sí, piensa en el mp3 que usas muchas veces, la música va grabada en ese tipo de ficheros.*



¿Has pensado la **diversidad de ficheros** que existe, según la información que se guarda?

Las fotos que haces con tu cámara digital, o con el móvil, se guardan en ficheros. Así, existe una gran cantidad de ficheros de imagen, según el método que usen para guardar la información. Por ejemplo, tenemos los ficheros de extensión: .jpg, .tiff, gif, .bmp, etc.

La música que oyes en tu mp3 o en el reproductor de mp3 de tu coche, está almacenada en ficheros que almacenan la información en formato mp3.

Los sistemas operativos, como Linux, Windows, etc., están constituidos por un montón de instrucciones e información que se guarda en ficheros.

El propio código fuente de los lenguajes de programación, como Java, C, etc., se guarda en ficheros de texto plano la mayoría de veces.

También se guarda en ficheros las películas en formato .avi, .mp4, etc.

Y por supuesto, se usan mucho actualmente los ficheros XML, que al fin y al cabo son ficheros de texto plano, pero que siguen una estructura determinada. XML se emplea mucho para el intercambio de información estructurada entre diferentes plataformas.

Para saber más

En el siguiente enlace a la wikipedia puedes saber más sobre el formato Mp3.

http://es.wikipedia.org/wiki/MP3

Indica si es verdadera o falsa la siguiente afirmación:

Un fichero .bmp guarda información de música codificada. ¿Verdadero o falso?

Verdadero. ○ Falso. ◉

7.- Utilización de los sistemas de ficheros.

Caso práctico

***Ana** está estudiando en la biblioteca, junto a **Antonio**. Está repasando lo que le explicaron en clase sobre las operaciones relativas a ficheros en Java. En concreto, está mirando lo relativo a crear carpetas o directorios, listar directorios, borrarlos, operar en definitiva con ellos. Va a repasar ahora en la biblioteca, para tener claros los conceptos y cuando llegue de vuelta a casa, probar a compilar algunos ejemplos que a ella misma se le ocurran.*

Has visto en los apartados anteriores cómo operar en ficheros: abrirlos, cerrarlos, escribir en ellos, etc.

Lo que no hemos visto es lo relativo a crear y borrar directorios, poder filtrar archivos, es decir, buscar sólo aquellos que tengan determinada característica, por ejemplo, que su extensión sea: .txt.

Ahora veremos cómo hacer estas cosas, y también como borrar ficheros, y crearlos, aunque crearlos ya lo hemos visto en algunos ejemplos anteriores.

Para saber más

Accediendo a este enlace, tendrás una visión detallada sobre la organización de ficheros.

Organización de Ficheros y Métodos de Enlace http://www.um.es/docencia/barzana/IAGP/Iagp7.html

7.1.- Clase File.

La clase `File` proporciona una representación abstracta de ficheros y directorios.

Esta clase, permite examinar y manipular archivos y directorios, independientemente de la plataforma en la que se esté trabajando: Linux, Windows, etc.

Las instancias de la clase `File` representan nombres de archivo, no los archivos en sí mismos.

El archivo correspondiente a un nombre dado podría ser que no existiera, por ello, habrá que controlar las posibles excepciones.

Al trabajar con File, las rutas pueden ser:

- Relativas al directorio actual.
- Absolutas si la ruta que le pasamos como parámetro empieza por
  - La barra "/" en Unix, Linux.
  - Letra de unidad (C:, D:, etc.) en Windows.
  - UNC(universal naming convention) en windows, como por ejemplo:

```
File miFile=new File("\\\\mimaquina\\download\\prueba.txt");
```

A través del objeto `File`, un programa puede examinar los atributos del archivo, cambiar su nombre, borrarlo o cambiar sus permisos. Dado un objeto `file`, podemos hacer las siguientes operaciones con él:

- **Renombrar** el archivo, con el método `renameTo()` (*Consiste en una serie de sentencias para llevar a cabo una acción. Un método está asociado a una clase. Puede tener parámetros de entrada y puede devolver un valor de retorno*). El objeto File dejará de referirse al archivo renombrado, ya que el `String` con el nombre del archivo en el objeto `File` no cambia.

- **Borrar** el archivo, con el método `delete()`. También, con `deleteOnExit()` se borra cuando finaliza la ejecución de la máquina virtual Java.
- **Crear** un nuevo fichero con un nombre único. El método estático `createTempFile()` crea un fichero temporal y devuelve un objeto `File` que apunta a él. Es útil para crear archivos temporales, que luego se borran, asegurándonos tener un nombre de archivo no repetido.
- **Establecer** la fecha y la hora de modificación del archivo con `setLastModified()`. Por ejemplo, se podría hacer: `new File("prueba.txt").setLastModified(new Date().getTime());` para establecerle la fecha actual al fichero que se le pasa como parámetro, en este caso *prueba.txt*.
- **Crear** un directorio con el método `mkdir()`. También existe `mkdirs()`, que crea los directorios superiores si no existen.
- **Listar** el contenido de un directorio. Los métodos `list()` y `listFiles()` listan el contenido de un directorio `list()` devuelve un vector de String con los nombres de los archivos, `listFiles()` devuelve un vector de objetos `File`.
- **Listar** los nombres de archivo de la raíz del sistema de archivos, mediante el método estático `listRoots()`.

Un objeto de la clase File representa un fichero en sí mismo. ¿Verdadero o falso?

Verdadero. Falso.

7.2.- Interface FilenameFilter.

En ocasiones nos interesa ver la lista de los archivos que encajan con un determinado criterio.

Así, nos puede interesar un filtro para ver los ficheros modificados después de una fecha, o los que tienen un tamaño mayor del que indiquemos, etc.

El interface `FilenameFilter` se puede usar para crear filtros que establezcan criterios de filtrado relativos al nombre de los ficheros. Una clase que lo implemente debe definir e implementar el método:

```
boolean accept(File dir, String nombre)
```

Este método devolverá verdadero (`true`), en el caso de que el fichero cuyo nombre se indica en el parámetro `nombre` aparezca en la lista de los ficheros del directorio indicado por el parámetro `dir`.

En el siguiente ejemplo vemos cómo se listan los ficheros de la carpeta c:\datos que tengan la extensión .odt. Usamos `try` y `catch` para capturar las posibles excepciones, como que no exista dicha carpeta.

```
/*
 * To change this template, choose Tools | Templates
 * and open the template in the editor.
 */

package filtrarficheros;
import java.io.File;
import java.io.FilenameFilter;

public class Filtro implements FilenameFilter {
    String extension;
    Filtro(String extension){
        this.extension=extension;
    }
    public boolean accept(File dir, String name){
        return name.endsWith(extension);
    }
```

```
    public static void main(String[] args) {
        try {
            File fichero=new File("c:\\datos\\.");
            String[] listadeArchivos = fichero.list();
            listadeArchivos = fichero.list(new Filtrar(".odt"));
            int numarchivos = listadeArchivos.length ;

            if (numarchivos < 1)
                System.out.println("No hay archivos que listar");
            else
            {
                for(int conta = 0; conta < listadeArchivos.length; conta++)
                    System.out.println(listadeArchivos[conta]);
            }
        }
        catch (Exception ex) {
            System.out.println("Error al buscar en la ruta indicada");
        }
    }
}
```

Una clase que implemente `FileNameFilter` puede o no implementar el método `accept`.

Verdadero. ◯ Falso. ◉

7.3.- Creación y eliminación de ficheros y directorios.

Podemos **crear un fichero** del siguiente modo:

- ✓ Creamos el objeto que encapsula el fichero, por ejemplo, suponiendo que vamos a crear un fichero llamado miFichero.txt, en la carpeta C:\\prueba, haríamos:

```
File fichero = new File("c:\\prueba\\miFichero.txt");
```

- ✓ A partir del objeto `File` creamos el fichero físicamente, con la siguiente instrucción, que devuelve un boolean con valor true si se creó correctamente, o false si no se pudo crear:

```
fichero.createNewFile()
```

Para **borrar un fichero**, podemos usar la clase `File`, comprobando previamente si existe, del siguiente modo:

- ✓ Fijamos el nombre de la carpeta y del fichero con:

```
File fichero = new File("C:\\prueba", "agenda.txt");
```

- ✓ Comprobamos si existe el fichero con `exists()` y si es así lo borramos con:

```
fichero.delete();
```

Para **crear directorios**, podríamos hacer:

```
package crearcarpetas;

import java.io.File;
/**
 *
 * @author JJBH
 */
public class CrearDirec {

    public static void main(String[] args) {
      try {
          // Declaración de variables
```

```
          String directorio = "C:\\micarpeta";
          String varios = "carpeta1/carpeta2/carpeta3";

          // Crear un directorio
          boolean exito = (new File(directorio)).mkdir();
          if (exito)
              System.out.println("Directorio: " + directorio + " creado");
          // Crear varios directorios
          exito = (new File(varios)).mkdirs();
          if (exito)
              System.out.println("Directorios: " + varios + " creados");
   }catch (Exception e){
      System.err.println("Error: " + e.getMessage());
   }
    }

}
```

Para **borrar un directorio** con Java tenemos que borrar cada uno de los ficheros y directorios que éste contenga. Al poder almacenar otros directorios, se podría recorrer recursivamente el directorio para ir borrando todos los ficheros.

Se puede listar el contenido del directorio con:

```
File[] ficheros = directorio.listFiles();
```

y entonces poder ir borrando. Si el elemento es un directorio, lo sabemos mediante el método isDirectory,

8.- Almacenamiento de objetos en ficheros. Persistencia. Serialización.

Caso práctico

*Para la aplicación de la clínica veterinaria **María** le propone a **Juan** emplear un fichero para guardar los datos de los clientes de la clínica. -Como vamos a guardar datos de la clase Cliente, tendremos que serializar los datos.*

¿Qué es la **serialización**? Es un proceso por el que **un objeto se convierte en una secuencia de bytes** con la que más tarde se podrá reconstruir el valor de sus variables. Esto permite guardar un objeto en un archivo.

Para serializar un objeto:

- ✓ éste debe **implementar el interface** `java.io.Serializable`. Este interface no tiene métodos, sólo se usa para informar a la JVM (Java Virtual Machine) que un objeto va a ser serializado.
- ✓ Todos los objetos incluidos en él tienen que implementar el interfaz Serializable.

Todos los **tipos primitivos en Java son serializables** por defecto. (Al igual que los arrays y otros muchos tipos estándar).

Para leer y escribir objetos serializables a un stream se utilizan las clases java: `ObjectInputStream` y `ObjectOutputStream`.

En el siguiente ejemplo se puede ver cómo leer un objeto serializado que se guardó antes. En este caso, se trata de un String serializado:

```
FileInputStream fich = new FileInputStream("str.out");

ObjectInputStream os = new ObjectInputStream(fich);

Object o = os.readObject();
```

Así vemos que `readObject` lee un objeto desde el flujo de entrada fich. Cuando se leen objetos desde un flujo, se debe tener en cuenta qué tipo de objetos se esperan en el flujo, y se han de leer en el mismo orden en que se guardaron.

Para serializar un fichero basta con implementar el interface `Serializable`.

Verdadero. ☐ Falso. ☐

Para saber más

En el siguiente enlace a puedes ver un poco más sobre serialización.

Serialización en Java. http://nereida.deioc.ull.es/~cleon/psd/0405/prct/p5/pspdf/p5.ps

Más sobre serialización en Java. http://www2.uah.es/jcaceres/cursos/java/archivos/capsulas/capsula_serializable.pdf

8.1.- Serialización: utilidad.

La serialización en Java se desarrolló para utilizarse con RMI. RMI necesitaba un modo de convertir los parámetros necesarios a enviar a un objeto en una máquina remota, y también para devolver valores desde ella, en forma de flujos de bytes. Para datos primitivos es fácil, pero para objetos más complejos no tanto, y ese mecanismo es precisamente lo que proporciona la serialización.

El método `writeObject` se utiliza para guardar un objeto a través de un flujo de salida. El objeto pasado a `writeObject` debe implementar el interfaz `Serializable`.

```
FileOutputStream fisal = new FileOutputStream("cadenas.out");

ObjectOutputStream oos = new ObjectOutputStream(fisal);

Oos.writeObject();
```

La serialización de objetos se emplea también en la arquitectura de componentes software JavaBean. Las clases bean se cargan en herramientas de construcción de software visual, como NetBeans. Con la paleta de diseño se puede personalizar el bean asignando fuentes, tamaños, texto y otras propiedades.

Una vez que se ha personalizado el bean, para guardarlo, se emplea la serialización: se almacena el objeto con el valor de sus campos en un fichero con extensión .ser, que suele emplazarse dentro de un fichero .jar.

Para saber más

En este enlace a puedes ver un vídeo en el que se crea una aplicación sobre serialización. No está hecha con NetBeans, sino con Eclipse, pero eso no presenta ningún inconveniente.

http://www.youtube.com/watch?v=4eU6WMOVMh4&feature=player_embedded