

*PROGRAMACIÓN

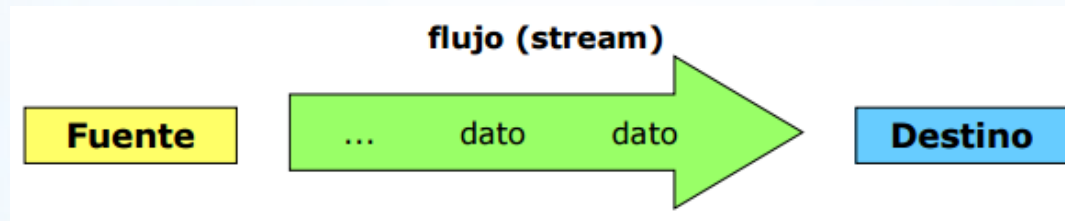
TERCER TRIMESTRE

2020/05/21

Tutoría - Ut8. Lectura y Escritura de información.
Ficheros de texto y serialización.

FLUJO DE DATOS - Stream

- * La manera de representar las entradas y salidas de datos en *Java*, y en muchos lenguajes de programación actuales, es a base de **streams** (flujos de datos). Podemos definir un flujo de datos como una secuencia ordenada de datos que se transmite desde una fuente hasta un destino.
- * Un *stream* es una conexión entre el programa y la fuente o destino de los datos. La información se traslada *en serie* (un carácter a continuación de otro) a través de esta conexión.



- * Se utiliza para:
 - Representar la lectura/escritura de archivos.
 - Escribir en el teclado, enviar al monitor...
 - La comunicación a través de Internet (TCP/IP)
 - La lectura de información a través de un puerto serie.

CLASIFICACIÓN DE LOS FLUJOS

- * Según el tipo de información que manejan:
 - Flujos de bytes: clases *InputStream* y *OutputStream*
 - Flujos de caracteres: clases *Reader* y *Writer*

- * Según la dirección del flujo de datos:
 - Entrada: *InputStream*, *Reader*
 - Salida: *OutputStream*, *Writer*
 - Lectura/Escritura: *RandomAccessFile*

- * Según la forma en la que se realiza el acceso.
 - Secuencial
 - Aleatorio - (*RandomAccessFile*)

TIPOS DE FLUJOS

- * Existen dos tipos de flujos, flujos de bytes (byte streams) y flujos de caracteres (character streams).
- * 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.

CLASES RELATIVAS A FLUJOS

Algunas de las clases del paquete **java.io** relativas a flujos más usadas son:

- * **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.

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.

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.

RUTAS DE LOS FICHEROS

Siempre que se trabaja con un fichero que estará almacenado en un dispositivo de almacenamiento permanente, se deben usar, para su localización la ruta de los ficheros tal y como se usan en MS-DOS, o Windows, es decir, por ejemplo:

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

O bien:

```
c:/datos/carpetadatos/fichero.dat
```

Cuando operamos con rutas de ficheros, el carácter separador entre directorios o carpetas pueden 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, de manera que nuestras aplicaciones sean lo más portables posibles se pueden hacer funciones que nos transformen una ruta en la adecuada según el separador del sistema.

TRABAJANDO CON FICHEROS. La clase File

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

- 1. Abrir o bien crear** si no existe el fichero.
- 2. Hacer las operaciones** que necesitemos.
- 3. 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, debe saltar una excepción para que nuestro programa no finalice de forma abrupta.

FICHEROS. La clase File

- * La clase File nos permite instanciar cualquier elemento del sistema de ficheros o archivos y por lo tanto no nos proporciona métodos de acceso a los archivos, sino operaciones a nivel de sistema de archivos (listado de archivos, crear carpetas, borrar ficheros, cambiar nombre, ...).
- * Se encuentra en el paquete java.io. Un objeto File representa un archivo o un directorio y sirve para obtener información (permisos, tamaño,...).
- * Esta clase dispone de varios constructores. El parámetro path indica el camino hacia el directorio donde se encuentra el archivo, y name indica el nombre del archivo:
 - * `File(String path)`
 - * `File(String path, String name)`
- * Si se utiliza como único argumento una cadena, esta representa el nombre del archivo. También puede recibir, opcionalmente, un segundo parámetro con una ruta segunda que se define a partir de la posición de la primera.
 - * `File archivo1=new File("/datos/bd.txt");`
 - * `File carpeta=new File("datos");`
- * El primer formato utiliza una ruta absoluta y el segundo una ruta relativa. En Java el separador de archivos tanto para Windows como para Linux es el símbolo /.
- * Otra posibilidad de construcción es utilizar como primer parámetro un objeto File ya hecho. A esto se añade un segundo parámetro que es una ruta que cuenta desde la posición actual.
 - * `File carpeta1=new File("c:/datos");` // ó `c\\datos`
 - * `File archivo1=new File(carpeta1,"bd.txt");`

FICHEROS. La clase File. Algunos métodos

método	USO
<code>boolean exists()</code>	Devuelve true si existe la carpeta o archivo.
<code>boolean canRead()</code>	Devuelve true si el archivo se puede leer
<code>boolean canWrite()</code>	Devuelve true si el archivo se puede escribir
<code>boolean isHidden()</code>	Devuelve true si el objeto File es oculto
<code>boolean isDirectory()</code>	Devuelve true si el objeto File es una carpeta y false si es un archivo o si no existe.
<code>boolean mkdir()</code>	Intenta crear una carpeta y devuelve true si fue posible hacerlo
<code>boolean delete()</code>	Borra la carpeta y devuelve true si puedo hacerlo
<code>boolean isFile()</code>	Devuelve true si el objeto File es un archivo y false si es carpeta o si no existe.
<code>boolean renameTo(File f2)</code>	Cambia el nombre del archivo por el que posee el archivo pasado como argumento. Devuelve true si se pudo completar la operación.
<code>boolean delete()</code>	Borra el archivo y devuelve true si puedo hacerlo
<code>long length()</code>	Devuelve el tamaño del archivo en bytes (en el caso del texto devuelve los caracteres del archivo)

TIPOS DE FICHEROS/ARCHIVOS.

Para manejar correctamente un fichero es esencial tener en cuenta el tipo de contenido que este almacenará.

- * Si el contenido es texto, el acceso será secuencial. Esto implica que si se necesita acceder al carácter contenido en la posición i , será necesario recorrer desde el primero hasta el $i-1$.
- * Si el contenido es binario, el acceso podrá realizarse de forma secuencial o mediante acceso directo. Este último permite acceder a un byte ubicado en una determinada posición sin necesidad de recorrer los anteriores (de forma similar a como se accedería al elemento de un array)
- * Contenido binario serializado, que nos permite el almacenamiento de objetos de una clase mediante la transformación de estos a un conjunto de bytes, proceso que se denomina serialización.

FICHEROS DE TEXTO

- * Trabajan con ficheros de caracteres, los del Bloc de notas por ejemplo.
- * Se pueden realizar lecturas y grabaciones carácter a carácter o bien por líneas.
- * Se utilizarán las clases `FileInputStream` y `BufferedInputStream` para la lectura con el método `read()` y las clases `FileOutputStream` y `BufferedOutputStream` con el método `write(caracter)` para salida o grabación.
- * `FileReader` y `BufferedReader()`, junto con el método `readLine()` se utilizan para la lectura de líneas.
- * En el caso de archivos es importante utilizar el **buffer** puesto que la tarea de escribir en disco es muy lenta respecto a los procesos del programa y realizar las operaciones de lectura/escritura de golpe y no de una en una hace mucho más eficiente el acceso.

ARCHIVOS DE TEXTO .- Lectura

* Ejemplo:

```
String texto = new String();
try {
    FileReader fr = new FileReader( "archivo.txt");
    entrada = new BufferedReader(fr);
    String s;
    while((s = entrada.readLine()) != null)
        texto += s;
    entrada.close();
}
catch (FileNotFoundException fnfex) {
    System.out.println("Archivo no encontrado: " + fnfex);
}
catch (IOException ioex) {
    ...
}
```

SERIALIZACIÓN

- * Consiste en convertir **objetos** en una secuencia de bytes para posteriormente almacenarlos o transmitirlos.
- * Para poder serializar un objeto, éste deberá implementar la interface *Serializable* y además todas sus propiedades deben ser serializables.
- * Todos los tipos primitivos implementan esta interface, así que el único problema es que intentemos grabar alguna propiedad perteneciente a una clase creada por nosotros, para lo cual, esa clase deberá ser serializable.
- * Las clases con las que se puede hacer esta lectura y grabación son *FileOutputStream*, *FileInputStream*, *ObjectInputStream* y *ObjectOutputStream*, estas últimas con sus métodos `readObject()` y `writeObject()`.
- * Los objetos del tipo **static** no pueden ser serializados y deberían almacenarse de forma independiente.

SERIALIZACIÓN.- Ejemplo


```
import java.io.*;

public class TestSerializable {

    public static void main(String[] args) {
        Persona persona1 = new Persona("Antonia", "Palomero");
        Persona persona2 = new Persona("Juan", "Alonso");
        ObjectOutputStream dos;
        try{
            dos = new ObjectOutputStream(new FileOutputStream("prueba.txt"));
            dos.writeObject(persona1);
            dos.writeObject(persona2);
            dos.close();
        }
        catch (FileNotFoundException e){
            e.printStackTrace();
        }
        catch (IOException e){
            e.printStackTrace();
        }
    }
}


import java.io.Serializable;

public class Persona implements Serializable{
    String nombre;
    String apellidos;
    Persona (String nombre, String apellidos){
        this.nombre = nombre;
        this.apellidos = apellidos;
    }
}
```



SERIALIZACIÓN.- Ejemplo

```
public class TestSerializable {  
  
    public static void main(String[] args) {  
        ObjectInputStream dis;  
        Persona p1=null,p2=null;  
        try{  
            dis = new ObjectInputStream(new FileInputStream("prueba.txt"));  
            p1 = (Persona)dis.readObject();  
            p2 = (Persona)dis.readObject();  
        }  
        catch (FileNotFoundException e){  
            e.printStackTrace();  
        }  
        catch (IOException e){  
            e.printStackTrace();  
        }  
        catch (ClassNotFoundException e) {  
            e.printStackTrace();  
        }  
        System.out.println(p1.nombre);  
        System.out.println(p2.nombre);  
    }  
}
```



Console X

<terminated> TestSerializable [Java Application] C
Antonia
Juan