UD1. Acceso a ficheros

Módulo: Acceso a datos

Unión Europea
Fondo Social Europeo
El FSE invierte en tu futuro

Germán Gascón Grau
g.gascongrau@edu.gva.es

Introducción

Los **sistemas de archivos** son una parte fundamental de los sistemas operativos.

Permiten **organizar** y **almacenar** datos de **manera eficiente** en discos y otros dispositivos de almacenamiento.

Cada sistema operativo utiliza su propio sistema de archivos, optimizado para sus características y necesidades.

En esta presentación veremos:

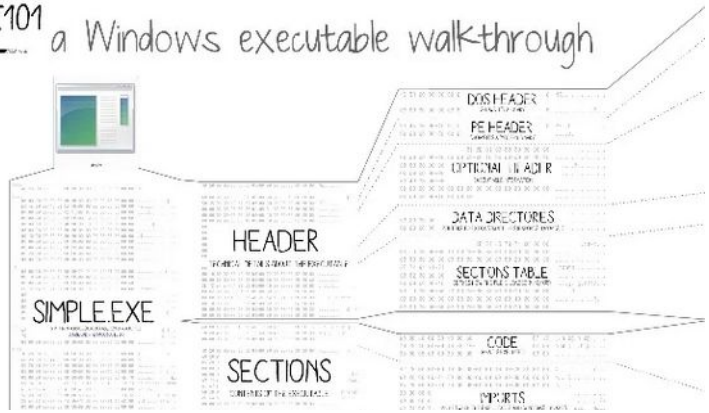
- Los sistemas de archivos en los principales sistemas operativos (Windows, Linux y macOS).
- Cómo Java simplifica el trabajo con archivos mediante una serie de clases

Fichero

Un **fichero** o **archivo** es un conjunto de bytes almacenado en un dispositivo de memoria de forma permanente.

Los archivos tienen una serie de **propiedades**: nombre, extensión, tamaño, ruta donde están ubicados, etc.

Los archivos están formados por **registros** y cada registro por **campos**.



Sistemas de archivos

Un **sistema de archivos** es la estructura que utilizan los sistemas operativos para almacenar y organizar archivos y directorios en un dispositivo de almacenamiento.

Permite **guardar**, **leer**, **escribir** y **gestionar datos** de forma eficiente.

Ejemplos de sistemas de archivos:

- **FAT** (File Allocation Table) - Usado en dispositivos USB y tarjetas SD.
- **NTFS** (New Technology File System) - Principal sistema de archivos en Windows.
- **ext4** (Fourth Extended Filesystem) - Usado principalmente en Linux.
- **APFS** (Apple File System) - Sistema de archivos moderno usado en macOS.

Sistemas de archivos en Windows

NTFS (New Technology File System)

- Sistema de archivos principal en Windows.
- Soporta permisos avanzados, cifrado, compresión y recuperación ante fallos.
- Ejemplos de uso:
  - Disco principal de un ordenador con Windows (C:).
  - Almacenamiento de datos corporativos por su sistema de permisos.

FAT32 / exFAT

- FAT32 es compatible con muchos dispositivos, pero tiene limitaciones en el tamaño de archivo (4 GB).
- exFAT es una versión mejorada, común en discos externos y memorias USB.

Sistemas de archivos en Linux

ext4 (Fourth Extended Filesystem)

- El sistema de archivos más común en distribuciones Linux.
- Ventajas: alta eficiencia, tolerancia a fallos y grandes capacidades de almacenamiento.

XFS

- Diseñado para alto rendimiento y grandes volúmenes de datos.

Btrfs (B-Tree File System)

- Ofrece snapshots y alta flexibilidad para sistemas avanzados.
- Uso típico:
  - Servidores y sistemas críticos que requieren alta fiabilidad.

Sistemas de archivos en macOS

APFS (Apple File System)

- Sistema de archivos moderno diseñado para dispositivos Apple.
- Optimizado para discos SSD y alto rendimiento.
- Características como snapshots, cifrado y clonación rápida.

HFS+ (Hierarchical File System Plus)

- Utilizado antes de APFS.
- Todavía en uso en algunos discos antiguos y dispositivos no actualizados.

Abstracción en Java

Java permite abstraernos tanto del Sistema Operativo como del sistema de archivos empleado a través de una serie de clases.

- File
- Files
- FileReader / FileWriter
- BufferedReader / BufferedWriter
- PrintWriter
- FileInputStream / FileOutputStream
- RandomAccessFile
- ObjectInputStream / ObjectOutputStream

La clase File

Un objeto de la clase **File** representa un archivo o directorio del sistema de archivos.

La propiedad estática **File.separator** permite indicar el separador correcto de cada Sistema Operativo.

Ejemplo de rutas en Windows: “C:\\Windows\\System32\\”

Ejemplo de rutas en Unix: “/home/ggascon/”

Constructores de la clase File

Constructor	Descripción
File(String pathname)	Este constructor crea un objeto File al que se le debe pasar la ruta a un archivo o directorio.
File(String parent, String child)	En este constructor se debe pasar la ruta y el nombre del archivo o directorio.
File(File parent, String child)	Como el objeto File puede representar un directorio o un archivo, el primer parámetro funcionará como directorio y el segundo parámetro será el nombre del archivo.

Métodos de la clase File

Método	Descripción
boolean isDirectory()	Sirve para saber si estamos trabajando con un archivo o con un directorio.
boolean isFile()	Complementario del anterior, sirve para saber si estamos trabajando con un archivo o con un directorio.
boolean exists()	Para asegurarnos de que el archivo existe realmente.
boolean delete()	Permite eliminar el archivo o directorio al que hace referencia el objeto F*ile*.
boolean renameTo(File dest)	Permite renombrar el archivo o directorio al que hace referencia el objeto F*ile*. Se le pasa como parámetro un objeto File con el nuevo nombre.
boolean createNewFile()	Crea el fichero vacío, si no existe y devuelve true. Si ya existe devuelve false. Puede generar IOException.

Métodos de la clase File

Método	Descripción
boolean canRead()	Nos permite averiguar si tenemos permiso de lectura sobre el archivo.
boolean canWrite()	Nos permite averiguar si tenemos permiso de escritura sobre el fichero.
String getPath()	Devuelve la ruta con la que se ha creado el objeto File.
String getAbsolutePath()	Devuelve la ruta absoluta del archivo (incluye el nombre del archivo).
String getName()	Devuelve el nombre del archivo.
String getParent()	Devuelve el directorio padre (del que cuelga el archivo o directorio).

Métodos de la clase File

Método	Descripción
long length()	Devuelve el tamaño del archivo en bytes.
boolean mkdir()	Crea un directorio.
String[] list()	Devuelve un array de objetos S*tring* con los nombres de los archivos / directorios que contiene el directorio.
String[] list(FilenameFilter filter)	Devuelve un array de objetos S*tring* con los nombres de los archivos / directorios que contiene el directorio que cumplen el filtro indicado (interfaz).
File[] listFiles()	Devuelve un array de objetos F*ile* con los nombres de los archivos / directorios que contiene el directorio.

La clase Files

La clase **Files** permite realizar operaciones comunes con archivos de forma sencilla a través de sus métodos estáticos:

Métodos para crear

- Files.createFile(), Files.createDirectory(), Files.createDirectories(), Files.createLink(), Files.createSymbolicLink(), Files.createTempFile(), Files.createTempDirectory().

Métodos para copiar

- Files.copy()

Métodos para borrar

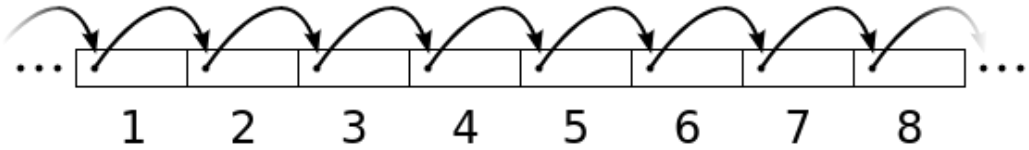
- Files.delete(), Files.deleteIfExists()

Métodos para obtener información

- Files.exists()

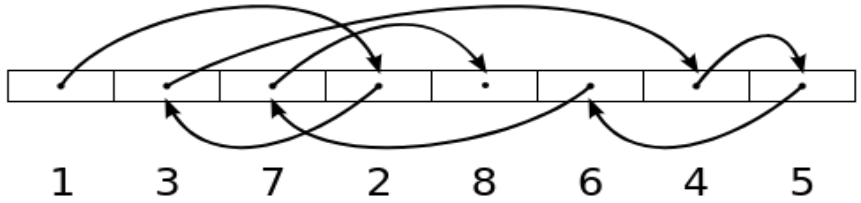
Formas de acceso a un archivo

Acceso **secuencial**



- Si se quiere acceder a un dato, se debe acceder previamente a las anteriores.
- Las inserciones se hacen al final, no es posible escribir entre los datos.

Acceso **aleatorio**



- Se puede acceder a la información en cualquier orden.
- Las lecturas y escrituras se pueden hacer donde sean necesarias.

Operaciones sobre ficheros (I)

Creación

- El fichero se crea en disco con un nombre.
- Se hace la operación una única vez.

Apertura

- Para poder trabajar con el contenido del archivo se debe abrir previamente.

Lectura

- Transferir información del archivo a la memoria principal, a través de variables.

Escritura

- Transferir información de la memoria al archivo, a través de variables.

Cierre

- El archivo debe cerrarse cuando acabamos de trabajar con él.

Operaciones sobre ficheros (II)

Altas

- Añadir un nuevo registro en el archivo.

Bajas

- Eliminar del archivo algún registro
  - Eliminación lógica: cambiar el valor de algún campo del registro
  - Eliminación física: reescribir el archivo en otro sin incluir el registro y renombrar el original.

Modificaciones

- Cambiar el contenido de algún registro.
- Implica la búsqueda del registro.

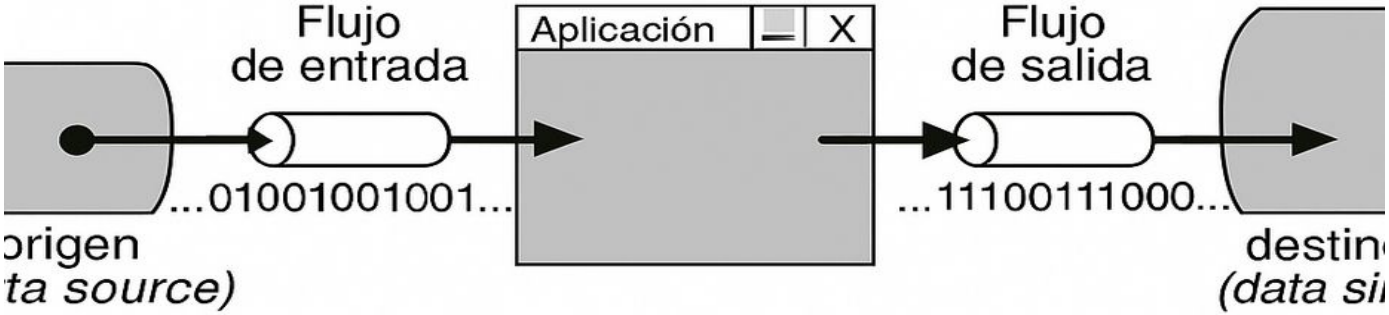
Consultas

- Buscar un registro determinado.

Flujos o Streams

Un **Stream** es un flujo de datos que representa una secuencia de elementos que se pueden procesar de forma declarativa.

Los streams permiten trabajar con colecciones de datos de manera más sencilla y concisa, usando operaciones como filtrado, transformación, mapeo y reducción sin modificar los datos originales.



Tipos de Flujos o Streams

Flujos de **caracteres** (16 bits)

- Lectura y escritura de caracteres (**texto**)
- Clases: **Reader** y **Writer**

Flujos de **bytes** (8 bits)

- Lectura y escritura de datos **binarios**.
- Clases: **InputStream** y **OutputStream**

Archivos de texto (I)

Guardan caracteres alfanuméricos en un formato estándar (ASCII, Unicode, UTF8, ...)

Acceso secuencial a la información de forma secuencial.

- Clase **FileReader**: Para leer caracteres
  - **FileNotFoundException** (No existe el nombre del archivo o no es válido)
- Clase **FileWriter**: Para escribir caracteres
  - **IOException** (El disco está lleno o protegido contra escritura)

Archivos de texto (II)

Pasos para trabajar con archivos de texto:

1. Invocar a la clase **File** (este paso es opcional).
2. Crear el flujo de entrada con **FileReader** o flujo de salida con **FileWriter** (puede implicar la creación del archivo).
3. Hacer las operaciones de **lectura** / **escritura** deseadas.
4. **Cerrar** los flujos de datos

Constructores de la clase FileReader

Constructor	Descripción
FileReader(File file)	Crea un objeto *FileReader* a partir del objeto File que apunta al archivo desde el que se quiere leer.
FileReader(String fileName)	Crea un objeto *FileReader* a partir del nombre del archivo desde el que se quiere leer. Se le pasa como parámetro la ruta donde se encuentra el archivo.

* Pueden lanzar la excepción *FileNotFoundException*.

Métodos de la clase FileReader

Método	Descripción
int read()	Lee un carácter y lo devuelve como un entero. Devuelve -1 cuando llega al final del archivo.
int read(char[] cbuf)	Lee 'n' caracteres (n <= cbuf.length) Devuelve el número de caracteres leídos. Los caracteres leídos están en las posiciones [0..n-1] del array cbuf. Devuelve -1 si el archivo se ha acabado.
int read(char[] cbuf, int off, int len)	Lee 'n' caracteres. Devuelve el número de caracteres leídos. Los caracteres leídos están en las posiciones [off ... off + len] del array cbuf. Devuelve -1 si el archivo se ha acabado.
void close()	Cierra el archivo.

* Todos los métodos anteriores pueden lanzar la excepción *IOException*.

Constructores de la clase FileWriter

Constructor	Descripción
FileWriter(File file)	Dado un objeto F*ile* que será el archivo sobre el que se quiere escribir, construye un objeto F*ileWriter* sobre él.
FileWriter(File file, boolean append)	Dado un objeto F*ile* que será el archivo sobre el que se quiere escribir, construye un objeto *FileWriter* sobre él al que se pueden **añadir datos** (No sobrescribe).
FileWriter(String fileName)	Dada la ruta del archivo sobre el que se quiere escribir, construye un objeto *FileWriter* sobre él.
FileWriter(String fileName, boolean append)	Dada la ruta del archivo sobre el que se quiere escribir, construye un objeto *FileWriter* sobre él al que se pueden **añadir datos** (No sobrescribe).

Métodos de la clase FileWriter

Métodos	Descripción
void write(int c)	Escribe un carácter
Writer append(char c)	Añadir un carácter a un fichero
void write(char [] cbuf)	Escribe en el fichero del array de caracteres.
void write(char [] cbuf, int off, int len)	Escribe en el fichero los 'n' caracteres del array cbuf a partir de la posición off.
void write(String str)	Escribe en el archivo la cadena 'str'.
void write(String str, int off, int len)	Escribe los caracteres de la cadena 'str' a partir de la posición off.
void flush()	Asegura que todos los caracteres queden bien escritos en disco, sin cerrar el archivo.
void close()	Cierra el fichero, asegurando que todo queda bien escrito en disco.

* Todos los métodos anteriores pueden lanzar la excepción *IOException*.

Ejemplo con flujos de caracteres

```
File f1 = new File ( "/home/ggascon/archivo1.txt");
// Crear los flujos de datos
try (FileReader lector = new FileReader(f1);
     FileWriter escritor = new FileWriter(f1)){
    // Ejemplos de operaciones de lectura / escritura
    int i = lector.read();
    escritor.write("Cadena a escribir");
} catch (IOException e) {
    System.out.println("Error: " +  e.getMessage());
}
```

Uso de Buffers

Un **buffer** es una **memoria intermedia** del programa que se utiliza para **mejorar el rendimiento** de las operaciones que requieren E/S.

Las operaciones de lectura / escritura sobre el archivo, se realizan sobre el buffer y no sobre el archivo, es decir, en realidad se lee / escribe en la **memoria RAM**, ya que esta es órdenes de magnitud más rápida que el almacenamiento externo.

El sistema operativo, cuando lo considere oportuno, guardará el contenido del buffer en el archivo.

Por este motivo no hay que quitar el **pendrive** sin darle a la operación de "**Extraer de forma segura**"

El programador también puede asegurarse que los datos del buffer se guardan a disco invocando al método **flush**().

La clase Buffered

Las clases **FileReader**, **FileWriter** y **FileInputStream** y **FileOutputStream** (que veremos para el manejo de archivos binarios) realizan las operaciones de lectura y escritura directamente sobre el dispositivo de almacenamiento.

Si la aplicación hace estas operaciones de forma muy repetida, su ejecución será más lenta.

Para mejorar esta situación, se pueden utilizar junto con las clases anteriores la clase **Buffered**.

La clase Buffered

La palabra Buffered hace referencia a la capacidad de almacenamiento temporal en la lectura y escritura:

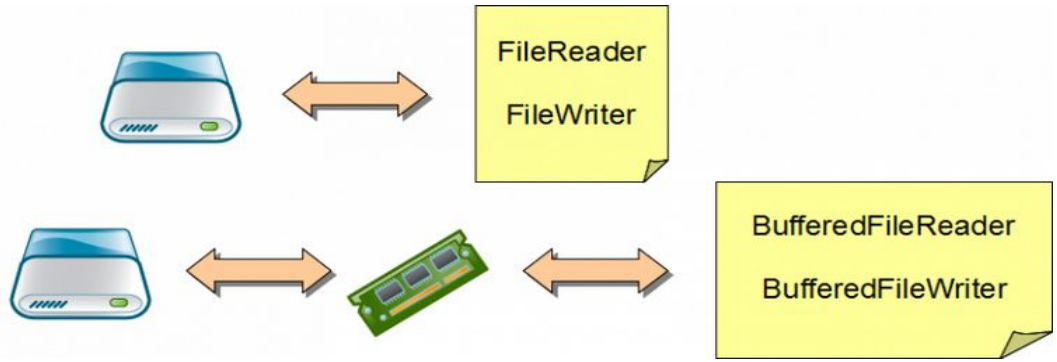
- En las operaciones de **lectura** se guardan **más datos de los que realmente se necesitan** en un momento determinado, de forma que en una operación posterior de lectura, es posible que los datos ya estén en memoria y no sea necesario acceder de nuevo al dispositivo.

- En las operaciones de **escritura**, los datos se guardan **en memoria** y no se vuelcan a disco hasta que no haya una cantidad suficiente de datos para mejorar el rendimiento. Cuidado esto puede ser peligroso sino se gestionan bien las excepciones.

Flujos de caracteres

Clase **Reader**

- FileReader
- BufferedReader



Clase **Writer**

- FileWriter
- BufferedWriter

La clase BufferedReader

La clase **BufferedReader** dispone de métodos para leer líneas completas.

Constructor	Descripción
BufferedReader(Reader in) * Necesitamos un FileReader	Crea un buffer para almacenar los caracteres del flujo de entrada (a través del cual se lee la información del archivo).
métodos	**Descripción**
String readLine()	Lee una línea de texto.
int read()	Lee un solo carácter.
void close()	Cierra el *buffer*.

* Los métodos anteriores pueden lanzar la excepción *IOException*.

La clase BufferedWriter

La clase **BufferedWriter** dispone de métodos para escribir líneas completas.

Constructor	Descripción
BufferedWriter(Writer out) * Necesitamos un FileWriter	Crea un buffer para almacenar los caracteres del flujo de salida (a través del cual se escribirá en el archivo).
métodos	**Descripción**
void newLine()	Escribe una línea en blanco.
void write(int c)	Escribe un carácter.
void write(String s)	Escribe una cadena.
void close()	Cierra el *buffer*.

* Los métodos anteriores pueden lanzar la excepción *IOException*.

Ejemplo con BufferedReader

```
// Crear el objeto File
File f1 = new File("/home/ggascon/archivo1.txt");
// Crear los flujos de datos
try (FileReader lector = new FileReader(f1); // entrada
     // Crear el buffer
     BufferedReader dadesLector = new BufferedReader(lector)) {
    // Operaciones lectura
    String linea = dadesLector.readLine();
    // No hace falta cerrar ya que estamos utilizando el closeable
} catch (IOException ioe) {
    ioe.printStackTrace();
}
```

Constructores de la clase PrintWriter

Constructor	Descripción
PrintWriter(File file)	Crea un objeto de tipo *PrintWrite* a partir de un fichero.
PrintWriter(Writer out)	Crea un objeto de tipo *PrintWrite* a partir de un *FileWriter*.
PrintWriter(String fileName)	Crea un objeto de tipo *PrintWrite* a partir de la ruta del archivo.

Métodos de la clase PrintWriter

Métodos	Descripción
void print(String s)	Escribe una cadena.
void println()	Escribe un salto de línea.
void print(int x)	Escribe un entero.
void println(char x)	Escribe un carácter y un salto de línea.

* Los métodos anteriores pueden lanzar la excepción *IOException*.

* print() y println() son similares a los de System.out

Ficheros binarios

Almacenan secuencias de dígitos binarios que no son legibles directamente por el usuario como ocurre con los archivos de texto.

Clases:

- **FileInputStream**: para lectura de datos.
- **FileOutputStream**: para escritura de datos.

Ficheros binarios (I)

Almacenan secuencias de dígitos binarios que no son legibles directamente por el usuario como ocurre con los archivos de texto.

Clases:

- **FileInputStream**: para lectura de datos.
- **FileOutputStream**: para escritura de datos.

Ficheros binarios (I)

Pasos para trabajar con ficheros binarios:

1. Invocar a la clase **File** (opcional)
2. Crear el flujo de entrada con **FileInputStream** o flujo de salida con **FileOutputStream** (implica la creación del archivo)
3. Realizar las **operaciones** de **lectura** / **escritura** deseadas.
4. **Cerrar** el archivo

Constructores de la clase FileInputStream

Constructor	Descripción
FileInputStream(File file)	Crea un objeto de tipo *FileInputStream* para leer datos desde el objeto F*ile* creado previamente.
FileInputStream(String fileName)	Crea un objeto de tipo *FileInputStream* para leer datos. Recibe como parámetro la ruta donde se encuentra el fichero con los datos.

Métodos de la clase FileInputStream

Método	Descripción
int read()	Lee un byte y lo devuelve pasado a entero. Devuelve -1 cuando llega al final del archivo
int read(byte[] b)	Lee 'n' bytes (n <= b.length) Devuelve el número de bytes leídos. Los bytes leídos están en las posiciones [0..n-1] del array b. Devuelve -1 si el archivo se ha acabado.
int read(byte[] b, int off, int len)	Lee 'len' bytes. Devuelve el número de bytes leídos. Los bytes leídos están en las posiciones [off ... off + len] del array b. Devuelve -1 si el archivo se ha acabado.
void close()	Cierra el archivo.
int available()	Una estimación del número de bytes que quedan por leer.

Constructores de la clase FileOutputStream

Constructor	Descripción
FileOutputStream(File file)	Crea un objeto de tipo *FileOutputStream* para escribir en el fichero al que hace referencia el objeto File.
FileOutputStream(File file, boolean append)	Crea un objeto de tipo *FileOutputStream* para añadir datos (no sobreescribir) el archivo al que hace referencia el objeto File.
FileOutputStream(String fileName)	Crea un objeto de tipo *FileOutputStream* para escribir datos. Recibe como parámetro la ruta donde se encuentra el archivo.
FileOutputStream(String fileName, boolean append)	Crea un objeto de tipo *FileOutputStream* para añadir datos. Recibe como parámetro la ruta donde se encuentra el archivo.

Métodos de la clase FileOutputStream

Método	Descripción
void write(int b)	Escribe un byte.
void write(byte[] b)	Escribe en el fichero del array de bytes.
void write(byte[] b, int off, int len)	Escribe los 'len' bytes del array b en el fichero, a partir de la posición off.
void close()	Cierra el fichero, asegurando que todo está bien escrito en disco.

Ejemplo con FileInputStream

```
// Crear el objeto File
File f1 = new File("/home/ggascon/archivo1.txt");
// Crear los flujos de datos
try (FileInputStream inputStream = new FileInputStream(f1)) { //
entrada
    // Operaciones lectura
    int dato = inputStream.read();
    // Mas operaciones si es necesario

    // No hace falta cerrar ya que estamos utilizando el closeable
} catch (IOException ioe) {
    System.out.println("Error: " + ioe.getMessage());
}
```

Ficheros de acceso aleatorio o directo

La clase **RandomAccessFile** permite el acceso directo cualquier posición de un archivo binario.

Permite abrir los ficheros:

- En modo lectura ("r")
- En modo escritura ("w")
- En modo lectura / escritura ("rw")

Ficheros de acceso aleatorio o directo

Al abrir este tipo de archivos se crea un **puntero** qua apunta al lugar del archivo donde se realizará la siguiente operación de lectura o escritura.

Cada vez que se hace una **lectura** o **escritura**, el puntero **se posiciona automáticamente** en el siguiente carácter.

Constructores de la clase RandomAcessFile

Constructor	Descripción
RandomAccessFile(File file, String modo)	Se crea el objeto a partir de un objeto File. El modo puede ser "r", "w" o "rw".
RandomAccessFile(String name, String modo)	Se crea el objeto pasándole como parámetro la ruta del archivo. El modo puede ser "r", "w" o "rw".

Métodos de la clase RandomAcessFile (I)

Método	Descripción
long getFilePointer()	Devuelve la posición actual del puntero.
void seek(long pos)	Coloca el puntero en la posición 'pos'. El primer carácter se encuentra en la posición 0.
long length()	Devuelve la longitud del archivo en bytes.
int skipBytes(int n)	Desplaza el puntero desde la posición actual el número de bytes indicado en el parámetro (n bytes).

Métodos de la clase RandomAcessFile (I)

Método	Descripción
int read()	Devuelve el byte leído en la posición marcada por el puntero.
final String readLine()	Devuelve la cadena de caracteres leída desde la posición actual del puntero hasta el primer salto de línea que encuentre.
public void write(int b)	Escribe en el fichero el byte pasado como parámetro.
public final void writeBytes(String s)	Escribe en el archivo la cadena pasada como parámetro. No incluye salto de línea, por lo tanto si se quiere añadir, se deberá incluir en la cadena de caracteres: writeBytes(cadena + "\ n").

Métodos de la clase RandomAcessFile (II)

Hay un método para cada tipo de datos básico: readChar, readInt, readDouble, readBoolean, ...

También existe un método write para cada tipo de datos básico: writeChar, writeInt, writeDouble, writeBoolean, ...

Archivo de properties

Los archivos de properties permiten simplificar la gestión de archivos cuyo contenido son parejas clave/valor, habitualmente utilizado en archivos de configuración.

```
1  #Archivo de configuración
2  #Sun Sep 21 23:31:11 CLST 2014
3  puerto=56730
4  ip=192.168.1.100
```

puerto e ip serían las claves, y 56730 y 192.168.1.100 los valores.

La clase **Properties** de Java manipular este tipo de archivos de forma muy sencilla.

Clase Properties

Extiende a HashTable, y por tanto tiene los mismos métodos que esta, y además:

load permite cargar el HashTable con las parejas clave/valor del archivo (InputStream/Reader) indicado como parámetro. Por ejemplo podríamos hacer:

```
properties.load(new FileReader("archivo.properties"));
properties.load(new BufferedReader(new FileReader("archivo.properties")));
properties.load(new FileInputStream("archivo.properties"));
```

Para cargar el HashTable desde un recurso de la aplicación:

```
properties.load(getClass().getResourceAsStream("archivo.properties"));
```

Clase Properties

getProperty método sobrecargado que permite obtener el valor asociado a cualquiera de las propiedades contenidas en un objeto Properties.

String getProperty(String key)

String getProperty(String key, String defaultValue)

```
String language = properties.getProperty("language");
language = properties.getProperty("language", "ES")
```

Clase Properties

setProperty permite establecer el valor asociado a una determinada clave.

Object setProperty(String key, String value)

```
properties.setProperty("language", "FR");
```

Clase Properties

El valor contenido por las propiedades de un objeto Properties se perderán si no se almacenan en un archivo.

store permite guardar los valores establecidos en el archivo (OutputStream/Writer) pasado como parámetro.

void store(Writer writer, String comments)

void store(OutputStream out, String comments)

```
properties.store(new BufferedWriter(new FileWriter("archivo.properties")),
"Ejemplo");
```

Serialización

La **serialización** es la técnica que permite transformar un objeto en memoria (con sus atributos y, en algunos casos, su estado) en un formato lineal que pueda:

- Guardarse en un archivo o base de datos.
- Transmitirse por una red.
- Interoperar entre diferentes plataformas o lenguajes.

El proceso inverso se llama **deserialización**: reconstruir el objeto original a partir de esa representación serializada.

Guardar el estado de los objetos

Los objetos tienen:

- comportamiento (métodos) → "viven" en la clase
- estado (atributos) → "viven" en el objeto

¿Cuando nos puede interesar guardar el estado de un objeto? Por ejemplo ...

- Guardar el estado de una partida (en un juego).
- Guardar el estado de un archivo antes de hacer una actualización.

Guardar el estado de los objetos

Con lo que hemos visto hasta ahora, escribiríamos el valor de cada instancia "a mano".

Pero podemos hacerlo de una forma OO "fácil".

Tendremos dos operaciones:

- Serializar
- Deserializar

Guardar el estado de los objetos

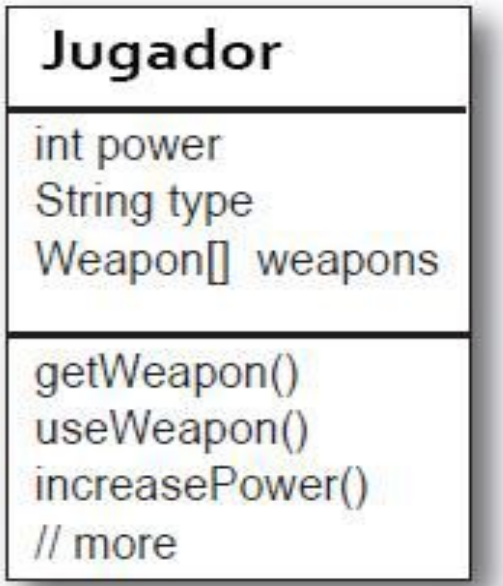
Si el objeto será utilizado por el **mismo programa Java** que lo ha creado:

- Podemos utilizar **serialización Java**
- Podemos utilizar algún formato estándar de transporte de datos **JSON** o **XML**

Si los datos serán utilizados por **otros programas**

- Utilizaremos el formato estándar de transporte de datos JSON o XML
- Los datos en formato JSON o XML pueden ser parseados de forma sencilla en el destino.

Ejemplo: Juego de aventura



Ejemplo: Juego de aventura

Como hemos comentado, tenemos dos opciones:

① **Opción 1**

Guardar los 3 objetos de los personajes serializados en un fichero

Se crea el fichero y se guarda el objeto serializado. El fichero si lo leemos como texto, no tendrá sentido:

```
"ÌsrGameCharacter
"%gê8MÛIpowerLjava/lang/
String;[weaponst[Ljava/lang/
String;xp2tlfur[Ljava.lang.String;≠“VÁ
È{Gxptbowtswordtdustsq~»tTrolluq~tb
are handstbig axsq~xtMagicianuq~tspe
llstinvisibility
```

② **Opción 2**

Escribir un fichero JSON

```
{jugadores: [
  {
     power: 50,
     type: Elf,
     weapons: [bow, sword, dust]
  },
  {
     power: 200,
     type: Troll,
     weapons: [bare, hands, big ax]
  }, .....
```

Serializar objetos en Java

1. Crear un **FileOutputStream**

```
FileOutputStream fileOutputStream = new FileOutputStream("MyGame.bin");
```

2. Crear un **ObjectOutputStream**

```
ObjectOutputStream os = new ObjectOutputStream(fileOutputStream);
```

3. Escribir el/los objeto/s

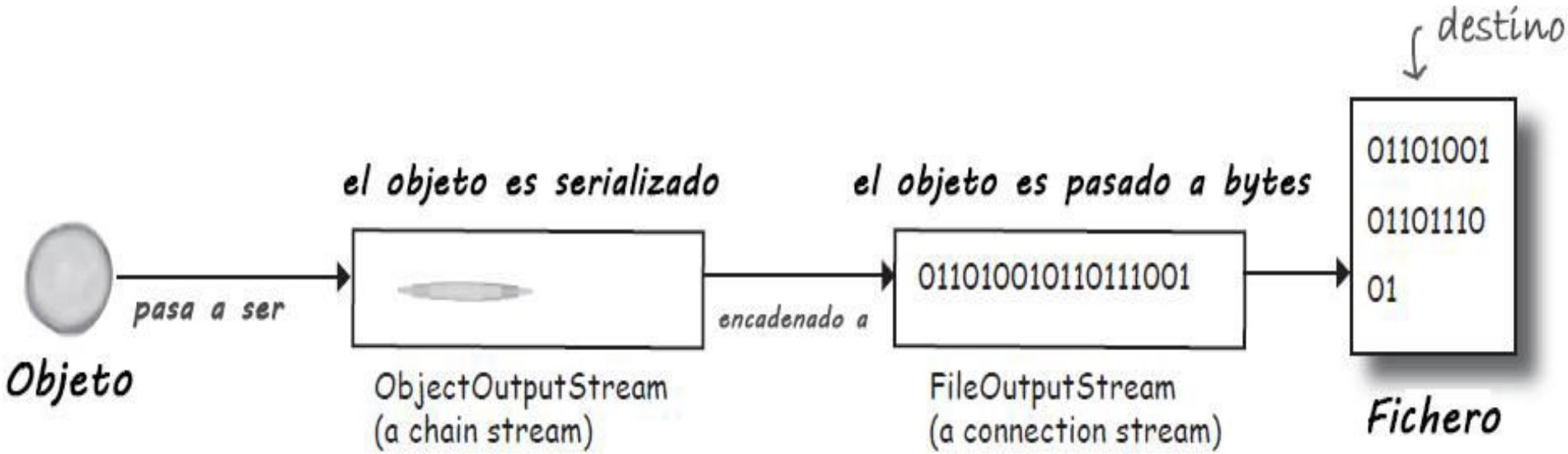
```
os.writeObject(entity1);

os.writeObject(entity2);
```

4. **Cerrar** el ObjectOutputStream

```
os.close();
```

Falta el tratamiento de excepciones

Proceso de serialización de Java



Interfaz Serializable

- Cuando un objeto se quiere serializar, deberá implementar la interfaz **Serializable**.

- Es una **interfaz sin métodos** a sobrescribir.

- Sus **subclases también serán serializables**, aunque no se especifique.

Reacción en cadena

- Cuando un objeto es serializado, todos los objetos a los que apuntan sus variables de instancia (también si son variables de referencia) también son serializados.
- Se produce una reacción en cadena automáticamente.
- La serialización guarda el esquema de objetos entero.

Serialización: todo o nada

- Si un objeto tiene como atributos objetos de otras clases que no son serializables, el proceso de serialización fallará.
- Obtendríamos una: **java.io.NotSerializableException**

Atributos transient

- Aquellos atributos que no deseemos que se guarden en el proceso de serialización podemos marcarlos con la palabra reservada **transient**.

```
public class Chat {
    private transient int currentId;
    private String username;
    private String[] messages;

    // Resto del código
}
```

Esto significa: no guardes el atributo currentId al serializar

- Los atributos username y messages serán guardados como parte del estado del objeto durante la serialización.

Deserialización de objetos en Java

- Es el proceso inverso de la serialización.
- Permite recuperar el estado original de los objetos tal y como estaban en el momento que fueron serializados.

Deserializar objetos en Java

1. Crear un **FileInputStream**

```
FileInputStream fileInputStream = new FileInputStream("MyGame.bin");
```

2. Crear un **ObjectInputStream**

```
ObjectInputStream os = new ObjectInputStream(fileInputStream);
```

3. **Leer** el/los objeto/s **en el mismo orden** en el que se serializaron

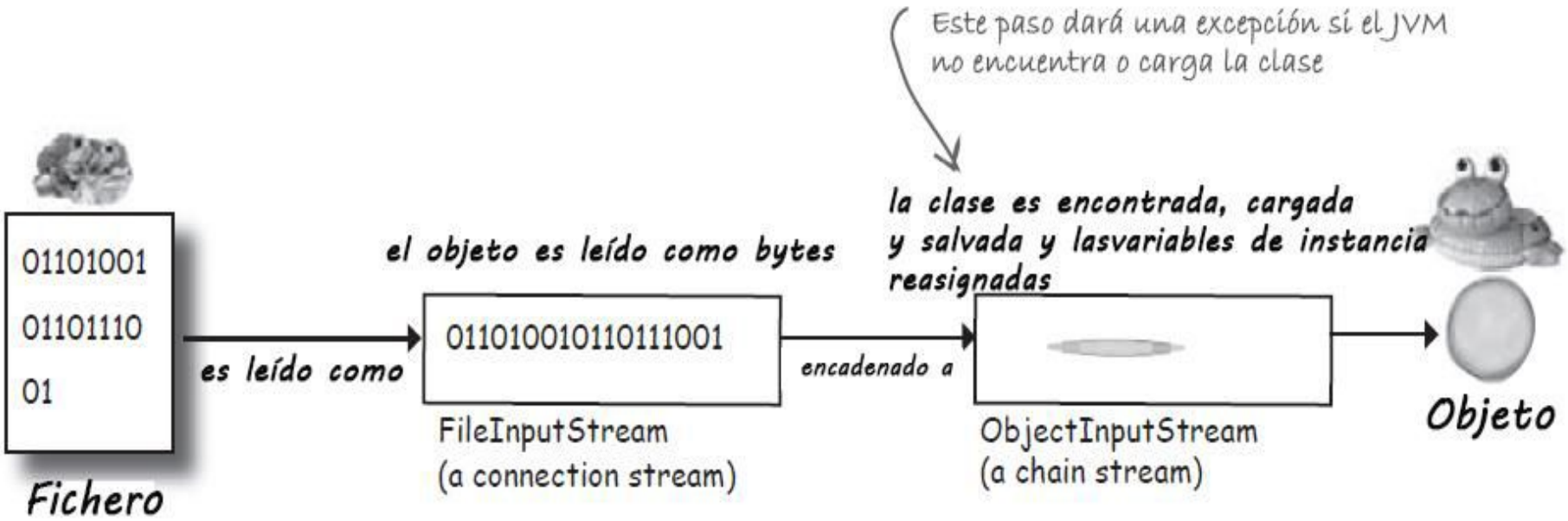
```
Entity entity1 = (Entity) os.readObject();

Object entity2 = (Entity) os.readObject();
```

4. **Cerrar** el ObjectInputStream

```
os.close();
```

Falta el tratamiento de excepciones

Proceso de deserialización en Java



Deserialización de objetos

- Los atributos del objeto contendrán los valores del objeto serializado.
- Los atributos transient tendrán valores por defecto para variables primitivas y valores null para las referencias a objetos.
- Las variables static no se guardan (no forman parte del objeto).

Resumen de la serialización/deserialización en Java

- Podemos guardar el estado de un objeto serializándolo.
- Aquellas clases que queramos que sean serializadas deben implementar la interfaz **Serializable**.
- Si una superclase es **Serializable**, la subclase automáticamente también lo es, aunque no lo especificamos explícitamente.
- Para **serializar** un objeto necesitamos un **ObjectOutputStream**.
- Para **serializar un objeto en un fichero** se necesita un **FileOutputStream** y encadenar a un **ObjectOutputStream**.
- Para serializar un objeto llamamos al método **writeObject(o)** mediante el ObjectOutputStream.
- Cuando un objeto es serializado, el grafo entero (el esquema de objetos) es serializado.

Resumen de la serialización/deserialización en Java

- Si algún objeto del grafo **no es serializable**, se producirá una **excepción** a menos que la variable sea transient.
- Marcamos una variable como **transient** si no es necesario serializarla. Las variables restauradas tendrán valores por defecto.
- En la deserialización leemos objetos con el método readObject(o), en el **mismo orden** en el que se guardaron inicialmente.
- El tipo de retorno del método **readObject(o)** es de tipo Object, por lo tanto se deberá **hacer un casting** a las variables para que tengan su tipo real.
- Los **atributos static no se serializan**. No tiene sentido guardar una variable que no forma parte del estado de cada objeto.

Serialización

- Si algún objeto del grafo **no es serializable**, se producirá una **excepción** a menos que la variable sea transient.
- Marcamos una variable como **transient** si no es necesario serializarla. Las variables restauradas tendrán valores por defecto.
- En la deserialización leemos objetos con el método readObject(o), en el **mismo orden** en el que se guardaron inicialmente.
- El tipo de retorno del método **readObject(o)** es de tipo Object, por lo tanto se deberá **hacer un casting** a las variables para que tengan su tipo real.
- Los **atributos static no se serializan**. No tiene sentido guardar una variable que no forma parte del estado de cada objeto.

Dudas / Sugerencias