Accés a dades

UF1: Persistència en fitxers

Fitxers

Un fitxer o arxiu és un conjunt de bits emmagatzemats en un dispositiu, com per exemple un disc dur. L'avantatge d'utilitzar fitxers és que les dades que guardem en ells es mantenen en el dispositiu encara que apaguem l'ordinador, és a dir, **no són volàtils**.

Les operacions bàsiques que es realitzen sobre qualsevol arxiu, independentment de la forma d'accés (seqüencial, directa, per bytes, per caràcters), són les següents:

- Crear el fitxer.
- Obrir el fitxer.
- Tancar el fitxer.
- Llegir les dades del fitxer.
- Escriure dades en el fitxer.

Classe File (I)

La classe 'File' és una classe que s'ha d'entendre com una referència a la ruta o localització de fitxers del sistema. **NO representa el contingut** de cap fitxer, sinó la ruta del sistema on es localitzen. Els objectes de la classe `File` representen rutes del Sistema de Fitxers.

Les instàncies de la classe `File` es troben estretament vinculades a la ruta amb la qual s'han creat. Això significa que les instàncies durant tot el seu cicle de vida només representaran una única ruta, la que se'ls va associar en el moment de la creació. La classe `File` **no disposa** de cap mètode ni mecanisme per modificar la ruta associada. En cas de necessitar noves rutes, caldrà sempre crear una nova instància i no serà possible reutilitzar les ja creades vinculant-les a rutes diferents.

La classe `File` encapsula pràcticament tota la funcionalitat necessària per gestionar un sistema de fitxers organitzat en arbre de directoris. És una gestió completa que inclou:

1. Funcions de manipulació i consulta de la pròpia estructura jeràrquica (creació, eliminació, obtenció de la ubicació, etc. de fitxers o carpetes)

2. Funcions de manipulació i consulta de les característiques particulars dels elements (noms, mida o capacitat, etc.)

3. Funcions de manipulació i consulta d'atributs específics de cada sistema operatiu i que, per tant, només serà funcional si el sistema operatiu amfitrió suporta també la funcionalitat. Ens referim, per exemple, als permisos d'escriptura, d'execució, atributs d'ocultació, etc.

Classe File (II)

Aquesta classe proporciona una sèrie d'utilitats relacionades amb els fitxers que ens proporcionen informació, el seu nom, els seus atributs, els directoris, etc. Pot representar el nom d'un fitxer en particular o els noms d'un conjunt de fitxers d'un directori, també es pot utilitzar per crear un nou directori, o un camí complet de directoris si aquest no existeix. Per crear un objecte File, es pot utilitzar qualsevol dels tres constructors següents:

- File(String directoriIFitxer):
  - en Linux: new File("/directori/fitxer.txt");
  - en Windows: new File("C:\\directori\\fitxer.txt");
- File(String directori, String fitxer):
- new File ("directori", "fitxer.txt");
- File(File directori, String fitxer):
- new File(new File("directori"),"fitxer.txt");

Exemples d'ús de la classe File on es mostren diferents maneres per declarar un fitxer:

- File fitxer1 = new File("C:\\EJERCICIS\\UNI1\\exemple1.txt");
- File fitxer1 = new File("/home/ejercicis/uni1/exemple1.txt");
- String directori = "C:/EJERCICIS/UNI1";
- File fitxer2 = new File(directori, "exemple2.txt");
- File fitxer3 = new File(direc, "exemple3.txt");

Classe File (III)

El següent exemple mostra la llista de fitxers en el directori actual. S'utilitza el mètode list(), que retorna un array de Strings amb els noms dels fitxers i directoris continguts en el directori associat a l'objecte File. Per indicar que estem en el directori actual creem un objecte File i li passem com a paràmetre ".":

```
import java.io.*;

public class VeureDir{

        public static void main (String[] args){

        System.out.println("Fitxers en el directori actual:");

        File f = new File(".");

        String[] arxius = f.list();

        for(int i=0; i<arxius.length;i++){

                System.out.println(arxius[i])

                }

        }

}
```

`public class VeureDir` Defineix una classe pública anomenada `VeureDir`. Aquesta classe conté el mètode principal (`main`), que és el punt d'entrada de l'aplicació.

`File f = new File(".");` Aquí es crea un objecte `File` que representa el directori actual. El punt (`"."`) indica el directori on s'està executant el programa.

`String[] arxius = f.list();` El mètode `list()` retorna un array de `String` amb els noms de tots els fitxers i directoris dins del directori actual. Aquest array es guarda a la variable `arxius`.

`for (int i = 0; i < arxius.length; i++)` Aquest bucle `for` recorre tots els elements de l'array `arxius`.

`System.out.println(arxius[i]);` Cada nom de fitxer o directori es mostra a la consola.

Classe File (IV)

Alguns mètodes importants de l'objecte File són:

Mètode	Descripció
getName()	Retorna el nom del fitxer o directori.
getPath()	Retorna el camí relatiu.
getAbsolutePath()	Retorna el camí absolut.
canRead()	Retorna *true* si el fitxer es pot llegir.
canWrite()	Retorna *true* si el fitxer es pot escriure.
length()	Ens retorna la mida del fitxer en bytes.
createNewFile()	Crea un nou fitxer buit associat a *File*, si i només si no existeix un fitxer amb aquest nom.
delete()	Esborra el fitxer o directori associat a *File*.
exist()	Retorna *true* si el fitxer/directori existeix.
getParent()	Retorna el nom del pare o *null* si no existeix.
isDirectory()	Retorna *true* si l'objecte *File* és un directori.
isFile()	Retorna *true* si l'objecte *File* és un fitxer.
mkdir()	Crea un directori amb el nom indicat en la creació de l'objecte *File*.
renameTo(File nouNom)	Reanomena l'arxiu.

Fluxos o streams

El sistema d'entrada/sortida en Java presenta una gran quantitat de classes que s'implementen en el paquet **java.io**. Utilitza l'abstracció de flux (stream) per tractar la comunicació d'informació entre una font i un destí. Aquesta informació pot estar en el disc dur, en memòria, en algun lloc de la xarxa o en un altre programa. Qualsevol programa que necessita obtenir informació d'una font o enviar informació a un destí ha d'obrir un stream. La vinculació del stream al dispositiu físic ho fa el sistema d'entrada i sortida de Java.

Es defineixen dos tipus de fluxos o streams:

● **Fluxos de bytes:** El seu ús està orientat a la lectura i escriptura de dades binàries. Totes les classes de fluxos de bytes són filles de les classes

InputStream i OutputStream.

● **Fluxos de caràcters**: Realitzen operacions d'entrada i sortida de caràcters. El flux de dades ve governat per les classes Reader i Writer.

Fluxos de bytes - InputStream

La classe **InputStream** representa les classes que produeixen entrades a la nostra aplicació des de diferents fonts. Aquestes fonts poden ser: un array de bytes, un objecte String, un arxiu, etc.

Els mètodes d'`InputStream` s'especifiquen tal com segueix:

- *int* **read()**. Llegeix el següent byte de dades del flux d'entrada i es retorna com un enter. Si no hi ha cap byte disponible perquè s'ha assolit el final de la seqüència, es retornarà -1. Si no hi ha cap dada disponible en el flux, el mètode es bloquejarà a l'espera d'alguna dada o de la marca que indiqui el final de la seqüència. En cas que s'hagi arribat al final de la seqüència de dades però no s'hagi detectat el final, es llançarà una excepció del tipus *IOException*.

- *int* **read(***byte[]buffer***)**. Llegeix un nombre de bytes del flux d'entrada i els emmagatzema dins el paràmetre anomenat *buffer* de tipus vector de *bytes*. El nombre de *bytes*llegits pot dependre de diversos factors externs i no se'n pot garantir cap mínim. Aquests, però, mai sobrepassaran la mida del vector de bytes on escriure'ls. El mètode retorna el nombre de bytes realment llegits com un enter. Si no hi ha cap byte disponible perquè en el flux s'ha arribat al final de la seqüència, es retornarà -1 com a indicador de final de lectura.

- *int* **read(***byte[]buffer*, *int offset*, *int len***)**. S'intenten llegir fins a *len* bytes de dades del flux d'entrada i els copia en el vector de *bytes* anomenat *buffer*. Com en el cas anterior, el nombre de *bytes*llegits pot dependre de diversos factors externs i no se'n pot garantir cap mínim. Aquests, però, mai sobrepassaran la quantitat indicada per *len, Offset* indica la posició dins de l'array *buffer* a partir de la qual es començaran a emmagatzemar les dades llegides. Si *offset* és 0, les dades es començaran a emmagatzemar des de l'inici del *buffer*. El mètode retorna el nombre de bytes realment llegits com un enter.

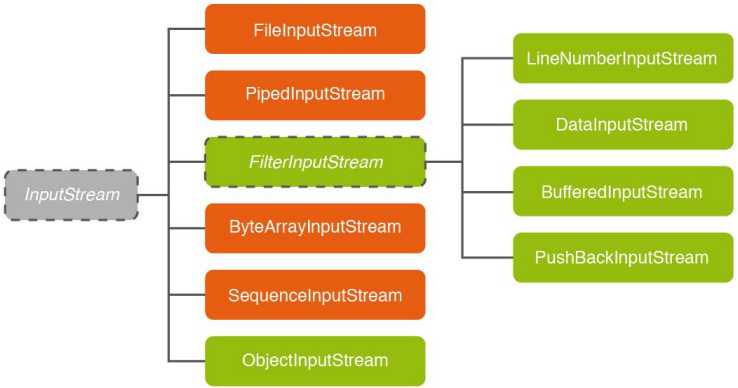
- *int* **available()**. Retorna el nombre de bytes que es troben disponibles per llegir (o saltar) en aquest flux d'entrada en el moment de l'execució del mètode.

- *long* **skip (***long bytesToSkip***)**. Salta i descarta *bytesToSkip* bytes de dades d'aquest flux d'entrada. Es retorna el nombre real de bytes omesos.

- void **close()**. Tanca aquest flux d'entrada i allibera els recursos del sistema associats.

Fluxos de bytes - InputStream

Jerarquia de les classes de Java que implementen fluxos orientats a bytes:



`DataInputStream` tindrà la funció de convertir els bytes del flux en dades de tipus bàsic de l'aplicació

i `BufferedInputStream` suporta un *buffer* (memòria intermèdia) extra per als fluxos d'entrada.

`LineNumberInputStream` afegeix numeració a cada una de les línies arribades des del flux. És a dir, cada vegada que detecta un salt de línia incrementa el recompte i afegeix l'índex a la nova línia.

`PushBackInputStream` és un flux d'entrada que permet retrocedir un byte en el flux a mida que avança la lectura.

Fluxos de bytes - OutputStream

La classe **OutputStream** inclou les classes que decideixen cap a on anirà la sortida. Per exemple a un array de bytes, a un arxiu, etc. El següent diagrama ens mostra la seva jerarquia.

Els mètodes d'`OutputStream` estan orientats a l'escriptura de la font de dades i s'especifiquen de la següent manera:

- *void* **write(***int byte***)**. Escriu un byte contingut al valor passat per paràmetre transferint-lo al flux de sortida. Si no pot escriure'l,es llança una excepció *IOException.*

- *void* **write(***byte[] data***)**. Escriu tots els bytes continguts al vector passat per paràmetre. Cal que *data* no sigui *null*, sino es llançarà també una excepció de tipus *NullPointerException*. Si no fos possible l'escriptura es llançaria un excepció.

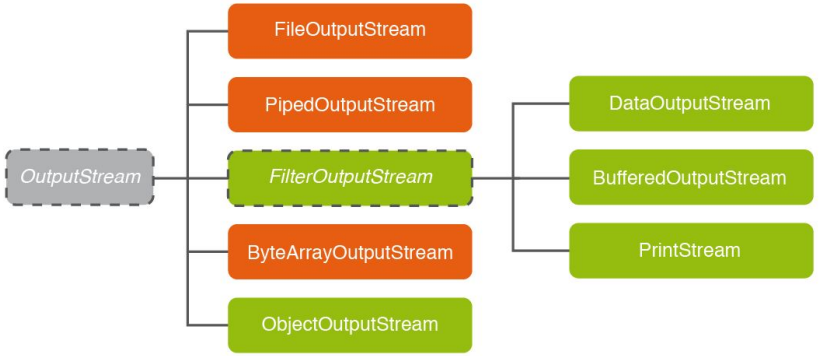
- *void* **write(***byte[] data, int offset, int len***)**. Escriu, si és possible, els *len* bytes ubicats al vector passat per paràmetre a partir de la posició *offset* del vector, transferint-los a la font de dades connectada per mitjà del flux. Si *len* fos negatiu o *offset* fos negatiu o *offset+len* fos més gran que la longitud del vector, es llançaria una excepció de tipus *IndexOutOfBoundsException*. Cal que *data* no sigui *null*, si no es llançarà també una excepció (*NullPointerException*). Si no fos possible l'escriptura es llançaria una excepció.

- *void* **flush()**. Sovint les fonts de dades poden tenir temps de processament elevats. En aquest casos serà normal l'ús de memòria intermèdia per minimitzar-ne els efectes. Aquest mètode buida la seqüència de sortida forçant a escriure els bytes que quedin encara a la memòria intermèdia.

- *void* **close()**. Tanca aquest flux de sortida i allibera els recursos del sistema associats.

Fluxos de bytes - OutputStream

Jerarquia de les classes de Java que implementen fluxos orientats a bytes:



`DataOutputStream` tindrà la funció de convertir els bytes del flux en dades de tipus bàsic de l'aplicació

`BufferedOutputStream`, suporta un *buffer* (memòria intermèdia) extra per als fluxos d'entrada.

`PrintStream` és un decorador amb utilitats destinades a la presentació de les dades (precisió en els decimals numèrics, format de les dates, etc.). Podríem dir que és una utilitat encarregada del format extern de les dades en contraposició a DataOutputStream, que s'encarrega de la representació interna de les dades (format intern).

Fluxos de caràcters - Reader / Writer

Les classes `Reader` i `Writer` formen la base per a la manipulació de fluxos de caràcters en Java. A diferència dels fluxos de bytes, que gestionen dades en forma de bytes (`InputStream`, `OutputStream`), `Reader` i `Writer` treballen amb caràcters Unicode, facilitant la manipulació de text.

- **`Reader`**: És una classe abstracta que defineix els mètodes per llegir dades en forma de caràcters. Els mètodes principals inclouen `read()` per llegir un sol caràcter o un array de caràcters, i `close()` per tancar el flux i alliberar recursos.
- **`Writer`**: També és una classe abstracta, però està dissenyada per a l'escriptura de caràcters. Els mètodes principals inclouen `write()` per escriure un sol caràcter, un array de caràcters o una cadena, i `flush()` per assegurar que totes les dades s'han escrit correctament.

Classes Concretes més Usades:

- **`FileReader`** i **`FileWriter`**: S'utilitzen per a la lectura i escriptura de caràcters en fitxers. Són les alternatives per a fitxers quan es treballa amb caràcters en comptes de bytes.
- **`CharArrayReader`** i **`CharArrayWriter`**: S'utilitzen per a la lectura i escriptura de caràcters en un array de caràcters. Aquests fluxos permeten treballar amb contingut de memòria com si fos un flux.
- **`BufferedReader`** i **`BufferedWriter`**: Aquestes classes afegeixen una capa de buferització als fluxos, millorant l'eficiència en la lectura i escriptura de caràcters, especialment quan es treballa amb grans volums de dades.

Fluxos de pont entre bytes i caràcters

Hi ha situacions on cal treballar amb fluxos de bytes (`InputStream`, `OutputStream`) i convertir-los en fluxos de caràcters o viceversa. Per aconseguir-ho, Java proporciona classes pont:

- **`InputStreamReader`**: Converteix un flux de bytes (`InputStream`) en un flux de caràcters (`Reader`). És útil quan estàs llegint dades de fonts que generen bytes (com fitxers binaris) però vols treballar amb caràcters.

  InputStream inputStream = new FileInputStream("input.txt");

  Reader reader = new InputStreamReader(inputStream, "UTF-8");

  Això llegeix bytes des de `input.txt` i els converteix a caràcters utilitzant la codificació `UTF-8`.

- **`OutputStreamWriter`**: Converteix un flux de caràcters (`Writer`) en un flux de bytes (`OutputStream`). És útil per escriure caràcters a una destinació que espera bytes.

  OutputStream outputStream = new FileOutputStream("output.txt");

  Writer writer = new OutputStreamWriter(outputStream, "UTF-8");

  Això escriu caràcters a `output.txt` convertint-los a bytes amb la codificació `UTF-8`.

Formes d’accedir a un fitxer

Quan treballem amb fitxers en programació, hi ha dues maneres principals d'accedir a la informació emmagatzemada en aquests fitxers:

1. **Accés Seqüencial:**
   - **Descripció:** En l'accés seqüencial, les dades o registres es llegeixen o s'escriuen en l'ordre en què es troben dins del fitxer. Això significa que per accedir a una dada específica, caldrà llegir totes les dades anteriors.
   - **Classes Utilitzades:**
     - **Per a bytes o dades binàries:** `FileInputStream` (per llegir) i `FileOutputStream` (per escriure).
     - **Per a caràcters:** `FileReader` (per llegir) i `FileWriter` (per escriure).
2. **Accés Directe o Aleatori:**
   - **Descripció:** Aquest tipus d'accés permet saltar directament a una dada o registre específic sense haver de llegir les dades anteriors. És a dir, es pot accedir a la informació en qualsevol ordre, de manera més ràpida i eficient.
   - **Classe Utilitzada:**
     - **Per a accés directe o aleatori:** `RandomAccessFile` permet tant llegir com escriure en qualsevol lloc del fitxer, en qualsevol moment.

Tipus d'Accés	Format	Classes Utilitzades
Seqüencial	Bytes o binari	`FileInputStream`, `FileOutputStream`
	Caràcters	`FileReader`, `FileWriter`
Directe o Aleatori	Tots dos	`RandomAccessFile`

Fitxers de text (I)

Per llegir caràcters d'un fitxer farem servir la classe `FileReader` i la classe `FileWriter` per escriure caràcters en un fitxer. Cada vegada que llegim o escrivim en un fitxer, ho hem de fer dins un bloc try – catch. Si utilitzem la classe FileReader, es pot generar l'excepció `FileNotFoundException`, i si utilitzem FileWriter es pot generar l'excepció `IOException`.

- `FileReader`**:** Els mètodes que proporciona la classe `FileReader` retornen el nombre de caràcters llegits, o -1 si s'ha arribat al final del fitxer.
  - Mètodes importants:
    - `int read()`: Llegeix un caràcter i el retorna com un enter (codi ASCII del caràcter).
    - `int read(char[] buf)`: Llegeix fins a `buf.length` caràcters. Els caràcters llegits del fitxer es van guardant a buf(Buffer). Retorna el nombre de caràcters llegits.
    - `int read(char[] buf, int desplacament, int n)`: Llegeix fins a n caràcters, començant per buf[desplacament] i retorna el nombre de caràcters llegits.
- `FileWriter`**:** Els mètodes que proporciona la classe `FileWriter` no retornen cap valor, els seus mètodes són **void**
  - Mètodes importants:
    - `void write(int c)`: Escriu un caràcter.
    - `void write(String str)`: Escriu una cadena de caràcters.
    - `void write(char[] buf)`: Escriu un array de caràcters.
    - `void write(char[] buf, int desplacament, int n)`: Escriu n caràcters de dades en l'array buf (buffer), començant per buf[desplacament].
    - `append(char c)`: Escriu un caràcter a un fitxer.

Fitxers de text (II)

Igual que abans declarem l'arxiu amb la classe `File`, i a continuació es crea el flux de sortida amb la classe `FileWriter`.

Hem de tenir en compte que si el fitxer existeix quan l'obrim sense especificar res, el que tenia emmagatzemat s'esborrarà. Si volem afegir caràcters al final de l'arxiu, utilitzarem la classe FileWriter, col·locant el valor **true** com a valor del segon paràmetre en el constructor:

FileWriter fit= new FileWriter(fitxer, true);

`FileReader` és una classe que permet llegir caràcter a caràcter des d'un fitxer de text. Tot i això, no disposa de mètodes específics per llegir línies completes de text. Si necessitem llegir línies senceres, podem utilitzar la classe `BufferedReader`, que proporciona el mètode `readLine()`, el qual llegeix una línia completa i la retorna com a cadena de text (o `null` si s'ha arribat al final del fitxer). Per utilitzar `BufferedReader`, normalment s'encadena amb un `FileReader`:

BufferedReader fitxer= new BufferedReader(new FileReader("NomFitxer"));

La classe `BufferedWriter` també deriva de la classe `FileWriter`. Per construir un `BufferedWriter` necessitem la classe `FileWriter`:

BufferedWriter fitxer= new BufferedWrriter(new FileWriter("NomFitxer"));

Podem utilitzar el mètode `newLine()`; per afegir un salt de línia.

La classe `PrintWriter`, que també deriva de `FileWriter`, disposa dels mètodes `print(String)` i `println(String)` , idèntics als de System.out, per escriure en un fitxer. Per a construir un `PrintWriter` necessitem la classe `FileWriter`.

Fitxers binaris (I)

Aquests tipus de fitxers emmagatzemen seqüències de dígits binaris que no són llegibles directament per l'usuari. Tenen l'avantatge que ocupen menys espai en disc. En Java, les dues classes que ens permeten treballar amb fitxers d'aquest tipus són `FileInputSream` i `FileOutputSream`, treballen amb fluxos de bytes i creen un enllaç entre el flux de bytes i el fitxer.

- Els mètodes que proporciona la classe `FileInputSream` per a la lectura són similars als mètodes de la classe `FileReader`:
  - Mètodes importants:
    - `int read()`: Llegeix un caràcter i el retorna com un enter.
    - `int read(byte[] buf)`: Llegeix fins a `buf.length` de bytes de dades. Els bytes llegits del fitxer es van guardant a buf(Buffer).
    - `int read(byte[] buf, int desplacament, int n)`: Llegeix fins a n bytes, començant per buf[desplacament] i retorna el nombre de bytesllegits.
- Els mètodes que proporciona la classe `FileOutputSream` per a l'escriptura són:
  - Mètodes importants:
    - `void write(int b)`: Escriu un byte.
    - `void write(byte[] buf)`: Escriu `buf.length` bytes.
    - `void write(byte[] buf, int desplacament, int n)`: Escriu n bytes de dades en l'array buf (buffer), començant per buf[desplacament].

Fitxers binaris (II)

Per afegir bytes al final de l'arxiu utilitzem `FileOutputSream` de la següent manera:

FileOutputStream fileout = new FileOutputStream(fitxer, true);

Per a llegir i escriure dades primitives (int, float, double, etc.) utilitzarem les classes `DataInputSream` i `DataOutputSream`. Aquestes classes proporcionen mètodes per a llegir i escriure tipus primitius, independentment del dispositiu on s'executin:

Mètodes per a la lectura	Mètodes per a l'escriptura
boolean readBoolean();	void writeBoolean(boolean v);
byte readByte();	void writeByte(int v);
	void writeBytes(String str);
char readChar();	void writeChar(int v);
	void writeChars(String str);
double readDouble();	void writeDouble(double v);
float readFloat();	void writeFloat(float v);
int readInt();	void writeInt(int v);
long readLong();	void writeLong(long v);
short readShort();	void writeShort(short v);
String readUTF();	void writeUTF(String str);

Fitxers binaris (III)

`FileInputStream` i **`FileOutputStream`** només tracten amb bytes de forma molt bàsica. No tenen la capacitat d'entendre o gestionar dades primitives com `int`, `float`, etc. en la seva forma nativa. Els fluxos de dades **`DataInputStream`** i **`DataOutputStream`** estenen aquesta funcionalitat per facilitar la manipulació de dades primitives.

Imagina que vols emmagatzemar o llegir un número sencer (`int`) o un nombre amb decimals (`double`) en un fitxer binari. Amb **`DataOutputStream`**, pots escriure directament aquestes dades primitives en el fitxer, i amb **`DataInputStream`**, pots llegir-les de nou de manera precisa.

```
import java.io.*;

public class ExempleDataOutputStream {

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

        File fitxer = new File("C:\\FichData.dat");

        FileOutputStream fileout = new FileOutputStream(fitxer);

        DataOutputStream dataOS = new DataOutputStream(fileout);

        // Escriu un int i un double al fitxer

        dataOS.writeInt(100);

        dataOS.writeDouble(123.45);        dataOS.close();

        }

}
```

```
public class ExempleDataInputStream {

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

        File fitxer = new File("C:\\FichData.dat");

        FileInputStream filein = new FileInputStream(fitxer);

        DataInputStream dataIS = new DataInputStream(filein);

        // Llegeix l'int i el double del fitxer

        int numero = dataIS.readInt();

        double valor = dataIS.readDouble();

        System.out.println("Numero: " + numero);

        System.out.println("Valor: " + valor);

        dataIS.close();

        }}
```

Objectes serialitzables

Serialització és el procés de convertir un objecte en una forma que es pugui emmagatzemar o transferir, com ara un fitxer o una transmissió de xarxa. En Java, això s'aconsegueix utilitzant la interfície `Serializable`. Si una classe implementa `Serializable`, això indica que els seus objectes poden ser serialitzats i, per tant, emmagatzemats com a dades binàries.

```
class Person implements Serializable {
```

Mètodes Clau

Quan es vol serialitzar o deserialitzar un objecte (recuperar-lo des del format binari), es fan servir les classes `ObjectOutputStream` i `ObjectInputStream`. Els dos mètodes més importants associats a aquests processos són:

- **`void writeObject(ObjectOutputStream stream) throws IOException`**: Aquest mètode escriu l'objecte en un flux de sortida (`ObjectOutputStream`), que es pot connectar a un fitxer per emmagatzemar l'objecte en format binari.
- **`void readObject(ObjectInputStream stream) throws IOException, ClassNotFoundException`**: Aquest mètode llegeix l'objecte des d'un flux d'entrada (`ObjectInputStream`), recuperant-lo del seu format binari original.

La classe **`ObjectOutputStream`** pot donar alguns problemes, per exemple, si escrivim dades en un fitxer, el tanquem, i després tornem a obrir-lo per afegir dades amb **`FileOutputStream(fitxer, true)`**. El problema és que escriu una nova capçalera al final dels objectes introduïts el primer cop i després es van afegint les noves dades. Això dóna problemes en llegir el fitxer, i es produeix l'excepció **`StreamCorruptedException`**.

Fitxers d’accés aleatori (I)

La classe `RandomAccessFile` de Java és una eina poderosa que permet treballar amb fitxers d'una manera diferent de les classes tradicionals d'entrada/sortida. Aquesta classe permet accedir al contingut d'un fitxer de forma **aleatòria**, el que significa que pots llegir i escriure dades en qualsevol posició del fitxer, sense seguir l'ordre seqüencial habitual. Això és especialment útil quan necessites modificar parts específiques d'un fitxer sense haver de llegir o escriure tot el contingut.

Característiques i ús de `RandomAccessFile`

1. **Creació d'un fitxer d'accés aleatori:** Pots crear un fitxer d'accés aleatori de dues maneres:
   - Passant el nom del fitxer com a cadena de text:

     RandomAccessFile fitxer = new RandomAccessFile("nom_fitxer", "modeAcces");

   - Utilitzant un objecte de la classe `File`

     ```
     File fitx = new File("nom_fitxer");

     RandomAccessFile fitxer = new RandomAccessFile(fitx, "modeAcces");
     ```

2. **Modes d'accés:**
   - `"r"`: Només lectura.
   - `"rw"`: Lectura i escriptura.

Fitxers d'accés aleatori (II)

3. **Funcionament de l'apuntador:** La classe `RandomAccessFile` treballa amb un **apuntador intern** que marca la posició actual dins del fitxer. Quan es crea un fitxer nou, l'apuntador es col·loca al començament (posició 0). A mesura que fas operacions de lectura (`read()`) o escriptura (`write()`), l'apuntador es mou automàticament per indicar la posició següent.

4. **Mètodes principals:**

- `long getFilePointer()`: Retorna la posició actual de l'apuntador dins del fitxer. Això et permet saber exactament on ets dins del fitxer.
- `void seek(long posicio)`: Mou l'apuntador a una posició específica dins del fitxer, comptant des del principi. Això és útil per accedir directament a la part del fitxer que t'interessa.
- `long length()`: Retorna la mida del fitxer en bytes. Això et permet saber quant ocupa el fitxer.
- `int skipBytes(int desplacament)`: Desplaça l'apuntador des de la posició actual un nombre determinat de bytes cap endavant.

Exemple d'ús

Imagina que tens un fitxer amb registres de dades i vols modificar només un registre en particular. Amb `RandomAccessFile`, pots anar directament a la posició d'aquest registre, modificar-lo i desar els canvis, tot això sense afectar la resta del fitxer. Aquesta capacitat d'accedir directament a qualsevol part del fitxer fa que `RandomAccessFile` sigui ideal per a aplicacions que necessiten un accés ràpid i flexible a les dades.

Treball amb fitxers XML

Els fitxers XML són eines molt versàtils que s'utilitzen en una àmplia varietat d'aplicacions informàtiques. Aquests fitxers permeten emmagatzemar dades en un format estructurat, i es poden utilitzar tant per transmetre informació entre sistemes com per guardar dades de manera estructurada per a usos futurs.

Ús dels Fitxers XML

Els fitxers XML es poden aplicar a diverses tasques, com ara:

- **Emmagatzematge de dades:** Poden contenir còpies de parts de bases de dades o ser utilitzats com a fonts de dades per a aquestes.
- **Configuració de programes:** Molts programes utilitzen fitxers XML per guardar la seva configuració.
- **Protocols de comunicació:** En protocols com SOAP, els fitxers XML s'utilitzen per enviar ordres a servidors remots i rebre les respostes.

Processadors XML (Parsers)

Per treballar amb fitxers XML i accedir a la seva informació, es fan servir processadors XML, també coneguts com a **parsers**. Aquests programes llegeixen el document XML i proporcionen accés al seu contingut i estructura. Hi ha dues principals tècniques de processament d'XML:

1. **DOM (Document Object Model):**
   - **Funcionament:** Aquest tipus de parser llegeix l'XML complet i emmagatzema tota l'estructura del document en memòria com un arbre. En aquest arbre, cada element XML es representa com un node, amb nodes pare, nodes fill i nodes finals (nodes sense descendents). Un cop creat l'arbre, es pot recórrer i analitzar l'XML fàcilment.
   - **Avantatges:** Ofereix una visió completa del document, permetent una navegació fàcil i la manipulació de l'estructura sencera.
   - **Inconvenients:** Requereix molts recursos de memòria i pot ser lent si el fitxer XML és gran o complex.
   - **Origen:** Té el seu origen en el consorci W3C, que defineix els estàndards per a XML i altres tecnologies web.
2. **SAX (Simple API for XML):**
   - **Funcionament:** A diferència del DOM, SAX processa l'XML de manera seqüencial. No crea un arbre complet del document en memòria. En canvi, llegeix el fitxer XML línia per línia i genera esdeveniments (com ara l'inici o fi d'un document, l'inici o fi d'una etiqueta, etc.) a mesura que troba elements XML. Cada esdeveniment invoca un mètode predefinit pel programador.
   - **Avantatges:** És molt eficient pel que fa a l'ús de memòria, ja que no necessita carregar tot el document a la vegada.
   - **Inconvenients:** No permet una navegació fàcil per l'estructura del document, ja que no guarda una visió completa de l'XML en memòria.

Accés a fitxers XML amb DOM

Components Principals per a l'Accés a Fitxers XML amb DOM

1. **Paquets Necessaris:**
   - **org.w3c.dom:** Aquest paquet conté les interfícies necessàries per treballar amb documents XML en forma d'arbre.
   - **javax.xml.parsers:** Proporciona classes abstractes com `DocumentBuilderFactory` i `DocumentBuilder`, que són essencials per crear i manipular el document DOM.
2. **DocumentBuilderFactory i DocumentBuilder:**
   - **DocumentBuilderFactory:** Aquesta classe és una fàbrica que ens permet obtenir una instància de `DocumentBuilder`, la qual és necessària per crear un document DOM.
   - **DocumentBuilder:** S'utilitza per construir el document DOM a partir d'una font de dades (com un fitxer o un InputStream).
3. **Interfícies Principals del DOM:**
   - **Document:** Representa un document XML complet. És l'arrel de l'arbre DOM i permet crear nous nodes.
   - **Element:** Representa un element XML, com ara `<empleat>`. Aquesta interfície permet manipular els elements i els seus atributs.
   - **Node:** Representa qualsevol node dins de l'arbre DOM, ja sigui un element, un text o un altre tipus de dades.
   - **NodeList:** Conté una llista de nodes fills d'un node.
   - **Attr:** Representa un atribut d'un element XML.
   - **Text:** Representa el text dins d'un element XML.
   - **CharacterData:** Inclou qualsevol text dins del document XML, permetent manipular-lo.
   - **DocumentType:** Proporciona informació sobre el tipus de document, incloent la definició DTD (`<!DOCTYPE>`).

Exemple pràctic: crear fitxer XML amb DOM (I)

1. Importar els paquets necessaris:

```
import org.w3c.dom.*;

import javax.xml.parsers.*;

import javax.xml.transform.*;

import javax.xml.transform.dom.*;

import javax.xml.transform.stream.*;

import java.io.*;
```

2. Crear una Instància de `DocumentBuilderFactory`:

```
DocumentBuilderFactory factory = DocumentBuilderFactory.newInstance();
```

Exemple pràctic: crear fitxer XML amb DOM (II)

3. Crear un Document Buit de nom document amb un Node Arrel “Empleats” i assignem la versió d'XML.

```
DocumentBuilder builder = factory.newDocumentBuilder();

DOMImplementation implementation = builder.getDOMImplementation();

Document document = implementation.createDocument(null, "Empleats", null);

document.setXmlVersion("1.0");
```

4. **Construir l'Arbre DOM:**

- Afegir node arrel:

```
Element arrel = document.createElement("empleat");

document.getDocumentElement().appendChild(arrel);
```

- Afegir Etiquetes d'Empleats:

```
Element elem = document.createElement("id");

Text text = document.createTextNode("123");

arrel.appendChild(elem);

elem.appendChild(text);
```

Exemple pràctic: crear fitxer XML amb DOM (III)

5. Crear la Font XML a partir del Document:

```
Source source = new DOMSource(document);
```

6. Especificar el fitxer de sortida:

```
Result result = new StreamResult(new File("Empleats.xml"));
```

7. Transformar el document en un fitxer XML:

```
Transformer transformer = TransformerFactory.newInstance().newTransformer();
transformer.transform(source, result);
```

8. Mostrar resultat per pantalla (opcional):

```
Result console = new StreamResult(System.out);
transformer.transform(source, console);
```

Accés a fitxers XML amb SAX

SAX (Simple API for XML) és una API basada en esdeveniments per a processar documents XML de manera seqüencial, és a dir, l'XML es llegeix i es processa element per element a mesura que es troba en el document. Això contrasta amb **DOM (Document Object Model)**, que carrega tot el document XML en memòria abans de processar-lo.

Característiques de SAX:

- **Seqüencial**: SAX llegeix el document XML seqüencialment, el que significa que consumeix menys memòria, especialment útil per a documents grans.
- **Basat en Esdeveniments**: Quan SAX troba parts específiques del document, genera esdeveniments com `startDocument()`, `endDocument()`, `startElement()`, `endElement()`, i `characters()`.
- **Sense Visió Global**: A diferència de DOM, no es té una visió completa del document perquè SAX només processa una petita part a la vegada.

Avantatges i Desavantatges de SAX

- **Avantatges**:
  - Baix consum de memòria, ideal per a grans fitxers XML.
  - Velocitat de processament elevada.
- **Desavantatges**:
  - No es pot accedir a qualsevol part del document en qualsevol moment (no hi ha navegació aleatòria).
  - La programació amb SAX pot ser més complexa que amb DOM.

Interfícies i Classes Clau en SAX

ContentHandler: Rep notificacions dels esdeveniments que es produeixen mentre es llegeix el document XML.

- Els mètodes més importants d'aquesta interfície són:
  - `startDocument()`: Notifica l'inici del document XML.
  - `endDocument()`: Notifica el final del document XML.
  - `startElement()`: Notifica l'inici d'un element en el document XML.
  - `endElement()`: Notifica el final d'un element en el document XML.
  - `characters()`: Notifica la presència de text dins d'un element XML.

DTDHandler: Gestiona esdeveniments relacionats amb la definició de tipus de document (DTD), com les entitats i les notacions.

ErrorHandler: Defineix com gestionar els errors que es puguin produir durant el processament del document XML.

- Els mètodes inclouen:
  - `warning(SAXParseException e)`: Gestió d'avisos.
  - `error(SAXParseException e)`: Gestió d'errors recuperables.
  - `fatalError(SAXParseException e)`: Gestió d'errors fatals.

EntityResolver: S'encarrega de resoldre entitats externes com a part de la lectura del document XML. Això és útil quan es vol evitar la descàrrega de recursos externs o quan es vol personalitzar la manera com es resolen les referències a entitats.

DefaultHandler: És una classe que proporciona implementacions per defecte per a totes les interfícies esmentades anteriorment (`ContentHandler`, `DTDHandler`, `ErrorHandler`, `EntityResolver`). Això permet al programador sobrescriure només els mètodes necessaris.

- És habitual utilitzar `DefaultHandler` com a punt de partida per implementar un parser SAX personalitzat.

Com implementar SAX (I)

En Java, per treballar amb SAX, es necessita crear un processador XML (`XMLReader`). Anteriorment, aquest es creava amb `XMLReaderFactory.createXMLReader()`, però aquest mètode està marcat com a obsolet en versions recents de Java. En lloc d'això, la forma recomanada de crear un `XMLReader` és utilitzant una instància de `SAXParserFactory` per crear un `SAXParser` i després obtenir el `XMLReader`.

1. **Incloure les Classes i Interfícies Necessàries:** SAX es troba en els paquets `org.xml.sax` i `org.xml.sax.helpers`. Aquestes classes permeten gestionar esdeveniments com l'inici i el final del document, elements i text.

```
import org.xml.sax.Attributes;

import org.xml.sax.InputSource;

import org.xml.sax.SAXException;

import org.xml.sax.XMLReader;

import org.xml.sax.helpers.DefaultHandler;

import javax.xml.parsers.SAXParser;

import javax.xml.parsers.SAXParserFactory;

import java.io.IOException;
```

Com implementar SAX (II)

2. **Crear un `SAXParser` i obtenir un `XMLReader`:** `SAXParserFactory` és una fàbrica que produeix instàncies de `SAXParser`, la qual cosa és una altra forma d'obtenir un `XMLReader` en Java sense utilitzar el mètode obsolet. Un cop tens un `SAXParser`, pots obtenir un `XMLReader` cridant al mètode `getXMLReader()`.

```
SAXParserFactory factory = SAXParserFactory.newInstance();

SAXParser saxParser = factory.newSAXParser();

XMLReader processador = saxParser.getXMLReader();
```

3. **Assignar un Gestor de Contingut:** El processador necessita saber quins mètodes ha de cridar quan es produeixin esdeveniments. Això es fa creant una classe que hereti de `DefaultHandler`, on es defineixen els mètodes que gestionaran els esdeveniments, com ara `startDocument()`, `endDocument()`, `startElement()`, etc.

```
GestioContingut gestor = new GestioContingut();

processador.setContentHandler(gestor);
```

4. **Llegir el Document XML:** El document es llegeix a partir d'un objecte `InputSource`, que es pot crear amb el nom del fitxer XML. El processament es realitza mitjançant el mètode `parse()` del processador.

```
InputSource fileXML = new InputSource("proves.xml");

processador.parse(fileXML);
```

Conversió fitxers XML amb XSL

XSL (Extensible Stylesheet Language) és una família de recomanacions del World Wide Web Consortium (W3C) que defineix com crear fulls d'estil en llenguatge XML.

- **Objectiu:** Presentar i transformar dades XML en formats visuals com HTML.
- **Components:** XSL es divideix en tres parts:
  - **XSLT:** Transformació XML a XML (o altres formats).
  - **XPath:** Llenguatge de consulta per seleccionar nodes en documents XML.
  - **XSL-FO:** Format de sortida per a la creació de documents imprimibles.

Procés de Transformació

Volem convertir un fitxer XML en un fitxer HTML mitjançant un fitxer XSL.

Passos generals:

1. **Crear Fitxers:**
   - **XML:** Conté les dades a transformar.
   - **XSL:** Defineix les regles de transformació per presentar les dades.
2. **Aplicar Transformació:**
   - Utilitzar el fitxer XSL per transformar el fitxer XML en un altre fitxer ja pot ser un altre XML un HTML etc...

Conversió fitxers XML amb XSL

S'ha de crear un objecte `Transformer`, que s'obté de crear una instància de `TransformerFactory` i aplicant el mètode `newTransformer` a la font XSL que utilitzarem per aplicar el full d'estils XSL a l'arxiu XML:

```
String fullEstil = "arxiu.xsl";

String arxiuXml = "arxiu.xml";

File paginaHtml = new File("arxiu.html");

FileOutputStream os = new FileOutputStream(paginaHtml);

Source estils = new Streamsource(fullestil);

Source dades = new StreamSource(arxiuXML);

Result result = new StreamResult(os);

Tranformer transformer =

TranformerFactory.newInstance().newTransformer(estils);

Transformer.transform(dades, result);
```

XSL: Defineix com presentar les dades.
XML: Conté les dades a transformar.
HTML: Fitxer resultant amb la presentació de les dades.

`Source estils`: Representa el fitxer XSL.
`Source dades`: Representa el fitxer XML.
`Result result`: Representa el fitxer HTML on s'escriurà el resultat.

`TransformerFactory.newInstance()`: Crea una instància de `TransformerFactory`.
`factory.newTransformer(estils)`: Crea un `Transformer` utilitzant el fitxer XSL com a full d'estil.

`transformer.transform(dades, result)`: Aplica la transformació i escriu el resultat al fitxer HTML.