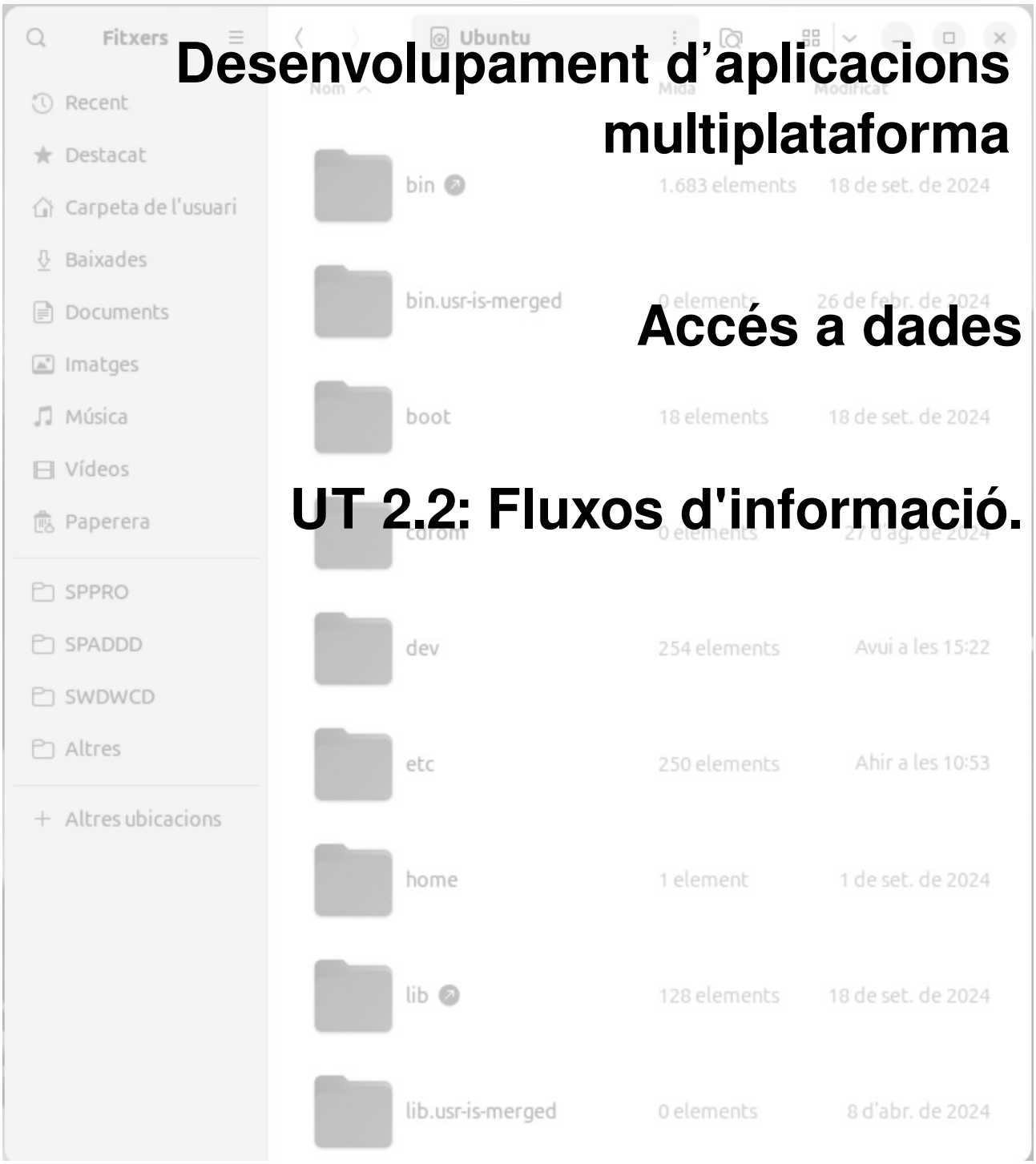
PAU CASESNOVES
Centre Integrat de Formació Professional



Desenvolupament d'aplicacions multiplataforma

Accés a dades

UT 2.2: Fluxos d'informació.

Joan Pons Tugores

Índex de continguts

Persistència de dades.... 3
Flux (Stream).... 4
Byte streams.... 5
Parèntesi: try-with-resources.... 7
Character streams.... 8
Buffered streams.... 9
Data streams.... 11
Object streams.... 13
Escriptura i lectura d'objectes complexos.... 15

Persistència de dades

Persistència de dades: com mantenir les dades entre execució i execució de l'aplicació. Tot el que havíem fet fins ara treballava amb la RAM de l'ordinador, per tant quan acabava el programa es perdia la informació que havia gestionat.

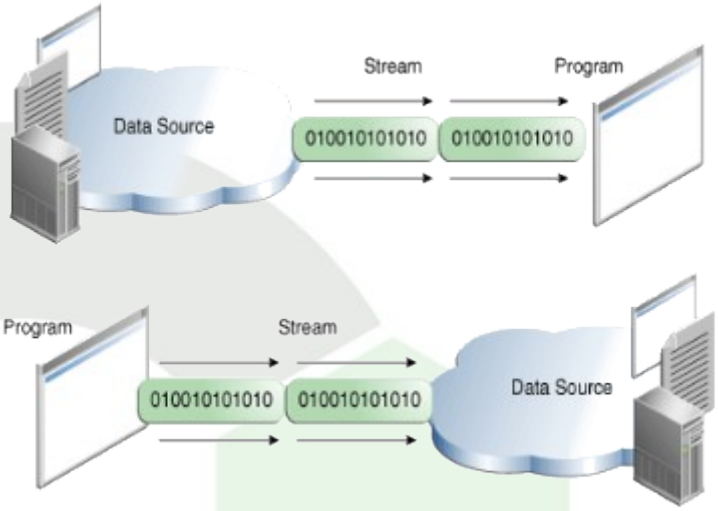
Bàsicament hi ha dues possibilitats per implementar la persistència:

- **Fitxers**: en java es tracten com a fluxos, *Streams*. Van molt bé per aplicacions petites: No muntarem un servidor de bases de dades per guardar només les dades de configuració de l'aplicació, per exemple.
- **Bases de dades**: Quan la quantitat i estructura de les dades és més complexa un fitxer o conjunt de fitxers no ens basten, llavors si que val la pena utilitzar un sistema gestor de bases de dades.

Anem a veure els *streams*.

Flux (Stream)

Un flux de dades és una seqüència de dades. Un programa utilitzarà un flux d'entrada, *input stream*, per a llegir dades d'un origen, i un flux de sortida, *output stream*, per a escriure dades a un destí.



Les operacions tant amb un com amb l'altre es realitzen amb un element a la vegada.

L'origen o el destí de les dades poden ser de diversos tipus: fitxers de disc, dispositius, altres programes, arrays de memòria, ... qualsevol cosa que pugui mantenir, generar o consumir dades.

Els fluxos poden manejar qualsevol tipus de dades, des de byte, fins objectes passant per tipus primitius.

Byte streams

Els nostres programes utilitzaran els fluxos de bytes quan volguem accedir a recursos a baix nivell. **No els hauríem d'utilitzar a no ser que sigui imprescindible**, ja que hi ha classes molt més especialitzades.

Els veurem perquè són la base, la superclasse, de tots els altres fluxos. Utilitzarem ***FileInputStream*** per a llegir un fitxer. El constructor espera la ruta del fitxer com a paràmetre:

```
String origen="/home/alumne/cara.jpg";

FileInputStream in = new FileInputStream(origen);
```

Totes les operacions amb *streams* poden llençar excepcions, per tant aniran dins un bloc *try-catch* o dins un mètode que les propagui.

Per a llegir utilitzarem *read*. Torna un *int* i no un *byte* perquè així pot tornar el valor -1 per indicar que ha acabat de llegir el fitxer. Per tant per llegir tots els bytes del fitxer:

```
while ((c = in.read()) != -1){

    byte b=(byte)c; //càsting a byte.
```

Els *streams* sempre s'han de tancar, millor dins un bloc finally per assegurar-nos. Deixar un recurs obert, com ara un fitxer, pot provocar problemes, per exemple per extreure un USB:

```
} finally {
      if (in != null)
            in.close();
}
```

Per escriure un fluxe de bytes podem utilitzar ***FileOutputStream***. El constructor també espera un String amb la ruta del fitxer on ha d'escriure les dades.

```
FileOutputStream out = new FileOutputStream(origen);
```

El mètode a utilitzar és write. Té com a paràmetre un int.

```
out.write(b);
```

Aquest mètode està sobrecarregat, per exemple amb un array de bytes com a paràmetre.

Les classes que defineixen els streams de bytes són abstractes, *InputStream* i *OutputStream*. Algunes subclasses són:

- FileInputStream i FileOutputStream: Per a treballar amb fitxers.
- ByteArrayInputStream i ByteArrayOutputStream: per a treballar amb arrays com a font o destinació de dades.
- AudioInputStream: Especialitzada en fonts d'audio.
- ObjectInputStream i ObjectOutputStream: Per treballar amb objectes.
- ...

Parèntesi: try-with-resources

Aquesta variant del try declara recursos, objectes que el programa ha de tancar quan ja no els necessiti més. S'assegura que els objectes declarats es tanquin, tant si el bloc de codi del try llença excepcions com si no.

Es poden utilitzar com a recursos qualsevol objecte que implementi la interfície *AutoCloseable*. Disponible a partir del Java 7.

```
try (FileInputStream in = new FileInputStream(origen)) {
        int c;
        while ((c = in.read()) != -1) {
            System.out.print((byte)c);
        }
}
```

Evidentment, **hi poden haver tants blocs catch com ens facin falta**. Abans d'executar-se, però, es tancaran els recursos declarats al try.

Si no capturam les excepcions les haurem de propagar.

Character streams

Java emmagatzema els caràcters en Unicode. Els fluxos de caràcters automàticament els transformen a i des de el conjunt de caràcters locals del sistema.

Totes les classes que implementen fluxos de caràcters descendeixen de *Reader* i *Writer*.

Per llegir fitxers de caràcters disposam de *FileReader* i *FileWriter*. Els constructors accepten, entre d'altres, la ruta del fitxer que llegiran o on escriuran.

```
try (FileReader in = new FileReader(origen)) {
```

O bé

```
try (FileWriter out = new FileWriter(origen)) {
```

El mètode *read* de ***FileReader*** llegeix un caràcter de l'stream. Si no n'hi ha cap de disponible bloca el programa. El torna en format int, per tant hem de fer el càsting a char. Podem comprovar l'estat del fluxe amb el mètode *ready*.

```
while (in.ready()) {

        char d = (char) in.read();
```

El mètode *write* de ***FileWriter*** escriu un caràcter al fitxer.

```
out.write('a');
```

Està sobrecarregat amb versions que reben per paràmetre String.

```
out.write("Hola");
```

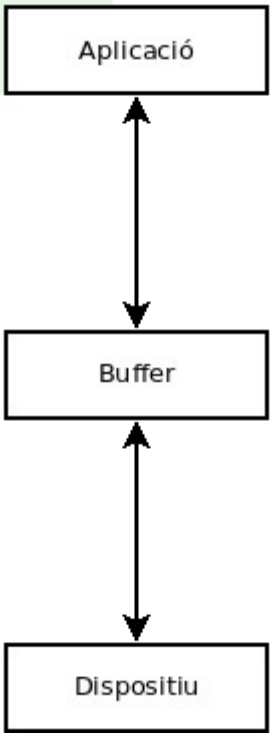
o arrays de char

```
char[] caracters={'H','o','l','a'};

out.write(caracters);
```

Tant *read* com *write* utilitzen *int* per manejar les dades llegides o escrites al fitxer.

Buffered streams



Els fluxos que hem vist fins ara no disposen de cap *buffer*. Això vol dir que cada petició de lectura o escriptura s'envia directament al SO, amb la conseqüent pèrdua d'eficiència, ja que normalment impliquen operacions de disc o xarxa que són molt costoses en temps.

Per millorar el rendiment, Java implementa fluxos amb buffer, amb una memòria intermèdia.

Durant la lectura, aquesta memòria intermèdia s'omple de cop i l'stream accedeix a les dades des de la memòria; en ser buida es torna a omplir.

Durant l'escriptura, les dades s'escriuen a aquesta memòria i en ser plena s'envien al disc o a la xarxa.

Disposam de 4 buffered streams: ***BufferedInputStream*** i ***BufferedOutputStream*** per *byte streams*, i ***BufferedReader*** i ***BufferedWriter*** per *streams* de caràcter.

Per a crear un *buffered stream*, al seu constructor li hem de passar un flux sense *buffer*:

```
BufferedReader inputStream =
        new BufferedReader(new FileReader("original.txt"));

BufferedWriter outputStream =
        new BufferedWriter(new FileWriter("sortida.txt"));
```

Per llegir *BufferedReader* té el mètode *readLine*. Torna un *String*, i quan acaben les dades de l'stream torna *null*.

```
while ((linia = in.readLine()) != null) {
```

Per escriure, *BufferedWriter* té el mètode *write* que rep un *String*. Si voleu que al fitxer hi hagi un salt de línia darrera la cadena, després heu d'executar el mètode *newLine*;

```
        out.write(linia);
        out.newLine();
```

Si volem escriure les dades al disc directament sense esperar a omplir el buffer podem executar el mètode *flush*.

```
outputStream.flush();
```

BufferedReader i *BufferedWriter* ens proporcionen operacions sobre línies de text. Per treballar amb fitxers de text són més eficients que *FileReader* i *FileWriter*.

Data streams

Els fluxos de dades soporten valors de tipus de dades primitius (boolean, char, byte, short, int, long, float, and double) i String. Tots els fluxos de dades implementen o bé la interfície *DataInput* o bé la *DataOutput*.

Com a exemple veurem els dos més utilitzats, *DataInputStream* i *DataOutputStream*. La primera proporciona mètodes per a llegir dades segons el tipus *readBoolean*, *readChar*, *readInt*, *readUTF*(per String), ... i la segona per escriure dades segons el tipus *writeBoolean*, ...

Per crear aquests streams:

```
DataOutputStream out;
DataInputStream in;

out = new DataOutputStream(
        new  BufferedOutputStream(new FileOutputStream(dataFile)));
in = new DataInputStream(
        new  BufferedInputStream(new FileInputStream(dataFile)));
```

Per crear-los els hem de passar com a argument un buffered stream.

DataOutputStream guarda les dades com una successió de bytes. És a dir, els mètodes writeInt, writeDouble, ... "transformen" aquests tipus primitius a bytes.

DataInputStream llegeix aquests bytes i els transforma a un determinat tipus segons el mètode que hem utilitzat per llegir-los. Per exemple *readBoolean* llegira un sol byte mentre que *readInt* en llegirà 4.

Per tant, **per llegir correctament un fitxer hem de saber en quin ordre s'han escrit les dades**.

L'única manera que tenim de saber quan acaba un fitxer és recollir l'excepció *EOFException* (End Of File Exception)

Per exemple, si volem escriure una llista de productes a un fitxer, amb el nom, el preu i les unitats que en tenim, ho faríem de la següent manera:

```
for (Producte producte : productes) {
    out.writeDouble(producte.getPreu());
    out.writeInt(profucte.getUnitats());
    out.writeUTF(producte.getNom());
}
```

Per llegir aquest fitxer hem d'anar alerta en llegir les dades en el mateix ordre. Si no ho feim bytes que formaven part del double a lo millor ara formen part d'un int, ... i no recuperarem les dades que hem guardat, en el millor dels casos tendrem altres valors i en el pitjor petarà.

```
try {
        ArrayList<Producte> productes=new ArrayList<>();
        while (true) {
                double preu = in.readDouble();
                int unitats = in.readInt();
                String nom = in.readUTF();
                productes.add(new Producte(nom, unitats, preu));
        }
} catch (EOFException e) {
}
```

Object streams

Els *object streams* soporten objectes, és a dir en lloc de guardar a disc un *double* un *int* i un *String*, podem guardar directament un Producte.

Per poder guardar un objecte a un stream, la seva classe ha d'implementar la interfície Serializable. Es tracta d'una interfície marcadora, és a dir, sense cap mètode, que només serveix per, utilitzant polimorfisme, facilitar la tasca de declarar els mètodes dels streams.

Utilitzarem els streams ***ObjectInputStream*** i ***ObjectOutputStream*** per a llegir i escriure objectes al flux.

```
ObjectOutputStream out;
ObjectInputStream in;

out = new ObjectOutputStream(
        new BufferedOutputStream(new FileOutputStream(dataFile)));

in = new ObjectInputStream(
        new BufferedInputStream(new FileInputStream(dataFile)));
```

Per crear-los necessitam passar al constructor un stream amb buffer com a argument.

Per enviar un objecte al flux:

```
out.writeObject(producte);
```

Per recuperar un objecte del flux:

```
Producte recuperat=(Producte) in.readObject();
```

Sempre torna *Object*, per tant hem de fer el càsting a la classe original de l'objecte.

Si l'objecte recuperat no és del tipus esperat es generarà una excepció del tipus *ClassNotFoundException*.

Aquest *stream* no dona cap senyal d'haver arribat al final de fitxer, *EOF*. El més habitual és posar el *while* que llegeix a *true* i capturar *EOFException*. Una altra possibilitat és guardar qualque tipus de metadata, un objecte inicial que indiqui quants n'hi ha o situar un objecte al final que poguem reconèixer com a final de fitxer.

```
try {
        ArrayList<Producte> productes=new ArrayList<>();
        while (true) {
                Producte recuperat=(Producte) in.readObject();
                productes.add(recuperat);
        }
} catch (EOFException e) {
}
```

Escriptura i lectura d'objectes complexos

Quan tots els atributs d'un objecte són tipus primitius és realment simple escriure aquest objecte a un *stream*. Però moltes vegades ens trobam que un objecte té un atribut que és un altre objecte. I aquest, en pot contenir un altre, i... Per exemple, un alumne pot incloure una llista amb les seves qualificacions.

En aquest cas, en recuperar l'objecte amb *readObject*, també haurem de recuperar tots aquests altres objectes que conté. En recuperar l'alumne, també haurà de recuperar la llista amb totes les seves qualificacions.

Això vol dir que el mètode *writeObject* ha de ser capaç de recórrer tota aquesta estructura i guardar-la de manera que el *readObject* la pugui reconstruir.

Si dos objectes fan referència a un altre, per exemple dos alumnes fan referència al mateix mòdul, en reconstruir-los es seguiran mantenint aquestes referències, no es crearan dos mòduls diferents.

Si escrivim dues vegades el mateix objecte al mateix *stream* estam guardant l'objecte una vegada. El que repetim són les referències, de forma que en tornar a recuperar les dades de *l'stream*, seguirem tenint un sol objecte amb dues referències.

En canvi, si escrivim el mateix objecte a dos *streams* diferents, i recuperam l'objecte dels dos streams, tendrem dos objectes diferents.

Per exemple per guardar al fitxer una llista de productes ho podem fer de la següent manera:

```
ObjectOutputStream out;
out = new ObjectOutputStream(
        new BufferedOutputStream(new FileOutputStream(dataFile)));
ArrayList<Producte> productes=new ArrayList<>();
```

```
... //Omplim la llista
out.writeObject(productes);
```

D'aquesta manera guardam tota la llista al fitxer. Per recuperar-la:

```
ObjectInputStream in;
in = new ObjectInputStream(
        new BufferedInputStream(new FileInputStream(dataFile)));

ArrayList<Producte> recuperats=(ArrayList<Producte>) in.readObject();
```

d'aquesta manera tenim dins recuperats la llista de productes que havíem guardat abans.