Índex

Introducció

En aquesta unitat, anomenada "Persistència en fitxers", s'estudia en detall la gestió de fitxers i del seu contingut. La unitat s'estructura en tres grans apartats: la gestió dels sistemes de fitxers, la gestió del contingut dels fitxers i l'estudi de diferents tècniques que permetin la persistència basada en fitxers dels objectes d'una aplicació.

A la primera part es dóna una visió general del que cal entendre com a sistema de fitxers. Es fa palès que cada sistema operatiu pot organitzar els seus fitxers de diferent manera i fer servir diferents sistemes per identificar la seva ubicació. La conseqüència d'això és que no existeix una forma estàndard de referenciar un fitxer, sinó que aquesta depèn de cada sistema operatiu.

Per tal d'apaivagar aquesta dependència Java ha desenvolupat la classe `File`, la qual permet abstraure la referència d'un fitxer amb independència de la notació pròpia que el sistema operatiu faci servir.

Les instàncies de tipus `File` poden representar tant un fitxer com un directori (o carpeta). En qualsevol cas, ens permetran interrogar-les per obtenir informació de l'element representat. Així, per exemple, podem saber si es tracta d'un fitxer o d'un directori, o bé obtenir informació del seu nom en qualsevol de les seves formes (absoluta o relativa), la seva mida, la data de creació, etc.

`File` no disposa de cap utilitat per obtenir una llista ordenada. L'ordenació s'aconseguirà fent servir les utilitats de Java a la classe System, per ordenar col·leccions usant un Comparator.

En l'apartat *Gestió de contingut dels fitxers* s'ofereixen diferents aproximacions al concepte de fitxers. Des d'un punt de vista de la informació, els fitxers són magatzems de dades estructurades, però des del punt de vista de l'aplicació, els fitxers es poden considerar recursos d'intercanvi de dades entre dos sistemes, l'un volàtil (la memòria RAM) i l'altre permanent (els dispositius d'emmagatzematge).

Des del primer punt de vista es fa palès que l'estructura de dades contenint la informació haurà d'adaptar-se a les limitacions dels sistemes d'emmagatzematge, produint-se sovint un desfasament entre l'estructura planificada per suportar la informació i l'estructura adaptada que acabarà utilitzant-se per emmagatzemar-la. Així, distingim entre l'estructura lògica, propera a la representació mental planificada, i l'estructura física o forma com es guarda realment la informació en el dispositiu.

Des del segon punt de vista apareix el concepte de flux de dades, entès com a transferència d'informació des dels dispositius persistents a la memòria de treball de l'aplicació o a l'inrevés.

La direcció de la transferència ens permet classificar els fluxos en fluxos d'entrada (del magatzem a l'aplicació) i fluxos de sortida (de l'aplicació al magatzem).

El concepte de flux és una abstracció que permet tractar totes les dades com a seqüències de Bytes. Ara bé, la complexitat de la representació seriada dels caràcters ens obliga a considerar de forma especial els fluxos destinats a treballar amb caràcters.

Un cop estudiats el fluxos, s'ofereix una àmplia visió del nou paquet d'entrada i sortida de Java anomenat nio.S'estudia en detall la classe `FileChannel` i la classe `Buffer`, veient com es poden fer treballar conjuntament per obtenir una gran eficiència, sense perdre, però, flexibilitat en el tractament de les dades, siguin del tipus que siguin.

En l'apartat *Persistència d'objectes en fitxers* veurem diverses formes d'emmagatzemar objectes. Bàsicament, estudiarem quatre tècniques. En primer lloc, veurem la persistència basada en la seriació per defecte que JAVA fa de qualsevol objecte. Aquesta tècnica aconsegueix emmagatzemar els objectes en una seqüència de bytes que emmagatzemarem en fitxers fent servir fluxos de bytes.

Es tracta d'una tècnica molt poc costosa per al programador, però que presenta moltes limitacions a l'hora de reutilitzar el format en altres llenguatges diferents a JAVA. També presenta limitacions de versionat quan es fan modificacions a les classes persistents i, per tant, malgrat que útil, no és una opció de persistència gaire estesa.

En segon lloc, estudiarem la creació d'un format binari basat en alguna codificació definida específicament en el codi. Es tracta d'un sistema costós per al programador, ja que ha de codificar el procés de persistència gairebé de forma sencera. Aquest sistema presenta també un desavantatge. Les dades binàries no es representen totes igual en les diferents arquitectures d'ordinadors. És a dir, no hi ha garantia que el format aconseguit es pugui llegir en qualsevol plataforma ni tan sols usant el llenguatge JAVA.

Les altres dues tècniques permeten fer persistència basada en formats XML. L'avantatge de fer servir formats XML és que aquests poden ser llegits per qualsevol llenguatge en qualsevol sistema operatiu i no varien en funció de l'arquitectura de l'ordinador. Presenten, això sí, el problema de les múltiples codificacions dels caràcters, però un cop fixada la codificació que farem servir, el format pot exportar-se a qualsevol plataforma i llegir-se usant qualsevol llenguatge de programació sense cap mena de problema.

La primera tècnica XML usada consistirà en la codificació específica d'utilitats que ens facilitin la transformació dels objectes en dades de text estructurades per mitjà d'etiquetes. Igual que la tècnica anterior, és també un sistema costós per al programador. Tot i això, és molt còmoda de fer servir en persistències que requereixin emmagatzemar poques dades o en persistències especials que requereixin formats complexos i difícils d'automatitzar, perquè el programador té tot el control.

La segona tècnica XML s'anomena *binding*. Consisteix en un conjunt d'utilitats que permeten automatitzar la persistència dels objectes en format XML a partir de la vinculació de les classes de l'aplicació a esquemes XML específics. La vinculació s'aconsegueix configurant "mapes" que ajudin a automatitzar la representació XML que cada classe persistent haurà de tenir. La biblioteca usada en aquesta tècnica és JAXB, un paquet estàndard incorporat al J2SE a partir de la versió 5.0.

Resultats d'aprenentatge

En acabar aquesta unitat, l'alumne:

1. Desenvolupa aplicacions que gestionen informació emmagatzemada en fitxers identificant el camp d'aplicació dels mateixos i utilitzant classes específiques.

- Utilitza classes per a la gestió de fitxers i directoris.
- Valora els avantatges i els inconvenients de les diferents formes d'accés
- Utilitza classes per recuperar informació emmagatzemada en un fitxer XML.
- Utilitza classes per emmagatzemar informació en un fitxer XML.
- Utilitza classes per convertir a un altre format informació continguda en un fitxer XML.
- Preveu i gestiona les excepcions.
- Prova i documenta les aplicacions desenvolupades.

1. Gestió del sistema de fitxers

En els sistemes informàtics actuals, en els quals un sol ordinador pot tenir ben bé més d'un milió de fitxers, resulta imprescindible un sistema que permeti una gestió eficaç de localització, de manera que els usuaris puguem moure'ns còmodament entre tants arxius. La majoria de sistemes de fitxers han incorporat contenidors jerarquitzats que actuen a mode de directoris facilitant la classificació, la identificació i localització dels arxius. Els directoris s'han acabat popularitzant sota la versió gràfica de carpetes.

Hem de tenir en compte, a més, que la necessitat desmesurada d'espai d'emmagatzematge ha dut els SO a treballar amb una gran quantitat de dispositius i a permetre l'accés remot a sistemes de fitxers aliens, distribuïts per la xarxa.

Per poder gestionar tanta varietat de sistemes de fitxers, alguns sistemes operatius com Linux o Unixprenen l'estratègia d'unificar tots els sistemes en un de sòl, per tal d'aconseguir una forma d'accés unificada i amb una única jerarquia que faciliti la referència a qualsevol dels seus components, amb independència del sistema de fitxers en què es trobin realment ubicats. En Linux, sigui quin sigui el dispositiu o el sistema remot real on s'emmagatzemarà l'arxiu, la ruta tindrà sempre la mateixa forma.

```
/cami/on/es/troba/el/Fitxer.txt
```

Per contra, l'estratègia d'altres SO com Windows passa per mantenir ben diferenciats cada un dels sistemes i dispositius on tingui accés. Per distingir el sistema al qual es vol fer referència, Windows usa una denominació específica que incorpora a la ruta de l'element a referenciar. Tot i que Microsoft ha apostat clarament per la convenció UNC, l'evolució d'aquest sistema operatiu, que té com a origen l'MS-DOS, ha fet que coexisteixi amb una altra convenció també molt estesa. Ens referim a la identificació dels dispositius i sistemes amb una lletra de l'alfabet seguida de dos punts. A continuació il·lustrem amb un exemple ambdues convencions. Hem marcat en negreta la denominació específica que identifica el sistema de fitxers o dispositiu específic:

```
F:\cami\on\es\troba\el\Fitxer.txt
\\Servidor7\cami\on\es\troba\el\Fitxer.txt
```

1.1 La classe File. Generalitats

En Java, per gestionar el sistema de fitxers s'utilitza bàsicament 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. Com que es tracta d'una ruta, la classe pot representar tant fitxers com carpetes o directoris.

> Els objectes de la classe `File` representen rutes del Sistema de Fitxers.

Si fem servir una classe per representar rutes, s'aconsegueix una total independència respecte de la notació que cada sistema operatiu utilitza per descriure-les. Recordem que Java és un llenguatge multiplataforma i, per tant, pot donar-se el cas que haguem de fer una aplicació desconeixent el SO on acabarà executant-se.

L'estratègia usada per cada SO no afecta la funcionalitat de la classe `File`, ja que aquesta, en col·laboració amb la màquina virtual, adaptarà les crides al SO amfitrió de forma transparent al programador, és a dir, sense necessitat que el programador hagi d'indicar o configurar res.

Anomenem *instàncies* d'una classe els objectes creats durant l'execució d'una aplicació. Per tant, *objecte* i *instància* es poden usar com a sinònims.

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.

Vegeu en la secció "Annexos" l'annex "Parametrització d'aplicacions. Tècniques de configuració", on s'expliquen diverses tècniques de parametrització d'aplicacions.

En implementacions tant properes al SO, els programadors de Java han de fer un esforç per independitzar les aplicacions implementades de les plataformes on s'executaran. Caldrà, doncs, anar amb cura, fent servir tècniques de parametrització que evitin escriure les rutes directament al codi, de manera que en traslladar les aplicacions de plataforma només calgui modificar les rutes en el sistema de configuració.

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.

Anem ara a veure amb més detall la funcionalitat de la classe `File`, així com la sintaxi dels seus mètodes. Però per tal de fer més amena la lectura, i fugir d'explicacions abstractes o descontextualitzades, centrarem el relat en la implementació d'una aplicació real, a mode d'exemple, que posi en pràctica la utilitat de la classe estudiada.

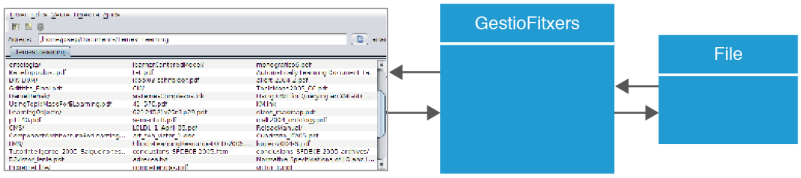
1.2 Funcionalitat de la classe File a partir d'un cas pràctic

L'exemple que ens ajudarà a familiaritzar-nos amb la classe `File` consistirà en implementar un senzill explorador de fitxers semblant al Nautilus de Linux o a l'Explorador de fitxers de Windows.

A l'annex "Biblioteques del mòdul" de la secció "Annexos", trobareu informació de les biblioteques que necessitareu fer servir en aquesta unitat, així com del lloc on podeu obtenir-les.

Cal dir, però, que l'estudi de les interfícies gràfiques necessàries per implementar una aplicació d'aquest estil cau totalment fora dels objectius d'aquest mòdul, per això no implementarem la interfície d'usuari de l'aplicació, sinó que n'usarem una d'existent i ja implementada. Aquesta delega tota la funcionalitat no gràfica a una instància de la interfície `GestioFitxers`.

FIGURA 1.1. Esquema de l'arquitectura de l'exemple que usarem per il·lustrar la funcionalitat de la classe "File"



Només haurem d'implementar `GestioFitxers`codificant una classe que fent ús d'objectes `Files`'adapti als requeriments de la interfície. Com es pot veure a l'arquitectura (figura 1.1), la implementació de `GestorFitxers` pren el paper d'adaptador o intermediari entre la interfície gràfica i les operacions de la classe `File`.

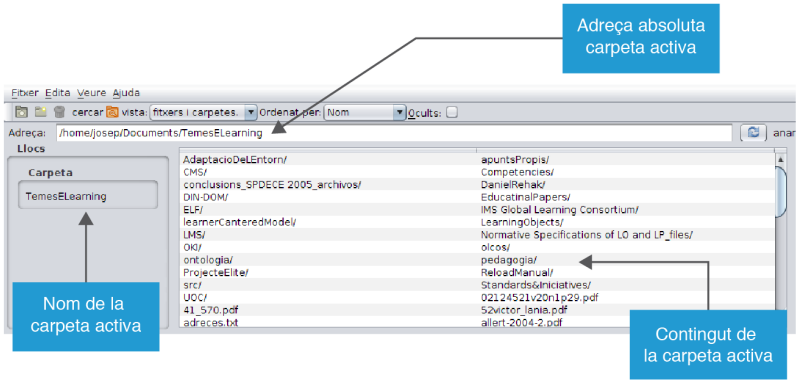
1.2.1 Obtenció d'informació bàsica

La interfície gràfica necessita mostrar la ruta absoluta de la carpeta activa, el nom de la mateixa i el seu contingut (vegeu la figura 1.2). La classe `File`ens ajudarà a recopilar aquesta informació, ja que disposa de tres mètodes per obtenir informació sobre el nom del fitxer o carpeta. El mètode `getName` obté el nom pròpiament dit de l'objecte `File`. És a dir, el nom relatiu que l'element tingui dins la carpeta on es troba contingut.

Els altres dos mètodes (`getAbsolutePathi getCanonicalPath`) permeten ambdós obtenir el nom absolut de l'element. Es tracta d'una cadena amb la ruta sencera des de l'arrel d'acord amb la notació emprada pel sistema operatiu amfitrió. Hi ha, però, sistemes operatius que permeten expressar rutes absolutes de manera redundant. Així, per exemple, en Linux la ruta */home/usuari/../../home/usuari/documents/../documents/arxiu.txt* és una forma redundant de

la ruta */home/usuari/documents/arxiu.txt*, ja que apunten al mateix fitxer. Si una instància de File s'ha creat usant una ruta redundant, aquesta es manté al llarg de tot el cicle de vida de la instància. El mètode `getAbsolutePath` retorna el valor absolut de la ruta però sense eliminar les redundàncies, cosa que pot arribar a complicar la comparació literal de dues rutes redundants. Per simplificar aquestes comparacions, la classe `File` disposa de `getCanonicalPath`, un mètode que retorna la ruta absoluta en la versió més simple.

FIGURA 1.2. Aspecte de la interfície gràfica de l'aplicació en la qual es destaca quina informació del sistema de fitxers es mostra.



Cal destacar que la ruta canònica és una ruta calculada dinàmicament a partir de la ruta original. Per tant, en cas que aquesta tingui errors de notació, no serà possible obtenir el valor canònic, sinó que es produirà una excepció avisant-nos de l'error.

A efectes del nostre exemple, no ens caldrà realitzar cap comparació literal entre rutes absolutes. Escollirem el mètode `getAbsolutePath` en detriment de `getCanonicalPath`, ja que malgrat l'ús d'exempcions és absolutament recomanable per aconseguir aplicacions robustes, la seva presència en el codi dificulta lleugerament la llegibilitat. És per això que hem optat per usar rutes redundants en el nostre gestor.

Quan un objecte de tipus `File` sigui una carpeta, el mètode `list` ens retorna un objecte de tipus Array amb el nom de tots els fitxers i carpetes que contingui. Si precisem informació addicional de cada element contingut, disposem del mètode `listFiles`, que en comptes de retornar els noms dels elements continguts ens retorna in *Array* d'objectes `File` vinculats a cada un dels elements.

Ja podem començar a codificar la classe que implementi `GestorFitxers`, que a partir d'ara anomenarem `GestorFitxersImpl`. Serà necessari mantenir una instància de `File`vinculada a la carpeta activa de la que obtinguem les dades a mostrar en la interfície gràfica i sobre la que recaurà qualsevol operació de fitxers que demandem.

D'altra banda, hem de tenir en compte que les interfícies gràfiques necessiten refrescar les dades força sovint. Per tal de no estar calculant el contingut de les carpetes cada vegada que refresquem, prendrem la decisió de mantenir emmagatzemat el contingut en una matriu d'objectes que caldrà anar sincronitzant només quan es produeixin canvis.

Començarem, doncs, afegint a la classe `GestioFitxersImpl` dos atributs que anomenarem `continguti` `carpetaDeTreball`.

```
1 public class GestioFitxersImpl implements GestioFitxers{
2     private Object[][] contingut;
3     private File carpetaDeTreball=null;
4     ...
```

El constructor inicialitzarà l'atribut `carpetaDeTreball` amb l'arrel del primer dels sistemes de fitxers disponibles i actualitzarà el valor de `contingut`. Per a la inicialització de la carpeta de treball, usarem un mètode *static* de `File` anomenat `listRoots` que retorna un vector d'objectes de tipus `File`, cada un d'ells vinculat a l'arrel d'un dels sistemes de fitxers disponible des del SO amfitrió.

En sistemes operatius com Unix o Linux, `File.listRoots()` retornarà un únic element, l'arrel de sistema de fitxers (/).

Per gestionar l'actualització del contingut crearem un mètode que anomenarem *actualitza*. És preferible modularitzar l'actualització del contingut, ja que a més del constructor, altres operacions provocaran també canvis en el contingut i requeriran d'actualització.

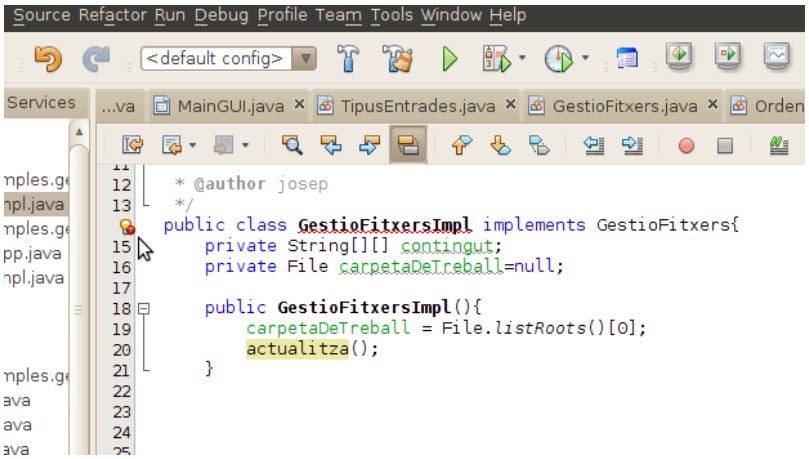
```
1 public GestioFitxersImpl(){
2     carpetaDeTreball = File.listRoots()[0];
3     actualitza();
4 }
```

A continuació, crearem l'esquelet de la classe d'acord amb la interfície `GestioFitxers`que implementa, per tal de poder anar provant els mètodes a mida que els anem codificant sense necessitat d'esperar a tenir-los tots implementats. La creació de l'esquelet consistirà en definir tots el mètodes de la interfície amb una única instrucció que llanci una excepció de tipus `UnsupportedOperationException`. Cada mètode hauria de presentar un aspecte semblant a:

```
1 @Override
2 public void amunt() {
3     throw new UnsupportedOperationException("Not supported yet.");
4 }
```

Podem crear manualment l'esquelet, o bé aprofitar la utilitat de propostes de l'IDE que s'activa quan detecta algun error ben definit (vegeu figura 1.3). Immediatament després d'afegir ***implements*** `GestioFitxers` a la declaració de la classe, observarem una bombeta al marge esquerre de l'editor. En clicar damunt la icona, apareixerà la proposta *implements all abstract methods*. Si acceptem la proposta, NetBeans ens crearà l'esquelet de la classe.

FIGURA 1.3. Figura que il·lustra l'aspecte de l'IDE en detectar un error ben definit



L'IDE mostra la bombeta del marge esquerre i en clicar-la, ens mostrarà una o més propostes que solucionarien el problema detectat

La bombeta que surt al marge dret d'una línia de codi (vegeu la figura 1.3) indica que NetBeans ens vol fer alguna proposta per eliminar algun error.

Si volem realitzar una primera prova, necessitarem almenys la ruta absoluta de la carpeta de treball, el nom curt de la mateixa i el seu contingut en forma de col·lecció de cadena de caràcters amb el nom dels seus elements.

Implementarem el mètode `getAdrecaCarpeta` que llegirà la ruta absoluta de la carpeta de treball i el mètode `getNomCarpeta()` que obtindrà el nom curt.

```
1  public String getAdrecaCarpeta(){
2      return carpetaDeTreball.getAbsolutePath();
3  }
4
5  public String getNomCarpeta(){
6      return carpetaDeTreball.getName();
7  }
```

Accessors

En Java, s'anomenen *accessors* d'atributs privats aquells mètodes que permeten llegir i escriure el contingut de l'atribut. Les convencions Java recomanen que l'accessor de lectura s'anomeni com l'atribut,anteposant-hi però el prefix get i canviant la primera inicial del nom de l'atribut a majúscula. L'accessor d'escriptura seguiria el mateix patró, però caldrà anteposar-hi el prefix set. Exemple: `getColumnes` i `setColumnes` seran els noms dels accessors de l'atribut `columnes`.

Com ja hem comentat, el contingut l'omplirem des del mètode `actualitza`. De moment, només omplirem la matriu `contingut`amb els noms dels elements de la carpeta, però més endavant caldrà modificar el mètode per aconseguir més informació. Per tal que els noms es distribueixin uniformement per l'espai que la interfície gràfica hi destina, els organitzarem en columnes. Per defecte, hi haurà tres columnes que podrem canviar a discreció. La interfície gràfica necessitarà també saber en quantes files i columnes s'ha aconseguit encabir el contingut. Afegirem, doncs, dos atributs, `columnesifiles`. El primer de lectura i escriptura amb els seus accessors corresponents. El segon només de lectura, perquè el nombre de files dependrà de la quantitat de fitxers existent a la carpeta. Per tant, no implementarem l'accessor d'escriptura.

```
1  ...
2  private int files=0;
3  private int columnes=3;
4
5  ...
6
7  @Override
8  public int getColumnes() {
9      return columnes;
10 }
11
12 @Override
13 public void setColumnes(int columnes) {
```

```
14      this.columnes = columnes;
15  }
16
17  @Override
18  public int getFiles() {
19      return files;
20  }
21  ...
22
23  private void actualitza(){
24      String[] fitxers = carpetaDeTreball.list(); //obtenir els noms
25      //calcular el nombre de files necessari
26      files = fitxers.length / columnes;
27      if(files*columnes < fitxers.length){
28          files++; //si hi ha residu necessitem un fila més
29      }
30
31      //dimensionar la matriu contingut d'acord als resultats
32
33      contingut = new String[files][columnes];
34      //Omplir el contingut amb els noms obtinguts
35      for(int i=0; i<columnes; i++){
36          for(int j=0; j<files; j++){
37              int ind = j*columnes+i;
38              if(ind<fitxers.length){
39                  contingut[j][i]=fitxers[ind];
40              }else{
41                  contingut[j][i]="";
42              }
43          }
44      }
45  }
46
47  @Override
48  public Object[][] getContingut() {
49      return contingut;
50  }
```

Si voleu, podeu fer ja una primera prova per veure el resultat del que acabeu d'implementar. Seguiu les indicacions de l'annex "Creació del projecte Gestió de Fitxers i importació de les biblioteques", que trobareu en la secció "Annexos".

Modificarem també l'accessor de lectura a l'atribut `contingut`, que s'anomena `getContingut` per tal que retorni el valor del mateix, adequadament actualitzat.

Un cop creats els principals mètodes per obtenir informació, començarem a implementar mètodes que ens permetin navegar pel sistema. Amb el mètode `setAdrecaCarpeta` podrem canviar la carpeta de treball a partir de la ruta passada com a paràmetre.

```
1   @Override
2   public void setAdrecaCarpeta(String adreca)
3                       throws GestioFitxersException{
4       File file = new File(adreca);
5       //es controla que l'adreça passada existeixi i sigui un directori
6       if(!file.isDirectory()){
7           throw new GestioFitxersException("Error. S'esperava "
8               + "un directori, però "
9               + file.getAbsolutePath() + " no és un directori.");
10      }
11      //es controla que es tinguin permisos per llegir la carpeta
12      if(!file.canRead()){
13          throw new GestioFitxersException("Alerta. No podeu accedir a "
14              + file.getAbsolutePath() + ". No teniu prou permisos");
15      }
16      //nova assignació de la carpeta de treball
17      carpetaDeTreball=file;
18      //es requereix actualitzar el contingut
19      actualitza();
20  }
```

Usarem el mètode de la classe `File` anomenat `isDirectory` per controlar que la ruta es correspongui a una carpeta existent, ja que si no existeix el sistema no sabrà si es tracta d'una carpeta o un fitxer i `isDirectory` retornarà fals. Si no és una carpeta, caldrà llançar una excepció informant del problema. També es llançarà una excepció en cas que no hi hagi permisos per poder llegir el contingut de la carpeta. Per comprovar-ho, usarem `canRead`.

La interfície ens indica que cal implementar el mètode `entraA` de forma que serveixi per canviar la *carpeta de treball* a la carpeta passada per paràmetre i expressada com una ruta relativa de la carpeta activa en el moment de l'execució del mètode. En la nostra aplicació servirà per entrar un nivell dins l'arbre de directoris. És a dir, és l'operació que usa la interfície gràfica per mostrar el contingut d'aquella carpeta sobre la qual es faci doble clic. Com en el cas de `setAdrecaCarpeta`, caldrà crear un nou objecte File, però aquest cop relatiu a la carpeta activa. Usarem el constructor, al qual li passarem dos paràmetres: el `File` de la *carpeta de treball,* que farà de ruta base i una cadena amb la ruta relativa a la base.

```
@Override
public void entraA(String nomCarpeta) throws GestioFitxersException{
    File file = new File(carpetaDeTreball, nomCarpeta);
    //es controla que el nom correspongui a una carpeta existent
    if(!file.isDirectory()){
        throw new GestioFitxersException("Error. S'ha trobat "
          + file.getAbsolutePath() + " però s'eperava un directori");
    }
    //es controla que es tinguin permisos per llegir la carpeta
    if(!file.canRead()){
        throw new GestioFitxersException("Alerta. No podeu accedir a "
          + file.getAbsolutePath() + ". No teniu prou permisos");
    }
    //nova assignació de la carpeta de treball
    carpetaDeTreball=file;
    //es requereix actualitzar el contingut
    actualitza();
}
```

Aquí caldrà també comprovar que existeixi el directori i la nova carpeta activa sigui accessible (es pugui llegir el seu contingut).

Per poder pujar de nivell, la `GestioFitxers` preveu el mètode `amunt`. Ho implementarem aprofitant la utilitat de la classe `File` que ens retorna la instància `File` pare. Ens referim a `getParentFile`.

```
@Override
public void amunt(){
    if(carpetaDeTreball.getParentFile()!=null){
        carpetaDeTreball = carpetaDeTreball.getParentFile();
        actualitza();
    }
}
```

L'execució de `getParentFile` sobre l'arrel del sistema de fitxers ens retorna un valor *null*. Usarem aquesta característica per assegurar de no sobrepassar mai l'arrel i evitar errors amb punters nuls.

1.2.2 Obtenció d'informació detallada

La classe `File` disposa encara de molts mètodes per obtenir informació detallada de cada element. Generalment, les aplicacions com la que estem fent permeten mostrar una finestra amb la informació detallada d'un dels components. La interfície gràfica està preparada per obrir una finestra amb la informació que la instància de `GestioFitxers` li passi en cridar al mètode `getInformacio`. El mètode `getInformacio` rep una cadena amb el nom del fitxer del qual es demana informació. Amb el nom haurem d'instanciar un objecte `File`i organitzar la seva informació dins una cadena de caràcters. Per tal d'anar afegint informació de mica en mica, usarem `StringBuilder`, que optimitza la concatenació successiva de cadenes, millor que la classe `String`.

Des de la interfície gràfica obtindrem la informació addicional seleccionant l'opció *Propietats* des del menú *Fitxer* i un cop s'obri el quadre de diàleg, seleccionant la la pestanya *Informació*.

La informació que extraurem de `File` serà, el nom, usant `getName`; si es tracta d'una carpeta o d'un fitxer; podem fer servir `isDirectory` o be `isFile`indistintament per saber-ho. Recollirem també la ruta absoluta canònica, extreta de `getCannonicalPath` i la data de la darrera modificació o de la creació si no s'ha modificat mai, usant el mètode `lastModified`.

En cas que es tracti d'una carpeta, s'indicarà també el nombre d'entrades que conté interrogant la longitud del vector, carregat amb el contingut i retornat per *list*. Fent servir els mètodes `getFreeSpace`, `getUsableSpaceo` `getTotalSpaceobtindrem` informació de l'espai lliure, l'espai disponible i l'espai total de la partició on es trobi la carpeta analitzada. L'espai disponible fa referència a l'espai que l'aplicació pot usar en el moment de l'execució i que no ha de coincidir pas amb l'espai lliure.

Si en comptes de tractar-se d'una carpeta fos un fitxer, caldrà mostrar només la seva mida en bytes usant el mètode `length`.

A la biblioteca que se us proporciona trobareu la classe `ByteFormat`, que automatitza la formatació de la mida dels fitxers escollint les unitats pertinents. Vegeu la documentació a l'annex "Documentació API *GestioFitxersBase*" de la secció "Annexos".

En ambdós casos indicarem també si el sistema l'ha marcat com ocult o resta visible a l'usuari. La informació l'extraurem del mètode `isHidden` de `File`. Vegem la implementació:

```
1  @Override
2  public String getInformacio(String nom)
3                  throws GestioFitxersException {
4      ByteFormat byteFormat= new ByteFormat("#,###.0", ByteFormat.BYTE);
5      StringBuilder strBuilder = new StringBuilder();
6      File file = new File(carpetaDeTreball, nom);
7      //Es controla que existeixi l'element a analitzar
8      if(!file.exists()){
9          throw new GestioFitxersException("Error. No es pot"
10            + " obtenir informació " + "de " + nom + ", no existeix.");
11     }
12     //es controla que es tinguin permisos per llegir la carpeta
13     if(!file.canRead()){
14         throw new GestioFitxersException("Alerta. No es pot "
15             + "accedir a " + nom + ". No teniu prou permisos");
16     }
17     //S'escriu el títol
18     strBuilder.append("INFORMACIÓ DEL SISTEMA");
19     strBuilder.append("\n\n");
20     //S'afegeix el nom
21     strBuilder.append("Nom: ");
```

```
22      strBuilder.append(nom);
23      strBuilder.append("\n");
24      //El tipus (carpeta o fitxer)
25      strBuilder.append("Tipus: ");
26      if(file.isFile()){
27          //es fitxer
28          strBuilder.append("fitxer");
29          strBuilder.append("\n");
30          //s'escriu La mida
31          strBuilder.append("Mida: ");
32          strBuilder.append(byteFormat.format(file.length()));
33          strBuilder.append("\n");
34      }else{
35          //es carpeta
36          strBuilder.append("carpeta");
37          strBuilder.append("\n");
38          //S'indica el nombre d'elements continguts
39          strBuilder.append("Contingut: ");
40          strBuilder.append(file.list().length);
41          strBuilder.append(" entrades\n");
42      }
43      //Afegim la ubicació
44      strBuilder.append("Ubicació: ");
45      /* Cal posar el try per exigències del llenguatge, però no
46       * controlarem aquest error doncs sabem que mai es produirà.
47       * Si hem arribat fins aquí és que l'adreça és bona */
48      try {
49          strBuilder.append(file.getCanonicalPath());
50      } catch (IOException ex) {/*Mai es produirà aquest error*/}
51      strBuilder.append("\n");
52      //Afegim la data de la última modificació
53      strBuilder.append("Última modificació: ");
54      Date date = new Date(file.lastModified());
55      strBuilder.append(date.toString());
56      strBuilder.append("\n");
57      //Indiquem si és o no un fitxer ocult
58      strBuilder.append("Ocult: ");
59      strBuilder.append((file.isHidden())?"Si":"No");
60      strBuilder.append("\n");
61
62      if(file.isDirectory()){
63          //Mostrem l'espai lliure
64          strBuilder.append("Espai lliure: ");
65          strBuilder.append(byteFormat.format(file.getFreeSpace()));
66          strBuilder.append("\n");
67          //Mostrem l'espai disponible
68          strBuilder.append("Espai disponible: ");
69          strBuilder.append(byteFormat.format(file.getUsableSpace()));
70          strBuilder.append("\n");
71          //Mostrem l'espai total
72          strBuilder.append("Espai total: ");
73          strBuilder.append(byteFormat.format(file.getTotalSpace()));
74          strBuilder.append("\n");
75      }
76
77      return strBuilder.toString();
78  }
```

Destacarem que en aquest cas, en no saber si es tracta d'un fitxer o una carpeta, la comprovació de l'existència la realitzarem mitjançant l'operació de `File` anomenada `exists`. Un altre fet remarcable és que malgrat sapiguem que `getCanonicalPath`no llançarà mai cap excepció, ja que el nom és correcte, ja que abans s'han obtingut altres dades, la rigidesa de les excepcions ens obliga a embolcallar la crida dins una sentència *try-catch*.

1.2.3 Filtratge, ordenació i classificació de fitxers

Arribats a aquest punt i abans de començar a implementar els mètodes de manipulació del sistema (creació o eliminació d'elements), cal adonar-nos que la visualització del contingut a la interfície gràfica no és gaire bona, ja que no es distingeixen directoris de fitxers, no es visualitzen en cap ordre determinat, els fitxers ocults apareixen sempre i no es mostra informació addicional dels elements.

Començarem per amagar els fitxers ocults. La classe `File`, en el moment de recopilar la informació del contingut d'un directori a través de les operacions `list` o `listFile`, pot acceptar l'ajuda d'un filtre que indiqui quins fitxers cal descartar.

Els filtres han d'implementar la interfície `FilenameFilter`, i poden usar-se indistintament en el mètode `list` i en el mètode `listFile`. Existeix també la interfície `FileFilter` amb un objectiu similar, però malauradament només és acceptada per `listFile`.

Els filtres de tipus `FilenameFilter` implementaran un mètode anomenat `accept` que rebrà dos paràmetres: un `File` indicant el directori on es troba ubicat l'element a avaluar i una cadena amb el nom d'aquest element. El mètode retornarà cert o fals en funció de si es vol descartar o incloure a la llista del contingut.

Davant la multiplicitat de filtres que es poden necessitar, les instàncies solen implementar-se com a classes anònimes o com a classes internes per tal de reduir la quantitat de classes del model, minimitzar l'acoblament i permetre que les instàncies creades tinguin accés total als atributs i mètodes de la classe amfitriona si fos necessari.

En aquesta aplicació optarem per crear una classe interna de tipus `FilenameFilter`, principalment per reduir l'acoblament i mantenir-la aïllada del model. Escollim `FilenameFilter`, ja que necessitarem aplicar el criteri de selecció tant sobre `listFile` com sobre `list`. El mètode accept generarà un `File` amb els paràmetres rebuts i retornarà cert si no és ocult o fals en cas contrari.

```
1 private class FiltreFitxersOcults implements FilenameFilter{
2     @Override
3     public boolean accept(File pfile, String string) {
4         File file = new File(pfile, string);
5         return !file.isHidden();
6     }
7 }
```

El filtre es farà efectiu quan el passem per paràmetre a List:

```
1 String[] fitxers = carpetaDeTreball.list(new FiltreFitxersOcults());
```

En el codi anterior, la variable fitxers contindrà tots els fitxers de la carpeta de treball menys els ocults.

Passem ara a l'ordenació. Usarem la utilitat anomenada `sort`de la classe `Arrays`. Aquest mètode permet ordenar un vector de qualsevol tipus usant una instància de *Comparator*. Aquesta interfície requereix d'un únic mètode amb el nom de *compare*, el qual, rebent dos objectes, podrà analitzar-los i dictaminar quina relació d'ordre s'estableix entre ells. El veredicte de l'anàlisi es retornarà fent servir la següent convenció: si el primer és menor que el segon es retornarà un valor enter negatiu; per contra, si és més gran, es retornarà un valor enter positiu. Es retornarà zero només si són iguals.

La classe `String` disposa ja d'una instància de `Comparator`. Es tracta de l'atribut estàtic anomenat `String.CASE_INSENSITIVE_ORDER`, que compara cadenes en ordre alfanumèric sense tenir en compte les majúscules o minúscules.

Aprofitarem els valors del tipus enumerat `TipusOrdre`, que podeu trobar a la biblioteca, per determinar l'ordenació del contingut. La categoria *DESORDENAT* indicarà que no desitgem cap mena d'ordenació (com fins ara). La categoria *NOM*, en canvi, indicarà que volem el contingut ordenat d'acord amb el nom de cada element. L'enumeració *TipusOrdre* disposa també de les categories *MIDA* i *DATA_MODIFICACIO*, que fareu servir en els exercicis.

De moment crearem dos atributs amb els accessors corresponents, per indicar al gestor si cal o no filtrar els fitxers ocults i quin tipus d'ordre s'espera. Afegirem també una instància privada del filtre `FiltreFitxersOcults` per no haver d'anar creant un objecte nou cada cop que actualitzem el contingut.

```
...
private TipusOrdre ordenat;
private boolean mostrarOcults;
private final FiltreFitxersOcults filtreFitxersOcults=
    new FiltreFitxersOcults();
...
@Override
public boolean getMostrarOcults() {
    return mostrarOcults;
}

@Override
public void setMostrarOcults(boolean ocults) {
    this.mostrarOcults=ocults;
    actualitza();
}

@Override
public TipusOrdre getOrdenat() {
    return ordenat;
}

@Override
public void setOrdenat(TipusOrdre ordenat) {
    this.ordenat=ordenat;
    actualitza();
}
```

En el mètode `actualitza` usarem els atributs creats per saber si cal ordenar i si cal amagar els fitxers ocults. La modificació del mètode donarà com a resultat:

```
private void actualitza(){
    String[] fitxers ; //obtenir els noms
    if(mostrarOcults){
```

```
         fitxers = carpetaDeTreball.list();
    }else{
        fitxers = carpetaDeTreball.list(filtreFitxersOcults);
    }
    columnes = columnesBase;
    //calcular el nombre de files necessari
    files = fitxers.length / columnes;
    if(files*columnes < fitxers.length){
        files++; //si hi ha residu necessitem un fila més
    }
    //ordenació del contingut
    if(ordenat==TipusOrdre.NOM){
        Arrays.sort(fitxers, String.CASE_INSENSITIVE_ORDER);
    }
    //dimensionar la matriu contingut d'acord als resultats
    contingut = new String[files][columnes];
    /* Omplir el contingut amb els noms dels elements de la
     * carpeta activa */
    for(int i=0; i<columnes; i++){
        for(int j=0; j<files; j++){
            int ind = j*columnes+i;
            if(ind<fitxers.length){
                contingut[j][i]=fitxers[ind];
            }else{
                contingut[j][i]="";
            }
        }
    }
}
```

1.2.4 Modificació del sistema de fitxers

Ara veurem com crear noves carpetes i fitxers, com eliminar-los, com canviar-los el nom i com canviar-los la data de modificació. La creació de noves carpetes es realitza executant `mkdir` o bé `mkdirs`. El primer, crearà el directori vinculat a la instància *File* des de la qual s'executa, però ho farà ubicat a la carpeta pare. És a dir, aquest mètode només té la capacitat de crear un sol nivell, mentre que `mkdirs`crearà totes les carpetes necessàries per tal que la ruta representada per la instància existeixi en finalitzar l'execució.

`mkdir`precisa que la ruta fins el nivell immediatament superior a la instància existeixi. En cas contrari no es crearà la ruta. Aquest mètode retorna cert si aconsegueix la creació i fals en cas contrari. De la mateixa manera, `mkdirs`retorna també cert en cas que la creació tingui èxit o fals si no en té.

`GestiFitxersImpl` no necessita `mkdirs`, perquè totes les creacions es faran des de la carpeta de treball, que serà la carpeta pare de nova creació. Caldrà, això sí, controlar els possibles errors que es puguin produir per manca de permisos d'escriptura a la carpeta de treball o deguts a l'existència prèvia d'una carpeta amb el mateix nom.

```
@Override
public void creaCarpeta(String nomCarpeta)
                  throws GestioFitxersException{
    File file = new File(carpetaDeTreball, nomCarpeta);
    if(!carpetaDeTreball.canWrite()){
        throw new GestioFitxersException("Error. No s'ha pogut crear "
```

```
            + nomCarpeta + ". No teniu suficients permisos");
    }
    if(file.exists()){
        throw new GestioFitxersException("Error. No s'ha pogut crear."
            + " Ja existeix un fitxer o carpeta amb el nom "
            + nomCarpeta);
    }
    if(!file.mkdir()){
        throw new GestioFitxersException("Error. No s'ha pogut crear "
            + nomCarpeta + ".");
    }
    actualitza();
}
```

La creació de fitxers serà similar, però usant el mètode `createNewFile`. Aquesta operació només crearà el fitxer en cas que no n'existeixi cap altre amb el mateix nom. A més, per aconseguir la creació caldrà que tota la ruta de la carpeta on s'hagi de fer la creació existeixi, sigui vàlida i tingui permisos d'escriptura.

> És important adonar-se que les instàncies de `File` d'elements no creats no es poden reconèixer encara ni com a fitxers ni com a carpetes, sinó com a **elements inexistents**.

Aquesta característica té un implicació important. No es pot usar un sola instància per crear la ruta i el fitxer vinculat a un `File` si cap dels dos existeix, ja que l'execució de `mkdirs` interpretaria que el darrer nom de la ruta és també una carpeta i la crearia com a tal. En intentar executar `createNewFile` no es produiria cap efecte perquè el nom ja existiria en forma de carpeta. Es necessiten com a mínim dues instàncies per realitzar la doble creació: una vinculada a la carpeta contenidora i l'altra al fitxer en qüestió.

La implementació a la classe `GestióFitxersImpl` tindrà una forma semblant a:

```
@Override
public void creaFitxer(String nomFitxer)
              throws GestioFitxersException {
    File file = new File(carpetaDeTreball, nomFitxer);
    if(!carpetaDeTreball.canWrite()){
        throw new GestioFitxersException("Error. No s'ha pogut crear "
            + nomFitxer + ". No teniu suficients permisos");
    }
    if(file.exists()){
        throw new GestioFitxersException("Error. No s'ha pogut crear."
            + " Ja existeix un fitxer o carpeta amb el nom "
            + nomFitxer);
    }
    try {
        if(!file.createNewFile()){
            throw new GestioFitxersException("Error. No s'ha pogut "
                + "crear " + nomFitxer + ".");
        }
    } catch (IOException ex) {
         throw new GestioFitxersException("S'ha produit un error "
            + "d'entrada o sortida: '" + ex.getMessage() + "'", ex);
    }
    actualitza();
}
```

És molt similar a l'operació de creació de carpetes. Aquí destacarem només la possibilitat que el sistema operatiu reporti una excepció d'entrada/sortida (*IOException*) si es troben problemes en escriure al dispositiu. Per evitar el llançament de múltiples tipus d'excepcions, encapsularem els errors que es produeixin dins una excepció de tipus `GestioFitxersException`.

Per a l'eliminació de carpetes o fitxers, usarem el mètode `delete`, que permet eliminar indistintament fitxers o carpetes. Igual que la resta de mètodes de modificació, retorna cert si l'eliminació s'ha pogut dur a terme i fals en cas contrari. Com que l'èxit o el fracàs de l'operació depèn en gran mesura del sistema operatiu, no disposa de controls d'error específics, sinó que caldrà implementar-los a `GestioFitxersImpl`.

```
@Override
public void elimina(String nom) throws GestioFitxersException{
    File file = new File(carpetaDeTreball, nom);
    if(!carpetaDeTreball.canWrite()){
        throw new GestioFitxersException("Error. No s'ha pogut "
            + "eliminar " + nom
            + ". No teniu suficients permisos");
    }
    if(!file.exists()){
        throw new GestioFitxersException("Error. S'intenta "
            + "eliminar " + nom + " però no existeix.");
    }
    if(!file.delete()){
        if(file.isDirectory() && file.list().length>0){
            throw new GestioFitxersException("Error. No s'ha pogut "
                + "eliminar. La carpeta " + nom + "no està buida.");
        }else{
            throw new GestioFitxersException("Error. No s'ha pogut "
                + "eliminar " + nom + ".");
        }
    }
    actualitza();
}
```

La utilitat que reanomena fitxers o carpetes presenta unes característiques semblants a les que ja s'han vist. És a dir, retorna cert si aconsegueix l'objectiu i fals en cas contrari. La diferència principal és que el mètode `rename` rep per paràmetre el nou nom amb el qual es desitja reanomenar la instància `File`.

```
@Override
public void reanomena(String nom, String nomNou)
                      throws GestioFitxersException{
    File file = new File(carpetaDeTreball, nom);
    File fileNou = new File(carpetaDeTreball, nomNou);
    if(!carpetaDeTreball.canWrite()){
        throw new GestioFitxersException("Error. No s'ha pogut "
            + "eliminar " + nom
            + ". No teniu suficients permisos");
    }
    if(!file.exists()){
        throw new GestioFitxersException("Error. No es pot fer el "
            + "canvi de nom, " + nom + " no existeix.");
    }
    if(!file.renameTo(fileNou)){
        throw new GestioFitxersException("Error. No s'ha pogut "
            + "canviar de nom, " + nom + ".");
    }
    actualitza();
}
```

Des de la interfície gràfica, la modificació de la data s'aconsegueix seleccionant l'opció *Propietats* des del menú *Fitxer* i un cop s'obri el quadre de diàleg, seleccionant la pestanya *Modificar*.

L'operació que permet modificar la data de modificació de fitxers o carpetes és `setLastModified`, la qual rep per paràmetre un valor `long` representant la data a assignar, en el mateix format que la retorna `lastModified`. La implementació a l'aplicació que estem construint serà:

```
@Override
public void setUltimaModificacio(String nom, long dataIHora)
                                    throws GestioFitxersException {
    File file = new File(carpetaDeTreball, nom);
    if(!file.exists()){
        throw new GestioFitxersException("Error. No es pot "
            + "obtenir modificar " + nom + ", no existeix.");
    }
    file.setLastModified(dataIHora);
}
```

Finalment farem esmentdels permisos d'accés que el sistema operatiu atorga. Es pot interrogar `File` per saber si un element es pot llegir (`canRead`), es pot escriure (`canWrite`) o be si és executable (`canExecute`). De manera similar a la resta de propietats, és possible modificar aquests permisos amb els corresponents mètodes: `setReadable`, `setWritable` o `setExecutable`. La classe `File` disposa de dues versions per a cada mètode, la versió en què es rep només un paràmetre de tipus booleà i la versió en què se'n reben dos. La darrera versió és útil per a sistemes operatius que distingeixen entre permisos de propietari i permisos d'altres usuaris. El primer valor lògic representa l'estat en què es vol deixar el permís, mentre que el segon valor representa si el canvi afectarà només el propietari (valor cert) o bé afectarà tothom (valor fals). Així, `file.setReadable(false, true)` farà que l'element al qual es trobi vinculat la instància `File` no sigui llegible pel propietari. Els permisos per a la resta d'usuaris no es modifiquen i continuen establerts al seu valor original. En canvi, `file.setReadable(true, false)` farà que`File` sigui llegible tant pel propietari com per la resta d'usuaris. Les versions d'un únic paràmetre són equivalents a les darreres amb el segon valor fixat a true.

Per l'exemple que implementem, usarem només les versions d'un únic paràmetre.

```
@Override
public boolean esPotEscriure(String nom)
                    throws GestioFitxersException {
    File file = new File(carpetaDeTreball, nom);
    if(!file.exists()){
        throw new GestioFitxersException("Error. No es pot obtenir "
            + "informació de " + nom + ", no existeix.");
    }
    return file.canWrite();
}

@Override
public boolean esPotExecutar(String nom)
                    throws GestioFitxersException {
    File file = new File(carpetaDeTreball, nom);
    if(!file.exists()){
        throw new GestioFitxersException("Error. No es pot obtenir "
            + "informació de " + nom + ", no existeix.");
    }
    return file.canExecute();
}

@Override
public void setEsPotLlegir(String nom, boolean permis)
                                throws GestioFitxersException {
    File file = new File(carpetaDeTreball, nom);
    if(!file.exists()){
```

```
28          throw new GestioFitxersException("Error. No es pot modificar "
29              + nom + ", no existeix.");
30      }
31      file.setReadable(permis);
32  }
33
34  @Override
35  public void setEsPotEscriure(String nom, boolean permis)
36                                  throws GestioFitxersException {
37      File file = new File(carpetaDeTreball, nom);
38      if(!file.exists()){
39          throw new GestioFitxersException("Error. No es pot modificar "
40              + nom + ", no existeix.");
41      }
42      file.setWritable(permis);
43  }
44
45  @Override
46  public void setEsPotExecutar(String nom, boolean permis)
47                                  throws GestioFitxersException {
48      File file = new File(carpetaDeTreball, nom);
49      if(!file.exists()){
50          throw new GestioFitxersException("Error. No es pot modificar "
51              + nom + ", no existeix.");
52      }
53      file.setExecutable(permis);
54  }
```

Les operacions de copiar o moure fitxers les veurem a l'apartat de "Gestió de Continguts" ja que requereixen traspassar el contingut d'una ubicació a una altra.

Per poder fer servir l'aplicació en plataformes Windows i poder seleccionar entre les diverses unitats d'emmagatzematge, caldrà complementar la implementació amb dos mètodes que ens aportin aquesta informació:

```
1   @Override
2   public int numArrels(){
3       return File.listRoots().length;
4   }
5
6   @Override
7   public String nomArrel(int id){
8       return File.listRoots()[id].toString();
9   }
```

Finalment, es podran també implementar les dues últimes operacions contemplades a la interfície `GestioFitxers`, que ens aportaran una visió més estètica de la interfície afegint-hi informació addicional.

```
1   @Override
2   public String getEspaiDisponibleCarpetaTreball() {
3       ByteFormat format = new ByteFormat("#,##0.00");
4       return format.format(carpetaDeTreball.getUsableSpace());
5   }
6
7   @Override
8   public String getEspaiTotalCarpetaTreball() {
9       ByteFormat format = new ByteFormat("#,##0.00");
10      return format.format(carpetaDeTreball.getTotalSpace());
11  }
```

2. Gestió del contingut dels fitxers

En aquest apartat també ens serviremd'un exemple per continuar avançant en l'estudi de les classes de Java que ens permetin gestionar els fitxers. Ara aprofundirem, específicament, en l'estudi de la gestió del contingut de fitxers.

Consulteu l'annex "Biblioteques del mòdul" en la secció "Annexos" per obtenir el codi de partida de l'aplicació, exemple que il·lustra l'ús de les classes Java.

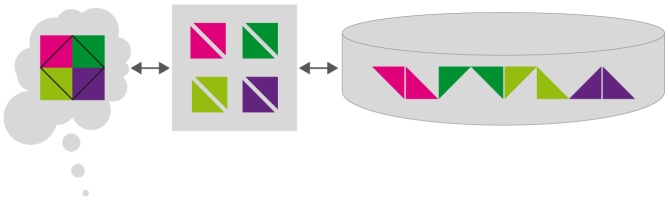
L'exemple que ens il·lustrarà l'aplicació i ús d'aquestes classes enllaça també amb l'anterior aplicació de gestió de fitxers, ampliant algunes de les seves utilitats, com ara la còpia o el trasllat de fitxers en diferents ubicacions o l'edició del seu contingut. Per tant, el punt de partida serà l'aplicació que heu implementat i que podeu aconseguir també als annexos d'aquests materials.

Abans però, caldrà clarificar el concepte de fitxers com a magatzems de contingut, i el concepte de contingut com a flux de dades.

2.1 Magatzems i fluxos de dades

Des del punt de vista de l'usuari, els fitxers són magatzems permanents d'informació guardada en forma de dades estructurades i ben organitzades de manera que la seva identificació i conseqüent interpretació siguin fàcils. Aquesta és una perspectiva centrada en la *representació mental* de la informació i s'anomena *estructura lògica*.

FIGURA 2.1. Estructura lògica i estructura física de les dades



L'Estructura lògica o representació conceptual de les dadesva canviant a mida que avança pels diferents nivells d'abstracció fins a arribar al dispositiu d'emmagatzematge. La forma final de les dades es coneix també com estructura física.

Ara bé, en el camí de la persistència, la representació mental original passarà per diferents nivells d'abstracció fins arribar a l'emmagatzematge físic (vegeu la figura 2.1). Cada nivell imposarà les seves limitacions a la representació del nivell anterior introduint canvis que alteraran l'estructura original. El primer nivell d'abstracció el trobem en el llenguatge de programació utilitzat, que depenent de les estructures d'informació suportades, els tipus de dades permesos, la representació en memòria d'aquest, etc., alterarà en alguna mesura la representació de partida. El Sistema Operatiu imposarà també les seves restriccions i ho farà també el *driver* o adaptador al sistema d'emmagatzematge, o els mecanismes físics i els suport escollits en últim terme. L'estructura final, com les dades acaben emmagatzemant-se, s'anomena també *estructura física*.

Parlarem d'**estructura lògica de la informació**quan ens referim a estructures de dades properes a la representació mental. En contraposició, parlarem d'**estructura física de les dades** quan ens referim a estructures de dades allunyades d'aquesta.

Generalment, els llenguatges d'alt nivell com Java suporten estructures de dades força properes a la representació mental. D'aquí que se solen classificar com a estructures lògiques. Malgrat tot, són estructures difícilment traslladables al nivell físic, ja que es troben farcides de referències que apunten a d'altres zones de la memòria principal on s'ubiquen part de les dades de l'estructura global. Requeriran, per tant, una adaptació a l'estructura física quan calgui emmagatzemar-les.

2.1.1 Seriació i flux de dades

Les referències a memòria són dades transitòries que varien en cada execució. Si guardéssim de forma persistent les referències a memòria, en recuperar-les obtindríem dades totalment incoherents perquè apuntarien a dades inexistents o ocupades per altra informació.

Si volem un emmagatzematge i una recuperació de les dades eficaç, caldrà assegurar que totes les dades referenciades s'emmagatzemen també al fitxer i es relacionen entre elles de forma que sigui factible localitzar-les i recuperar-les conjuntament.

La forma més senzilla de fer-ho consisteix a compactar les successives dades referenciades agrupant-les, una darrera l'altra, en una única seqüència de dades primitives lliure de referències.

Aquest procés s'anomena **seriació** i cal utilitzar-lo per transformar estructures complexes de la memòria principal en sèries de dades compactes i fàcilment emmagatzemables.

Les dades primitives s'emmagatzemen en els fitxers sense canvis, copiant literalment la seqüència de bits de la memòria al suport físic. Això significa, també, que l'emmagatzematge de la seqüència de dades primitives consistirà simplement en una còpia literal dels bits de la seqüència. És per això que la seqüència de dades primitives s'anomena també seqüència de bits.

El concepte de **seqüència de bits** ens aporta una visió estàtica dels fitxers en el sentit de magatzem de la seqüència.

Tots sabem que l'aigua corrent roman emmagatzemada en pantans i dipòsits abans de rajar de les nostres aixetes, però en el nostre imaginari els pantans i els dipòsits

queden lluny i en parlar d'aigua corrent tendim més aviat a pensar en aixetes i canonades, que són els estris que en últim terme usem per controlar l'aigua corrent.

De forma similar, des del punt de vista de l'aplicació, el que realment cobra importància és la transferència de dades, més que no pas el magatzem. L'estri que ens permet controlar aquestes transferències, de forma similar a les aixetes i canonades, l'anomenem **flux de dades**. És un concepte associat a la transmissió seqüencial d'una sèrie de dades des de l'aplicació al dispositiu d'emmagatzematge o a l'inrevés.

El concepte de **flux** ens dóna un visió dinàmica dels fitxers entenent-los com a processos d'intercanvi seqüencial de dades entre el magatzem i l'aplicació.

El concepte de flux (*stream*) no és pas exclusiu dels fitxers, sinó que es tracta d'una abstracció relacionada amb qualsevol procés de transmissió d'informació entre un contenidor de dades i una zona de la memòria primària controlada per l'aplicació.

Anomenarem **flux d'entrada** aquells processos de transmissió d'informació que traslladin dades des d'un contenidor qualsevol a la zona de memòria primària controlada per l'aplicació. És a dir, que enviïn dades a fi de ser processades durant l'execució.

Anomenarem **flux de sortida** aquells processos de transmissió d'informació que traslladin dades des de la zona de memòria primària controlada per l'aplicació cap a qualsevol altre contenidor de dades.

La transmissió de dades per mitjà de fluxos s'entén sempre de manera seqüencial, és a dir, l'ordre en què surten les dades de l'emissor és el mateix en què arriben al receptor.

2.1.2 Fluxos i tipus de dades

El concepte de flux es contraposa al concepte de tipus de dada en el sentit que en compactar totes les dades, els límits d'aquestes es difuminen fins a desaparèixer, donant com a resultat un flux continu de bytes, ja que la informació compactada no conté cap mena de marca d'on comença o acaba una dada.

Exemples de compactació de dades en un flux

Imaginem un tipus de dada numèric de 16 bits. La representació en binari del número 24941 seria 0110000101101101, i la del número 26979 seria 0110100101100011. En compactar ambdós valors en un flux obtindríem la seqüència 01100001011011010110100101100011.

Imaginem ara una cadena de text que conté la paraula amic en binari; el caràcter a esrepresenta: 01100001, el caràcter i: 01101101, el caràcter m: 01101001 i el caràcter c: 01100011. El flux de dades d'aquest text seria també 01100001011011010110100101100011.

De la mateixa manera, el número 1634560355 representat en 32 bits tindria també la mateixa seqüència 01100001011011010110100101100011.

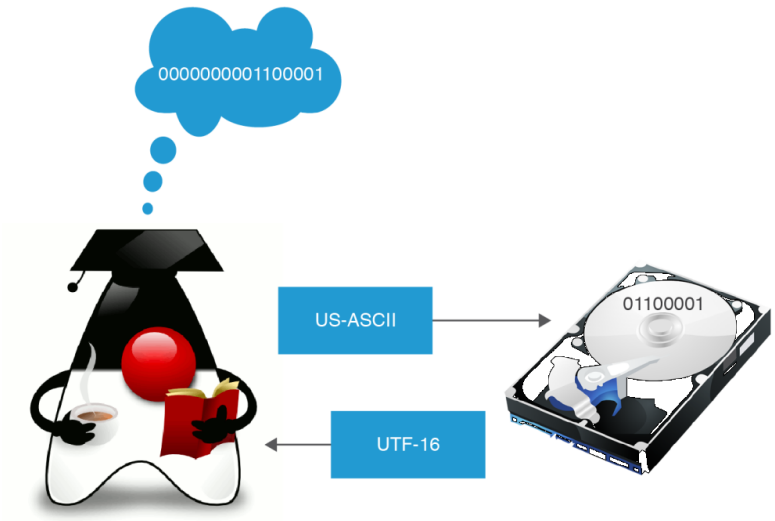
Donada una seqüència de bits, no hi ha manera, a priori, de saber la composició de les seves dades. Només si coneixem la mida i l'ordre en què es van compactar, podrem recuperar els valors originals del flux.

Sortosament, els tipus bàsics de dades tenen una mida prefixada i l'emmagatzematge es fa considerant tots el bits. Així, el valor numèric 1 d'un tipus enter de 16 bits s'empaquetarà com 0000000000000001 i ocuparà el mateix que qualsevol altre número del mateix tipus.

Les classes de Java orientades a fluxos transfereixen les dades de manera transparent al programador. No cal indicar la quantitat de bits que cal transferir, sinó que es dedueix a partir del tipus de dada que la variable representa.

Hi ha, però, una excepció amb el tipus *char*. La multitud d'estàndards de codificació de caràcters existents en l'actualitat i la diversitat de formats utilitzats a l'hora d'implementar les codificacions, usant segons el cas 8, 16, 32 bits o fins i tot una longitud variable en funció del caràcter a representar, fan que sigui molt difícil tractar aquest tipus de dada com una simple seqüència de bytes.

FIGURA 2.2. Internament, Java treballa amb caràcters de 16 bits per tal de suportar múltiples alfabets a banda de l'occidental



El programa podrà gestionar formats de caràcters de diverses longituds (8, 16 o 32 bits) usant classes de fluxos especialitzades en caràcters que realitzen la conversió de forma automatitzada.

Internament, Java representa el tipus caràcter amb una codificació UNICODE de 16 bits (UTF-16) per tal de suportar múltiples alfabets a banda de l'occidental (figura 2.2). Tot i així, és capaç de gestionar fonts de dades (fitxers entre d'altres) de diverses codificacions (ASCII, ISO-8859, UTF-8, UTF-16...). En funció de la codificació escollida, el nombre de bits usats en l'emmagatzematge variarà. Es fa necessari, doncs, un tractament especial a l'hora de fer la seriació i deseriació d'aquestes dades. Java disposa d'un jerarquia específica de classes orientades a fluxos de caràcters per tal de fer aquests canvis i transformacions totalment transparents al programador.

2.2 Manipulació dels fluxos

A Java trobem diverses jerarquies de classes orientades a fluxos: la jerarquia orientada a bytes, la jerarquia orientada a caràcters i una nova jerarquia que intenta aprofitar les utilitats dels sistemes operatius per obtenir transferències de dades molt eficients. Anem a estudiar-les.

2.2.1 Fluxos orientats a Bytes

Es tracta d'una important jerarquia de classes que intenta donar resposta a la multiplicitat funcional de fluxos de dades que es pugui donar en qualsevol situació. A l'arrel de la jerarquia trobem les classes `InputStream` i`OutputStream`. Són classes abstractes amb l'objectiu de definir el comportament comú de la resta de classes de la jerarquia. Així, `InputStream`, la superclasse d'on hereten la resta de classes orientades a ***fluxos de Bytes d'entrada***, especifica la sintaxi que han de tenir els mètodes de lectura de qualsevol font de dades, o les operacions de consulta que seran necessàries. `OutputStream`, en canvi, especifica els requeriments del comportament i la sintaxi de les classes orientades a ***fluxos de Bytes de sortida***. És a dir, escriptura de les dades a la font de dades.

Com s'especifiquen els mètodes d'InputStream

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 s'hagi detectat el final, es llançarà una excepció del tipus *IOException*. Es tracta d'un mètode abstracte, que les classes especifiques sobreescriuran adaptant-lo a una font de dades concreta, a un format determinat o una metodologia d'obtenció de dades adequada.

- *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. El mètode romandrà bloquejat en cas que no hi hagi cap dada disponible en el flux ni es trobi el final de la seqüència. Si se li passa un

paràmetre *null* es llançarà l'excepció *NullPointerException;* en canvi, si la longitud del paràmetre fos zero mai es llegiria cap byte però no es produiria cap excepció.

- *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*. El mètode retorna el nombre de bytes realment llegits com un enter. Si no hi ha bytes a llegir es bloquejarà esperant que arribin dades a través del flux o que arribi el final de la seqüència. Si troba el final de la seqüència retornarà -1, en cas contrari el nombre de Bytes llegits. Cal que *buffer* no sigui *null*. Si *len* fos negatiu o *offset* fos negatiu o *offset+len* fos més gran que la longitud del vector (*buffer*), es llançaria una excepció de tipus `IndexOutOfBoundsException`.
- Els bytes llegits s'emmagatzemen al vector de bytes començant per la posició *ofsset* de vector.
- *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. Aquest mètode és merament informatiu, ja que no hi ha garantia que en el moment que es faci la lectura o el salt les condicions no hagin canviat. Si el flux no es pogués llegir per qualsevol raó, es llançarà una excepció.
- *long* **skip** (*long bytesToSkip*). Salta i descarta *bytesToSkip* bytes de dades d'aquest flux d'entrada. No hi ha garantia que el mètode acabi saltant exactament *bytesToSkip* bytes, atès que pot ser que per diverses raons acabi saltant menys. Es retorna el nombre real de bytes omesos. Si *bytesToSkip* és zero o negatiu, no es produirà cap salt.
- void **close**(). Tanca aquest flux d'entrada i allibera els recursos del sistema associats.

Com s'especifiquen els mètodes d'OutputStream

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.

Totes les classes que deriven de les dues superclasses (`InputStream` i `OutputStream`) són també orientades a bytes i, o bé concreten els mètodes abstractes implementant-los específicament per una font de dades determinada (memòria, fitxers, *sockets*, etc.) o bé en milloren la funcionalitat (ús de tipus bàsics en comptes de bytes, ús de doble *buffer* per poder disposar de dos punters dins la font de dades, etc.). Sobre la concreció de la font de dades, farem especial esment a la classe `FileInputStream` i l'homòloga de sortida `FileOutputStrem`, ja que implementen l'accés als fitxers.

Fluxos de dades contra fitxers

Sobre aquestes classes, cal destacar que els constructors admeten un paràmetre de tipus `File` o de tipus cadena de caràcters representant la ruta del fitxer:

- **FileInputStream**(*File fitxer*) o**FileInputStream**(*String ruta*) Obren el fitxer especificat com a paràmetre, en mode lectura i creen una instància amb la qual podrem llegir el contingut usant els mètodes heretats d'`InputStream` (*read*, *skip*, *available*, etc.).
- **FileOutputStream**(*File fitxer*) o **FileOutputStream**(*String ruta*). Creen un fitxer a la ruta especificada com a paràmetre. En cas que ja existeixi el fitxer, n'esborren el contingut. Un cop creat el fitxer, s'obre en mode escriptura. La instància creada amb aquest constructor permetrà escriure en el fitxer usant els mètodes heretats d'`OutputStream` (*write*, *flush* i *close*).

A més, la classe `FileOutputStream` disposa de dos constructors addicionals per poder indicar al sistema que es vol obrir un fitxer existent en mode escriptura sense perdre'n el contingut:

- **FileOutputStream**(*File fitxer*, *boolean append*) o **FileOutputStream**(*String ruta*, *boolean append*). Si el paràmetre *append* és cert, obriran el fitxer existent especificat per la ruta del primer paràmetre, en mode escriptura. Les dades escrites des d'aquesta instància s'afegiran sempre al final del contingut existent. Si `append`té el valor *false* el fitxer es crearà (o

s'esborrarà i crearà de nou si ja existís) igual que si s'hagués instanciat amb algun dels constructors anteriors.

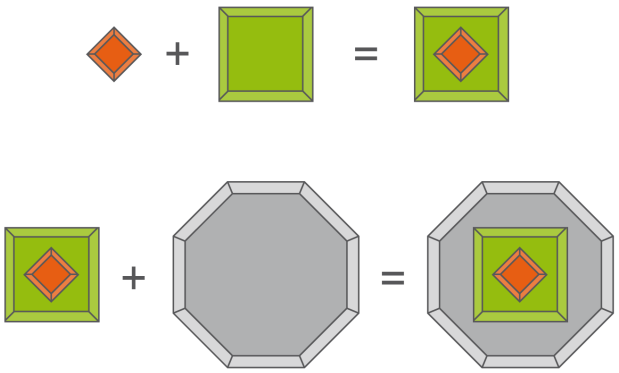
A banda dels constructors, la resta de mètodes coincideixen amb els de les seves superclasses respectives. Quelcom de semblant succeeix amb altres fonts de dades hereves també de les superclasses. Cal veure, però, que els mètodes descrits no són gaire útils a l'hora d'usar-los en una aplicació, ja que rarament es treballa amb bytes i malgrat que la conversió de dades bàsiques a vectors de bytes no és gaire complicada, afegeix una complexitat supèrflua que ens estalviaríem si les classes de fluxos incorporessin mètodes conversors.

Una solució fàcil podria ser incorporar aquestes utilitats a les superclasses (`InputStream`i `OutputStream`) de manera que s'heretessin en tota la jerarquia. Tot i l'aparent senzillesa de la solució, Java va decantar-se per una altra solució força més complexa, que li ha valgut en més d'una ocasió alguna que altra crítica, atès que obliga el programador a conèixer una extensa jerarquia de classes i realitzar diverses instanciacions per acabar obtenint un únic objecte *Stream.*

Fluxos decoradors de fluxos bàsics

La solució escollida es coneix tècnicament amb el nom de *decorator* usant la terminologia anglosaxona, és a dir, *decorador* (figura 2.3). Consisteix a embolcallar dos o més objectes, un dins l'altre, com si es tractessin de nines russes. Cada embolcall aporta una certa funcionalitat extra ("decora" l'objecte intern de forma diferent a l'original). D'aquesta manera és possible crear instàncies "a gust del consumidor", seleccionant només la funcionalitat que sigui necessària. També pot resultar més fàcil crear noves funcionalitats sense problemes de compatibilitat, ja que mai es modifiquen les classes originals, sinó que es crea un nou embolcall.

Figura 2.3. Representació gràfica que simbolitza el concepte "decorador"



Cada embolcall modifica la forma de l'anterior.

Veiem un exemple, suposem que partim d'un flux original de tipus `FileOutputStream` al qual li volem donar la capacitat de disposar de memòria intermèdia per agilitzar el procés de transferència de dades fent servir la classe `BufferedOutputStream`. A més, volem dotar al flux final de l'automatisme

de conversió de dades bàsiques a bytes. Farem servir `DataOutputStream`, i el procés de construcció d'una instància com aquesta seria:

```
...
FileOutputStream fos = new FileOutputStream(ruta);
BufferedOutputStream bos = new BufferedOutputStream(fos);
DataOutputStream streamFinal = new DataOutputStream(bos);
...
```

o bé optimitzant el nombre de variables,

```
...
DataOutputStream streamFinal = new DataOutputStream(
                                    new BufferedOutputStream(
                                        new FileOutputStream(ruta)));
...
```

Tots els decoradors de tipus byte formen part també de la jerarquia d'Streams. És a dir, els decoradors són també, a la vegada, *Streams* d'entrada o de sortida. Així podem definir l'ordre de construcció segons ens interessi.

A la figura 2.4 podeu veure un esquema de les jerarquies de fluxos d'entrada i sortida orientats a bytes. Observeu que la majoria de decoradors (color taronja) hereten de `FilterInputStrem` o `FilterOutputStream`,segons el cas. Probablement l'estructura interna de `ObjecteInputStream`/ `ObjectOutputStream`hagi obligat a descendir directament de les arrels corresponents, malgrat que es tracti també d'un *decorador* semblant als altres.

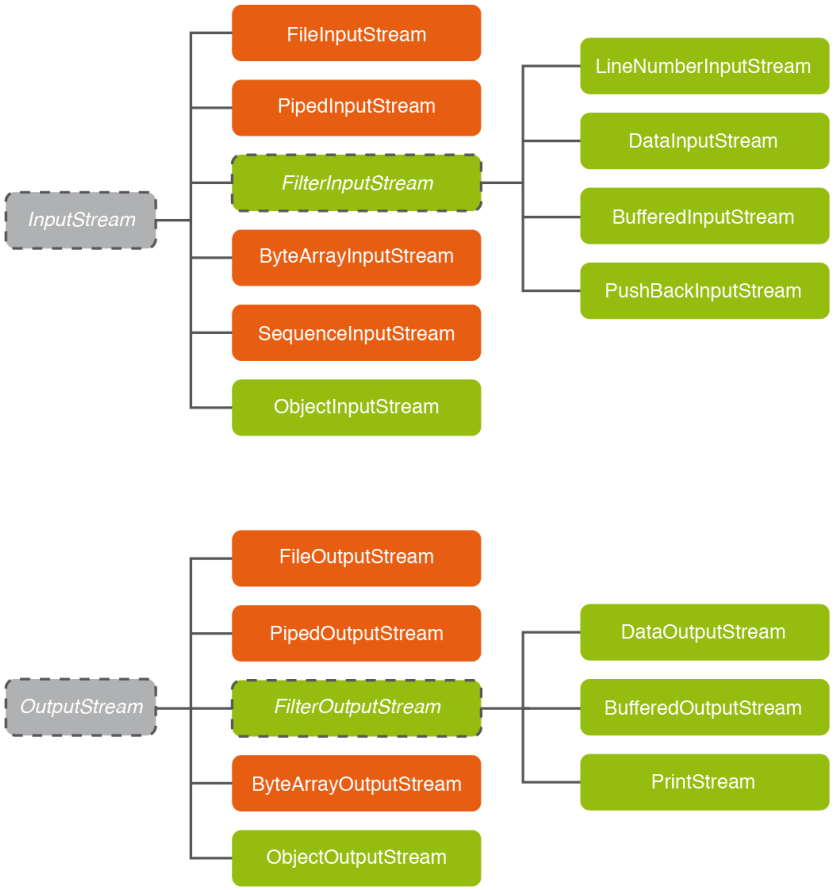
De forma breu, direm que `PrintStream` és un decorador amb utilitats destinades a la presentació de les dades (precisió en els decimals numèrics, format de les dates, etc.). Podríem dir que és una utilitat encarregada del format extern de les dades en contraposició a DataOutputStream, que s'encarrega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

`DataInputStream` i `BufferedInputStream` són el homòlegs de `DataOutputStream` i `BufferedOutputStream`, respectivament. La primera tindrà la funció de convertir els bytes del flux en dades de tipus bàsic de l'aplicació, i la segona 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.

FIGURA 2.4. Jerarquia de les classes de Java que implementen fluxos orientats a bytes



A l'esquema hem pintat de color taronja les classes de tipus "decorador", de color blau les que no ho són i de color verd les classes abstractes que no s'han definit encara. Les classes abstractes s'han escrit en cursiva i es simbolitzen amb línies de punts.

Altres fluxos importants

`ByteArrayOutpuStream` permet crear un flux de dades cap a la memòria RAM de l'aplicació i definir un *buffer* de memòria de tipus flux. De forma idèntica, `ByteArrayInputStream`defineix també un *buffer* de memòria però de tipus flux d'entrada, en el qual les dades es transfereixen del *buffer* a l'aplicació.

Recordem que la interfície *serializable* no té declarat cap mètode, sinó que només serveix per marcar quins objectes es permeten seriar i quins no.

Volem destacar també `ObjectInputStream` i `ObjectOutputStream`, les quals permeten automatitzar la seriació de qualsevol objecte que implementi la interfície *java.io.Serializable* des de l'aplicació cap a l'*Stream* que decora (sortida), o bé des de l'*Stream* decorat cap a l'aplicació (entrada).

Veurem amb més detall les classes `ObjectInputStream` i `ObjectOutputStream` a la secció "Seriació d'objectes" de l'apartat "Persistència d'objectes en fitxers" d'aquesta mateixa unitat.

Finalment, cal indicar que `SequenceInputStream` no és pròpiament un Stream, sinó una utilitat per gestionar i concatenar múltiples Streams. Estrictament no es pot considerar com un decorador, ja que no modifica un únic objecte sinó que gestiona la seqüenciació de diversos objectes. Òbviament no existeix una classe homòloga de sortida, ja que implicaria la divisió d'un flux de dades en múltiples fluxos i la seva implementació seria probablement més complexa que l'ús directe de cada un dels fluxos que es necessitin.

Fluxos d'accés relatiu

Tots els fluxos que acabem de veure treballen de forma seqüencial i en una única direcció. Existeix, però, un flux orientat a bytes que permet l'accés relatiu a qualsevol part del seu contingut. Ens referim a `RandomAccesFile`, un flux especialment dissenyat per permetre accés no seqüencial (anomenat també relatiu o aleatori) dins un fitxer. És un flux bidireccional, és a dir, un flux que permet tant l'escriptura com la lectura, malgrat que es pot configurar com a flux unidireccional. És tracta d'una classe independent de les dues jerarquies estudiades fins ara i per tant no serà possible "*decorar-lo*" amb cap dels "*decoradors*" específics de les jerarquies anteriors. Tot i així, *RandomAccesFile* implementa tots els mètodes de `DataInputStream`i de `DataOutputStream`, de manera que no li cal l'embolcall per treballar amb els tipus de dades primitius. En certa forma, podríem dir que es comporta com un `DataStream` d'entrada i sortida amb utilitats extres adequades per gestionar l'accés relatiu. Així, per exemple, disposa del mètode `seek` per realitzar salts dins el fitxer i aconseguir moure el punter intern de lectura o escriptura, del mètode `getFilePointer` per obtenir la posició del punter en el moment de la petició, o del mètode `length` per determinar la mida total de l'arxiu.

Els constructors de la classe admeten sempre dos paràmetres. El primer indica la ruta del fitxer que es desitja obrir i el segon, el mode en què es desitja fer-ho (només lectura o lectura i escriptura). La ruta pot indicar-se mitjançant un `String` o bé usant una instància de `File`. El mode admetrà qualsevol de les següents combinacions: "r" indicarà que es desitja obrir en el mode de *només lectura* i "rw" indicarà que també serà possible escriure en el fitxer. A banda d'aquestes dues opcions, podem indicar "rwd" per forçar que tot el que escrivim en el flux es buidi en el fitxer de manera immediata.

2.2.2 Fluxos orientats a caràcters

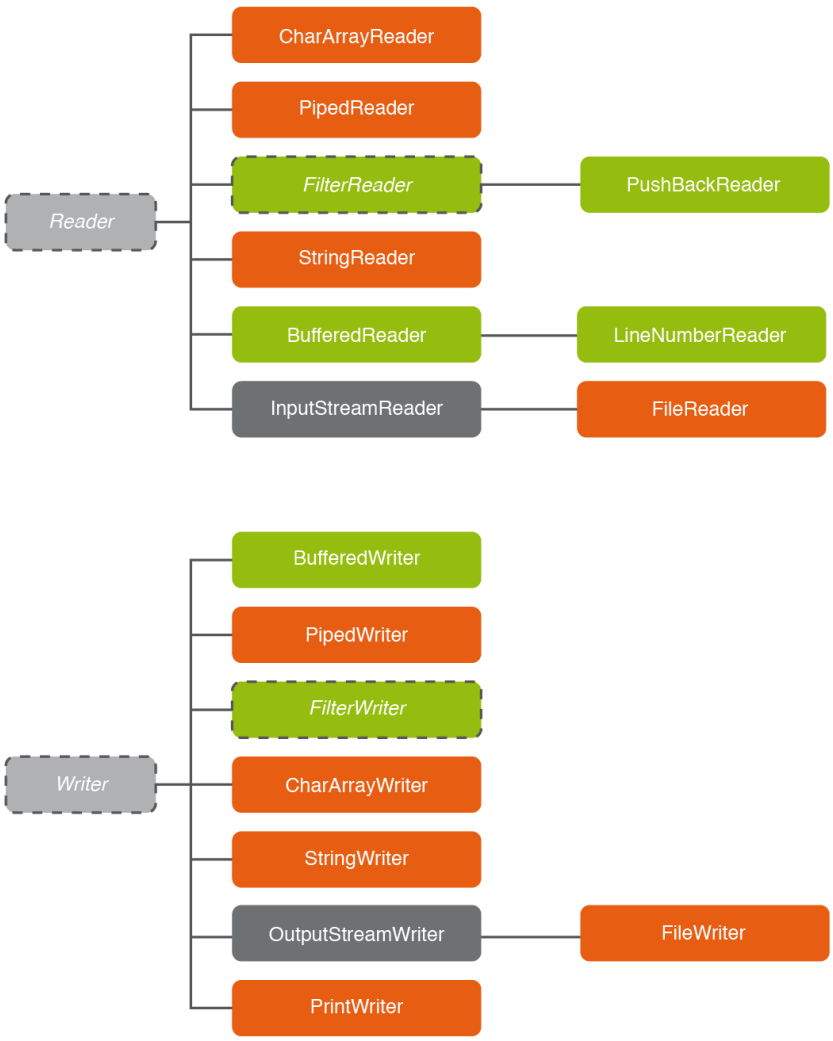
Treballar amb caràcters implica una gran dificultat, a causa sobretot de la diversitat de formats existents. Per poder solucionar-ho, Java disposa de dues jerarquies de classes semblants a les jerarquies de fluxos orientats a bytes.

Malgrat que l'estructura interna de les jerarquies no és ben bé igual en ambdós tipus de fluxos, les classes finalment implementades tenen característiques equivalents. Així, per exemple, `CharArrayReader/CharArrayWriter` són les equivalències de `ByteArrayInputStream/ ByteArrayOuputStream`, `PipedReader/ PipedWriter` es corresponen amb `PipedInputStream/ PipedOutputStream`, `BufferedReader/ BufferedWriter` les classes equivalents de `BufferedInputStream/ BufferedOutputStreamm`, i d'igual forma `FileReader/ FileWriterho` són de `FileInputStream/ FileOutputStream`, etc.

Com es pot veure a la figura 2.5, on es mostra la jerarquia dels fluxos orientats a caràcters, existeixen també "*decoradors*" de *Readers* i *Writers* que, de forma semblant als "*decoradors"* homòlegs orientats a bytes, aporten diferent funcionalitat

i poden combinar-se entre ells per tal d'obtenir diferents instàncies adaptades a cada necessitat.

Figura 2.5. Jerarquia de les classes de Java que implementen fluxos orientats a caràcters



A l'esquema hem pintat de color taronja les classes de tipus "decorador", de color blau les que no ho són i de color verd les classes abstractes que no s'han definit encara. La classe "InputStreamReader" i "OutputStreamWriter" juguen el paper "d'adaptadors" entre els fluxos orientats a bytes i els fluxos orientats a caràcters. Les classes abstractes s'han escrit en cursiva i es simbolitzen amb línies de punts.

Adaptadors de bytes a caràcters

Adaptadors

En programació orientada a objectes, les classes que aconsegueixen fer compatibles dues jerarquies independents s'anomenen adaptadors. Les classes InputStreamReader i OutputStreamWriter són d'aquest tipus perquè aconsegueixen adaptar qualsevol flux orientat a bytes i transformant-lo en un Reader o Writer segons el cas.

De la jerarquia de *Readers* o *Writers* destacarem la classe `InputStreamReader` i `OutputStreamWriter`. L'objectiu d'aquestes és transformar qualsevol flux orientat a bytes en un flux orientat a caràcters. La construcció d'aquestes instàncies és similar a la dels decoradors, els constructors esperen per paràmetre un InputSream o un OutputStream, segons es tracti d'una classe hereva de*Reader* o de *Writer,* i se'l guarden internament.

Cada cop que s'executi algun mètode d'introducció o extracció de contingut, la instància s'encarregarà de traduir la seqüència de text a una seqüència de bytes i passar-la a l'*Stream* original, o bé recollir la seqüència de bytes des de l'*Stream* original i convertir-la en seqüència de caràcters.

Són classes útils per a aplicacions que treballen amb fluxos de bytes externs, el contingut del qual són caràcters. A més, també permeten manipular fluxos de caràcters codificats en diferents formats.

En el constructor es pot indicar, a més del flux original, el tipus de codificació pertinent. Així, per exemple, si d'una font externa (fitxer, xarxa, etc.) obtinguéssim fluxos de caràcters amb una codificació diferent de la usada per Java (ISO-8859-1, per exemple), caldria crear un Reader o un Writer, passant-li el flux de bytes i indicant que la codificació usada és ISO-8859-1. Vegem-ne un exemple en el qual suposarem que tenim una font externa que disposa d'un mètode anomenat getInputStream i getOutputStream, els quals obtenen fluxos de bytes d'entrada i sortida de la font:

```
InputStreamReader reader;
OutputStreamWriter writer;
InputStream input = fonDadesEnt.getInputStream();
OutputStream output = fontDadesSort.getOutputStream();

reader=new InputStreamReader(input, "ISO"-8859-1);
writer=new OutputStreamWriter(output, "ISO"-8859-1);

...
```

Les classes d'emmagatzematge intern, com ara `CharArrayReader`, `CharArrayWriter`, `StringReader`, `StringWriter`, `PipedReader`, `PipedWriter` usen sempre la codificació pròpia de Java (unicode de 16 bits), ja que emmagatzemen les dades a la memòria basant-se en els tipus dades de tractament de caràcters de Java (*char* i *String*).

Les classes `FileReader` o `FileWriter`agafen la codificació per defecte del sistema operatiu amfitrió. L'usuari no pot seleccionar diferents sistemes de codificació en crear les instàncies. Així, una màquina virtual Java que corri sobre Windows usarà, per defecte, la codificació ISO-8859-1, però si corre sobre Linux, la codificació serà UTF-8.

Tot i això, és possible llegir o escriure fitxers fent servir codificacions alternatives. Podem crear objectes `InputStreamReader` o `OutputStreamWriter` a partir de fluxos de bytes vinculats a fitxers i indicar la codificació alternativa adequada. Imaginem que tenim un fitxer codificat en ISO-8859-1 i volem llegir-lo des d'un sistema operatiu Linux. Caldrà fer:

```
FileInputStream in = null;
File fileIn = new File("/home/josep/tmp/fitxerProva.txt");
InputStreamReader reader = new InputStreamReader(
                                    new FileInputStream(fileIn),
                                    "ISO-8859-1");
int charsLlegits=0;
char[] buffer = new char[1000];
while(charsLlegits!=-1){
    charsLlegits=reader.read(buffer);
    ...
}
...
```

Recodificació de fonts de caràcters

Amb l'ús d'aquesta tècnica és fàcil fer una utilitat que permeti transformar fluxos d'un format a un altre:

```
1  public void copiaDeChars(Reader reader, Writer writer)
2                                                      throws IOException {
3      int charsLlegits=0;
4      char[] buffer = new char[1000];
5      while(charsLlegits!=-1){
6          writer.write(buffer, 0, charsLlegits);
7          charsLlegits=reader.read(buffer);
8      }
9  }
10
11 public void recodificar(InputStream input,
12     String codificacioIn,
13     OutputStream output,
14     String codificacioOut) throws IOException {
15
16     InputStreamReader reader = new InputStreamReader(input,
17         codificacioIn);
18     OutputStreamWriter writer = new OutputStreamWriter(output,
19         codificacioOut);
20     copiaDeChars(reader, writer);
21
22     if (reader != null) {
23         reader.close();
24     }
25     if (writer != null) {
26         writer.close();
27     }
28 }
```

A l'exemple, `copiaDeChars` fa una còpia del contingut d'un objecte `Reader` a un objecte Writer. La funció *recodificar* obté per paràmetre el flux origen (*input*) i la codificació usada en aquest magatzem (`codificacioIn`). També per paràmetre se li indicarà el flux de sortida (*output*) on es desitja recodificar el contingut original i la codificació amb la qual es desitja tornar a emmagatzemar (`codificacioOut`). Amb els dos primers paràmetres s'instanciarà un *Reader* per llegir el contingut fent servir la codificació original. Amb els dos darrers paràmetres es podrà instanciar un Writer adaptat a la codificaciódesitjada. Tant el *Reader* com el *Writer* instanciats es passaran a `copiaDeChars` per realitzar la còpia del primer en el segon i aconseguir així la recodificació del contingut copiat.

2.2.3 Implementació d'utilitats

Arribats a aquest punt, ja estem en disposició de crear un conjunt d'utilitats que ens serviran de base per a l'aplicació pràctica que realitzarem més endavant. Aquests mètodes els agruparem a la classe `Utilitats`per tal de tenir-los tots disponibles amb una única instància.

Com ja s'ha vist, per manipular els fluxos de dades és necessari definir un *buffer* d'intercanvi o vector. Normalment treballarem amb vectors de bytes excepte en el cas de treballar amb *readers* o *writers,* atès que aquests ens permeten treballar directament amb vectors de *chars*.

Per tal de poder-nos adaptar a les diferents necessitats segons la mida del fitxer de treball, definirem un atribut que ens indicarà la mida màxima en bytes del *buffer* d'intercanvi. Per defecte el definirem de 1/2 Giga, però implementarem un constructor específic per canviar-ne el valor.

```
public class Utilitats {
    private int capacitatMaximaBufferEnBytes = 524288;

    public Utilitats() {
    }

    public Utilitats(int midaBuffer) {
        capacitatMaximaBufferEnBytes=midaBuffer;
    }
    ...
```

La primera utilitat que implementarem serà la còpia de fluxos, tant de bytes com de caràcters.

```
public void copiaDeBytes(InputStream input, OutputStream output)
    throws IOException{
    int bytesLlegits=0;
    byte[] buffer = new byte[capacitatMaximaBufferEnBytes];
    while(bytesLlegits!=-1){
        output.write(buffer, 0, bytesLlegits);
        bytesLlegits=input.read(buffer);
    }
}

public void copiaDeChars(Reader reader, Writer writer)
                                                throws IOException {
    int charsLlegits=0;
    char[] buffer = new char[midaBytesACaracters(
                                capacitatMaximaBufferEnBytes
                            )];
    while(charsLlegits!=-1){
        writer.write(buffer, 0, charsLlegits);
        charsLlegits=reader.read(buffer);
    }
}
```

Fixeu-vos que per determinar la mida del vector de caràcters (*buffer* d'intercanvi) sense que sobrepassi la capacitat definida per`capacitatMaximaBufferEnBytes`caldrà dividir per 2 aquesta quantitat. En lloc de fer servir l'operació *divisió*, usarem l'operació de *desplaçament de bits* perquè és una operació més eficient.

Desplaçament de bits

L'operació de desplaçament de bits és una operació que el processador pot realitzar de forma molt més ràpida que la pròpia operació de divisió o multiplicació. Aprofitem la característica que en els tipus de dades int o long un desplaçament de bits a la dreta divideix el número per 2 i un desplaçament a l'esquerra multiplica per 2. Així, 24 >> 1 donarà com a resultat 12. Mentre que 12 << 1 és equivalent a 24.

```
public long midaBytesACaracters(long bytes){
    return (bytes>>1);
}

public long midaCaractersABytes(long caracters){
    return (caracters<<1);
}
```

A continuació, usant les utilitats de còpia que acabem d'implementar, en crearem una de nova per copiar fitxers, tant a nivell de bytes com de caràcters.

```
public void copiaFitxersDeBytes(File origen, File desti)
                                                    throws IOException{
    FileInputStream input = null;
    FileOutputStream output = null;
```

```
    try {
        input = new FileInputStream(origen);
        output = new FileOutputStream(desti);
        copiaDeBytes(input, output);
    } finally {
        input.close();
        output.close();
    }
}

public void copiaFitxersDeCaracters(File origen, File desti)
    throws IOException {
    FileReader reader = null;
    FileWriter writer = null;
    try {
        reader = new FileReader(origen);
        writer = new FileWriter(desti);
        copiaDeChars(reader, writer);
    } finally {
        reader.close();
        writer.close();
    }
}
```

Cal anar amb compte de tancar sempre els fitxers en acabar la còpia, per tal de no malbaratar recursos del sistema. Per això és important posar el tancament dins la sentència *finally*. D'aquesta manera, assegurem que malgrat que durant la lectura es produís una excepció, abans d'abandonar el mètode a la cerca d'alguna sentència *catch* que capturi l'error, s'executarà obligatòriament el tancament.

El tancament de fitxers, però, presenta certs problemes que exposarem a continuació, intentant esbossar alguna solució adequada. Quan cal tancar més d'un fitxer, malgrat que les sentències de tancament es trobin dins el *finally*, si una de les sentències falla provocant una excepció (degut al fet que es tracta d'un fitxer inexistent o que el sistema ha esborrat abans del tancament, o qualsevol altre motiu), les sentències de tancament situades immediatament després de la que ha provocat l'error no s'executarien i es malbaratarien recursos del sistema.

La solució ideal és molt pesada d'implementar, ja que ens obligaria a crear sentències *try/finally* imbricades per cada fitxer a tancar de manera que s'asseguri el tancament de tots els fitxers a tancar.

```
...
} finally {
    try{
        f1.close();
    }finally{
        try{
            f2.close();
        }finally{
            f3.close();
            ...
        }
    }
}
```

Hi ha, però, una solució intermèdia, que malgrat que no és ideal simplifica força la implementació i assegura el tancament de tots aquells fitxers que puguin tancar-se. Es tracta de capturar les excepcions degudes als tancaments i deixar-les sense reportar. Per aconseguir-ho, caldrà fer el tancament en un mètode independent que capturi l'excepció.

```
public void intentarTancar(Closeable aTancar){
    try {
        if (aTancar != null) {
            aTancar.close();
        }
    } catch (IOException ex) {}
}
```

D'aquesta manera, aconseguirem que, malgrat que es produeixi una error en el tancament d'algun fitxer, es continuïn executant la resta de tancaments sense complicar el codi.

```
...

} finally {
    intentarTancar(f1);
    intentarTancar(f2);
    intentarTancar(f3);
    ...
    intentarTancar(fn);
}
```

Òbviament, aquest sistema presenta el problema que l'error podria passar inadvertit. Per això caldria afegir algun sistema d'enregistrament dels errors, com ara els fitxers *log*. El que es pretén amb això és que el sistema d'errors sigui consistent, però sense caure en una rigidesa que dificulti la implementació i acabi encarint el producte.

Si no usem cap sistema d'enregistrament *log* podem fer servir la sortida d'error estàndard per informar dels possibles errors en els tancaments de fitxers. Així, finalment, el mètode `copiaFitxersDeBytes`, `copiaFitxersDeCaractersi`intentarTancarquedaran com es mostra a continuació:

```
public void copiaFitxersDeBytes(File origen, File desti)
    throws IOException{
    FileInputStream input = null;
    FileOutputStream output = null;
    try {
        input = new FileInputStream(origen);
        output = new FileOutputStream(desti);
        copiaDeBytes(input, output);
    } finally {
        intentarTancar(input);
        intentarTancar(output);
    }
}

public void copiaFitxersDeCaracters(File origen, File desti)
    throws IOException{
    FileReader reader = null;
    FileWriter writer = null;
    try {
        reader = new FileReader(origen);
        writer = new FileWriter(desti);
        copiaDeChars(reader, writer);
    } finally {
        intentarTancar(reader);
        intentarTancar(writer);
    }
}

```

```
29 public void intentarTancar(Closeable aTancar){
30     try {
31         if (aTancar != null) {
32             aTancar.close();
33         }
34     } catch (IOException ex) {
35         ex.printStackTrace(System.err);
36     }
37 }
```

Segur que en força ocasions necessitarem escriure una cadena `String` en un fitxer o a l'inrevés, convertir en una cadena `String` el contingut d'un fitxer de text.

```
1  public void copiarStringAWriter(String text, Writer writer)
2      throws IOException{
3      int length = text.length();
4      int offset = 0;
5      while(offset<length){
6          int charsAEscriure =
7             Math.min(midaBytesACaracters(capacitatMaximaBufferEnBytes),
8             length - offset);
9      writer.write(text, offset, charsAEscriure);
10     offset+=charsAEscriure;
11     }
12 }
13
14 public String copiarReaderAString(Reader reader) throws IOException{
15     StringBuilder stringBuilder = new StringBuilder();
16     int charsLlegits=0;
17     char[] buffer =
18         new char[midaBytesACaracters(capacitatMaximaBufferEnBytes)];
19     while(charsLlegits!=-1){
20         stringBuilder.append(buffer, 0, charsLlegits);
21         charsLlegits=reader.read(buffer);
22     }
23     return stringBuilder.toString();
24 }
```

Del primer mètode (`copiarStringAWriter`) destacarem que cal comprovar si és més gran el text a escriure que no pas el *buffer* d'intercanvi. En cas afirmatiu, caldrà escriure el text en diverses passades.

Del segon mètode (`copiarReaderAString`), cal explicar que `StringBuilder` és una classe especialment concebuda per concatenar cadenes de forma més eficient que l'operació suma. D'altra banda, el mètode `read` intenta llegir el nombre de caràcters que caben en el *buffer*, però no hi ha garantia que ho aconsegueixi; per això, cal assegurar que el fitxer es llegeix íntegrament, moment en el qual el mètode *read* retorni el valor -1.

Abans de continuar amb la implementació presentarem les innovadores classes de fitxers Java que s'han incorporat amb força a les darreres versions.

2.3 Fluxos eficients: Channels i Buffers

L'acrònim *nio* prové de *new input/output*.

Una de les crítiques constants que ha rebut Java en relació amb la biblioteca d'entrada i sortida és la seva baixa eficiència. Des de la versió 1.4 es va introduir

un nova biblioteca i es va reestructurar l'antiga per tal de guanyar velocitat en l'intercanvi de dades en els processos d'entrada i sortida. La biblioteca que dóna major eficiència es troba en el paquet *java.nio* i se sustenta bàsicament sobre dues jerarquies principals, les classes que implementen la interfície `Channel` i la jerarquia que hereta de *Buffer*. La reestructuració de la biblioteca antiga implica només canvis interns: s'han substituït estructures antigues per les classes del paquet *nio* a fi d'incrementar l'eficiència de totes les classes d'entrada i sortida Java.

2.3.1 Conceptes

Channel té el paper de connector a la font de dades, però no en el sentit dels fluxos, sinó més aviat com la porta d'accés al magatzem de dades. La missió d'un *buffer* seria doble. D'una banda, la d'introduir o extreure informació d'un *Channel*, i de l'altra, la de dotar de totes aquelles utilitats necessàries per gestionar còmodament l'intercanvi de dades des del punt de vista del programador (utilitat de conversió dels tipus bàsics, seriació i deseriació d'objectes, suport de diversos formats d'emmagatzematge, gestió del flux, etc.).

La importància del paquet *nio*, però, se sustenta en el fet que tant les implementacions de *Channel* com *Buffer* i els seu derivats utilitzen internament utilitats molt eficients, pròpies de cada sistema operatiu. És a dir, que per exemple la classe `FileChannel` o la classe `ByteBuffer` podrien tenir rendiments diferents en diferents sistemes operatius, ja que cada màquina virtual específica disposa del seu paquet *nio* reescrit i adaptat a les característiques de la plataforma on correrà.

2.3.2 Instanciació d'un Channel

En la versió 1.6 de Java, els Channels no es poden instanciar usant la típica instrucció *new*. Per obtenir un `Channel` és necessari que un altre objecte l'instanciï i ens el retorni en executar algun dels seus mètodes. L'objecte instanciador se sol conèixer amb el nom de *factory* quan la seva funció principal es limita a "fabricar" instàncies.

Es tracta d'una tècnica força usada en programació orientada a objectes, ja sigui per independitzar interfícies o classes abstractes, de les classes finals que les implementin o per restringir i gestionar totes les possibles instàncies generades en una aplicació. En el cas que ens ocupa es compleixen ambdós requisits; desconeixem la classe final, ja que com hem dit en darrer terme depèn del sistema operatiu i, a més, cal evitar, per raons d'eficiència i coherència de dades, que es multipliquin el nombre de canals associats a una mateixa font de dades.

En el context dels fitxers, el paper de *fabricants de canals* és assumit pels objectes de la jerarquia *Stream* Com ja s'ha comentat, la biblioteca *java.io* s'ha reestructurat

de manera que internament es fan servir `Channels` específics, concretament instàncies de la classe `FileChannel`. Des de la versió 1.4, les classes d'entrada i sortida disposen d'un mètode anomenat `getChannel` que retorna el canal usat internament.

Per obtenir un `FileChannel`, doncs, necessitarem sempre la instància de l'*Stream* corresponent, i si no la tinguéssim, hauríem de crear-la. Vegeu tot seguit com obtenir un `FileChannel` a partir d'un `FileInputStream` ja creat:

```
1 FileInputStream istream;
2 ...
3 FileChannel channel = istream.getChannel();
4 ...
```

Vegeu ara com obtenir un `FileChannel` a partir d'un `FileInputStream` que no necessitem:

```
1 FileChannel channel = new FileInputStream(ruta).getChannel();
2 ...
```

Per introduir o treure dades d'un canal ens cal usar un *Buffer*. La introducció es farà executant algun dels mètodes *read*, i l'extracció, usant `write`. A més, `FileChannel` disposa de mètodes per realitzar transferències massives de dades entre canals, de mètodes de posicionament dins el fitxer i de mètodes per gestionar el bloqueig de fitxers.

2.3.3 Instanciació d'un Buffer

ByteBuffer segueix la mateixa tècnica constructora que `FileChannel`, és a dir, no disposa de constructor públic i per tant precisa d'una classe instanciadora que faci el paper de *factory*. En aquest cas, és la pròpia classe `ByteBuffer`la que pren aquest paper per tal de poder servir la classe específica del sistema operatiu amfitrió. Disposa de dos mètodes instanciadors. Són mètodes estàtics anomenats `allocate` i `allocateDirect`.

```
1 ByteBuffer byteBufffer = ByteBuffer.allocate(1024);
2 ...
3 ByteBuffer byteBufffer = ByteBuffer.allocateDirect(1024);
```

La principal diferència entre `allocate` i `allocateDirect` és que el segon genera instàncies que estan molt més lligades al sistema operatiu. En funció del sistema amfitrió, la instància aconseguida amb `allocateDirec` pot arribar a ser molt eficient, però sempre gastarà més recursos de memòria i processament que la instància aconseguida amb *allocate*.

2.3.4 Buffers de bytes

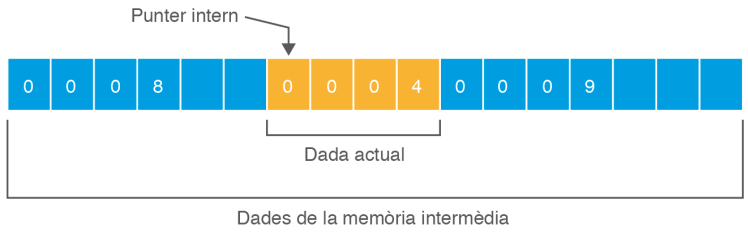
La principal classe de tipus `Buffer` responsable de l'intercanvi és `ByteBuffer`. S'anomena així perquè internament les dades estan representades en bytes. Tot i això, cal dir que aquesta classe ja disposa de mètodes conversors per a tots els tipus bàsics de Java, tant de lectura (`getInt`, `getFloat`, `getDouble` o `getChar`) com d'escriptura (`putInt`, `putFloat`, `putDouble` o `putChar`).

Accés absolut versus accés relatiu al buffer

Tant els mètodes d'escriptura com els de lectura admeten un accés relatiu o absolut. Internament, el *buffer* incorpora un punter a les dades que avança a mida que es llegeix o s'escriu (usant el mètode `getXXX` o `putXXX`), però és possible escriure o llegir una posició determinada del *buffer* indicant-la com a paràmetre. En aquest cas parlem de lectura o escriptura absoluta, ja que l'acció es realitza a la posició indicada. A més, cap de les accions absolutes incrementen el punter intern.

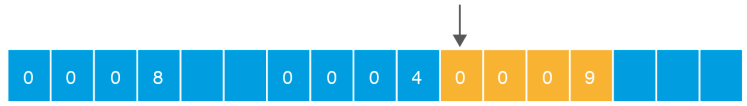
Sigui l'esquema de la figura 2.6, la representació d'un *buffer* amb el punter intern assenyalant un enter (int).

Figura 2.6. Buffer amb el punter intern assenyalant un enter



Si sobre aquest *buffer* executem `getInt()` ens retornarà el valor 4 i mourà el punter intern fins a la següent dada (figura 2.7).

Figura 2.7. Resultat d'executar getInt()



En canvi, si executem `getInt(0)`, ens retornarà el valor 8, però el punter intern no es mourà. De manera que, si seguidament tornem a executar `getInt()`, obtindrem el valor 9.

Quelcom de semblant passa amb els mètodes *put* l'execució de `putDouble(3.5)`: escriuria 8 bytes a partir de la posició del punter intern i el desplaçaria al byte situat a la posició ubicada just després del darrer byte escrit. Per contra, `putDouble(4, 9.5)` escriuria 8 bytes començant en el 5è byte del *buffer*, però no mouria el punter intern.

Vegeu a continuació dos exemples equivalents d'escriptura relativa i absoluta respectivament:

```
1 ...
2 ByteBuffer byteBuffer = ByteBuffer.allocate(MIDA_BYTES);
3
```

```
...
for (int i = 0; i< (MIDA_BYTES/BYTES_PER_INT); i++){
    byteBuffer.putInt(i);
}

...
for (int i = 0; i< MIDA_BYTES; i+=BYTES_PER_INT){
    byteBuffer.putInt(i, i/BYTES_PER_INT);
}
...
```

2.3.5 Buffers de tipus específics

En cas de treballar amb un únic tipus de dada primitiu, és possible obtenir, a partir d'una instància de `ByteBuffer`, diversos *buffers* específics per a un tipus de dada concret. És a dir, `ByteBuffer` actuaria de `Factory` generant instàncies específiques per treballar amb dades de tipus `char`, `long`, `double`, etc.

```
ByteBuffer bufferOriginal = ByteBuffer.allocate(MIDA_MAXIMA);

...
LongBuffer bufferDeLongs = bufferOriginal.asLongBuffer();

...
CharBuffer bufferDeChars = bufferOriginal.asCharBuffer();

...
DoubleBuffer bufferDeDoubles = bufferOriginal.asDoubleBuffer();

...
```

El principal avantatge de treballar amb *buffers* específics consisteix en el fet que els mètodes d'assignació i obtenció de dades genèriques (*get* i *put*) són específics per al tipus de dada concreta de treball del *buffer*. Així, el mètode `get` d'un `LongBuffer` retornarà *longs*, mentre que el d'un `DoubleBuffer`retornarà *doubles*. De la mateixa manera, el mètode *put* d'un *ShortBuffer* acceptarà només *shorts*, i el d'un `FloatBuffer`, per contra, acceptarà només *floats*.

Malgrat que seria possible crear directament un *buffer* especific usant el mètode instanciador anomenat *allocate*, el més comú és obtenir les instàncies a partir d'un ByteBuffer. La raó és ben senzilla: totes les instàncies generades per un mateix *ByteBuffer* comparteixen les mateixes dades en la mateixa ubicació de memòria, fet que significa que qualsevol canvi en les dades d'algun d'ells repercuteix també en les dades de tots els altres.

Feu la prova. Instancieu tres *buffers*: un de tipus *ByteBuffer*, un segon de tipus *ShortBuffer* i el darrer de tipus *CharBuffer*. Implementeu un algoritme que escrigui, usant el *buffer* de tipus *short,*tots els valors compresos entre *32* i *127*. Seguidament, feu que l'algorisme recuperi els valor assignats, però fent servir el *buffer* de tipus *char*. Observareu que, malgrat que s'hagin entrat valors numèrics, recuperareu tots els caràcters compresos entre el codi *32* i el codi *127*.

Cal destacar que els *buffers* comparteixen només les dades, no pas els seus estats interns. És a dir, cada *buffer* manté de forma independent la seva posició, les seves marques internes, etc. Vegeu-ho en el següent exemple, una variant de la prova que us hem proposat on es van recuperant de forma independent les dades de cada *buffer*:

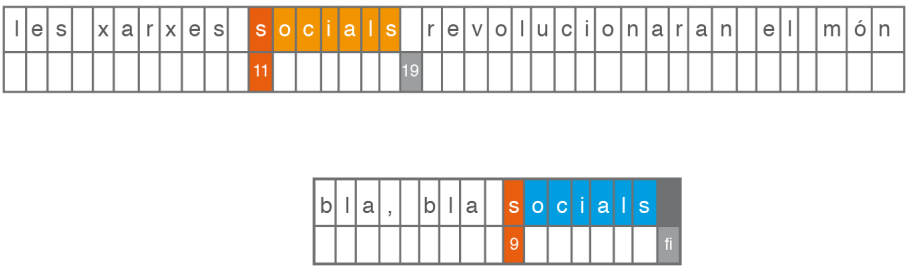
```
ByteBuffer buffer = ByteBuffer.allocate(500);
CharBuffer charBuffer = buffer.asCharBuffer();
ShortBuffer shortBuffer = buffer.asShortBuffer();

//clear, neteja el buffer de qualsevol dada que pogués haver-hi
shortBuffer.clear();

for(short value=32;value<128; value++){
    shortBuffer.put(value); //Assignem cada valor usant put
}

/* flip, marca la posició actual com el limit de dades entrades
 * i endarrereix la posició actual del buffer a la posició
 * inicial. En el nostre cas marcarà la posició 94 com a límit
 * de les dades i es situarà a la posició zero*/
shortBuffer.flip();

/* Com que es tracta de buffers independents cal indicar al
 * buffer de caràcters que el seu limit és el mateix que el del
 * shortBuffer. EL mètode limit, retorna la posició límit si no
 * se li passa cap paràmetre, però assigna com a nou límit, el
 * valor passat per paràmetre. Així, el valor retornat per
 * limit() del shortBuffer es assignat com a limit en el
 * charBuffer. */
charBuffer.limit(shortBuffer.limit());

System.out.println("Cars:");
int i=0;

/* Volem escriure matriu (de 6x32) de correspondència entre cada
 * caràcter i el seu codi, de manera que a les línies senars
 * aparegui el valor del codi i la les línies parells el
 * caràcter corresponent aliniat correctament.*/
while(i<charBuffer.limit()){
    for(int j=0; j<32; j++){
        //shortBuffer avança 32 posicions, però no pas charBuffer
        System.out.printf("%4d", shortBuffer.get());
    }
    System.out.println();
    for(int j=0; j<32; j++){
        //charBuffer avança 32 posicions, però no pas shortBuffer
        System.out.printf("%4c", charBuffer.get());
        i++;
    }
    System.out.println();
}
System.out.println();
```

2.3.6 Treball combinat de Buffers i Channels

Quan es fan treballar conjuntament *buffers* i *channels*, cal tenir en compte que cada un d'ells disposa d'un cursor o punter de posicionament propi i independent. El procés de lectura o escriptura intenta arribar, sempre que sigui possible, fins al final del *buffer*. Per exemple, mireu la figura 2.8 si la taula més gran (color

groc) representés un fitxer amb el seu punter posicionat al byte 11, i la taula més petita (color blau) representés un *buffer* amb el seu punter posicionat al byte 9, el procés de lectura del `Channel`, mètode `read`, aconseguiria escriure, en absència de bloquejos, la paraula "socials" en el *buffer* i omplir-lo totalment des de la posició 9 fins al final. La posició del `FileChannel` avançarà també fins al byte 19.

FIGURA 2.8. Treball combinat de Buffers i Channels



Cal tenir en compte que els bloquejos dels fitxers poden alterar aquest comportament. De fet, tant el procés de lectura com el d'escriptura s'interrompran en intentar operar en alguna zona bloquejada del fitxer. Per assegurar l'escriptura sencera del *buffer* caldria controlar que el procés arribi al final.

```
1  public void escriu(ByteBuffer buffer, FileChannel channel)
2                                                    throws IOException{
3      //Mentre no s'hagi escrit tot el contingut del buffer...
4      while(buffer.position()<buffer.limit()){
5          channel.write(buffer);
6      }
7  }
```

De forma semblant caldrà controlar també la lectura, però en aquest cas cal tenir en compte que la lectura parcial pot ser deguda al fet que s'ha arribat al final del fitxer. Aprofitarem el fet que el procés de lectura retorna el valor -1 quan s'intenta llegir més enllà del límit del fitxer.

```
1  public void llegeix(int mida, ByteBuffer buffer, FileChannel channel)
2      throws IOException{
3      int llegit = 0;
4      int totalLLegit=0;
5      //Mentre no s'hagi llegit la quantitat desitjada...
6      while(llegit!=-1 && totalLLegit<mida){
7          llegit = channel.read(buffer);
8          totalLLegit+=llegit;
9      }
10 }
```

La utilitat anterior llegirà el contingut del fitxer des de la posició del cursor del `Channel` fins que s'hagin llegit el nombre de bytes indicats pel paràmetre mida, o bé fins que s'hagi arribat al final del fitxer, cas en què la variable llegida prendrà el valor -1.

Cal tenir en compte que `FileChannel` facilita també una versió absoluta del procés de lectura i del d'escriptura. És a dir, una versió on s'ignora la posició del

cursor del Channel, ja que es començarà a operar a partir de la posició indicada per paràmetre.

```
public void escriu(long offsetFitxer, ByteBuffer buffer,
                        FileChannel channel) throws IOException{
    int escrit = 0;
    //Mentre no s'hagi escrit tot el contingut del buffer...
    while(escrit<buffer.limit()){
        escrit += channel.write(buffer, offsetFitxer+escrit);
    }
}

public void llegeix(long offsetFitxer, int mida, ByteBuffer buffer,
    FileChannel channel) throws IOException{
    int llegit = 0;
    int totalLLegit=0;
    //Mentre no s'hagi llegit la quantitat desitjada...
    while(llegit!=-1 && totalLLegit<mida){
        llegit = channel.read(buffer, offsetFitxer+totalLLegit);
        totalLLegit+=llegit;
    }
}
```

La diferència entre aquests dos mètodes i els dos anteriors és que, en el cas d'*escriu,* aquesta implementació emmagatzema el contingut del *buffer* a la posició indicada per *offsetFitxer* sense canviar la posició interna del cursor del `FileChannel`. En canvi, l'anterior emmagatzemava el contingut a la posició del cursor del `FileChannel` i en acabar, el cursor avançava la quantitat de bytes emmagatzemada.

Quelcom de semblant passa amb *llegeix*. La lectura en la darrera implementació s'inicia a la posició indicada per *offsetFitxer*, i, com és de suposar, el cursor del `FileChannel` no es mou de lloc.

2.3.7 Transferència directa

Encara hi ha una operació més del paquet *nio* que val la pena esmentar. Es tracta de la transferència directa de dades entre *Channels*. Són dues operacions equivalents anomenades `transferFromo` `transferTo`, que aprofiten al màxim la capacitat de còpia massiva dels sistemes operatius. Són operacions en què es realitza una connexió directa a través dels recursos específics del sistema operatiu. Aquests mètodes no fan servir *buffers* gestionats des de Java, sinó que es delega directament sobre les eines externes. Aquestes eines són ideals per realitzar còpies massives entre fitxers quan la màquina virtual corri en sistemes potents. En cas contrari, no se'n treu rendiment.

Acabarem de completar la classe `Utilitats` que hem anat implementant afegint un mètode de còpia massiva. Tant el mètode `transferTo`com `transfetFrom` són sensibles als fitxers bloquejats i poden interrompre la còpia sense haver acabat si es troben un bloqueig durant l'execució. Caldrà assegurar la còpia sencera del fitxer usant un bucle adequat. En el codi podeu veure aquesta instrucció ressaltada.

```java
public void copiaFitxersDeBytesNio(File origen, File desti)
    throws IOException {
    long size;
    long count = 0;
    FileChannel input = null;
    FileChannel output = null;
    try {
        //Obrim el canal origen
        input = new FileInputStream(origen).getChannel();
        //Obrim el canal destí
        output = new FileOutputStream(desti).getChannel();
        //Calculem la mida
        size = input.size();
        //Mentre no s'hagi copiat tot el fitxer...
        while (count < size) {
            count += output.transferFrom(input, 0, size - count);
        }
    } finally {
        //Tancar els canals
        intentarTancar(input);
        intentarTancar(output);
    }
}
```

2.4 Estudi pràctic de la funcionalitat d'entrades i sortides de dades bàsiques

En aquest apartat continuarem la implementació del gestor de fitxers. Ara afegirem al gestor la capacitat de moure o copiar fitxers d'una ubicació a una altra, així com la capacitat d'editar el contingut de fitxers de text. La sintaxi de la nova funcionalitat es troba descrita a la interfície `GestioFitxersPlus` de la biblioteca que se us proporciona amb el material del mòdul. Es tracta d'un complement de la interfície `GestioFitxers`. És a dir, que hereta d'ella, i per tant la classe que la implementi haurà d'implementar tots els mètodes de `GestioFitxersPlus`, a més dels de `GestioFitxers`.

Podem reutilitzar la classe `GestorFitxersImple`, que ja hem implementat. La millor forma d'aconseguir-ho és heretant, de manera que només hem de codificar els nous mètodes declarats a `GestioFitxersPlus`.

Començarem per la declaració i els atributs. Com que usarem algunes de les utilitats implementades a la classe `Utilitats`, hi declararem un atribut d'aquest tipus.

```java
public class GestioFitxersPlusImpl extends GestioFitxersImpl
    implements GestioFitxersPlus {
    private Utilitats utilitats=new Utilitats();
    ...
```

De forma semblant a la vegada anterior, deixeu que NetBeans generi l'esquelet de la classe clicant la bombeta del marge esquerre.

2.4.1 Còpia i trasllat de fitxers

Anem a implementar ara el mètode de trasllat de fitxers anomenat *moure*. La implementació consistirà en realitzar una còpia del fitxer origen a la ubicació de destí i a continuació eliminarem el fitxer origen. Per fer la còpia usarem la funció que hem implementat a la classe `Utilitats`.

Abans d'iniciar la còpia caldrà comprovar que el fitxer destí no existeixi (de forma que mai es pugui perdre el contingut de cap fitxer existent) i que el fitxer origen sigui accessible. Per tal de simplificar el codi del mètode crearem dues funcions privades que realitzaran aquesta feina. El mètode `TestejarExistencia` rep dos paràmetres: el fitxer a testejar i el *test* a realitzar. Un *test* amb valor *cert* significarà que és necessari que el fitxer existeixi per poder continuar, mentre que un *test* amb valor *fals* el que indicarà és que no hi hauria d'haver cap fitxer amb aquell nom.

El mètode `TestejarAccés`, de forma semblant, comprova que el fitxer passat per paràmetre existeix i es troba accessible per ser llegit.

Els controls es fan llançant una excepció en cas que no es passi el test. Com que les crides als controls es posen a l'inici del mètode, abans de començar cap acció específica, el llançament de l'excepció impedirà l'execució del codi restant: còpia i eliminació i actualització del nou contingut per tal de reflectir els canvis a la pantalla.

```
@Override
public void moure(File origen, File desti)
                                        throws GestioFitxersException{
    TestejarExistencia(desti, false);
    TestejarAcces(origen);
    try {
        utilitats.copiaFitxersDeBytesNio(origen, desti);
        origen.delete();
    } catch (IOException ex) {
        throw new GestioFitxersException("S'ha produit un error "
            + "d'entrada o sortida: '" + ex.getMessage() + "'", ex);
    }
    actualitza();
}

private void TestejarExistencia(File file, boolean test)
    throws GestioFitxersException{
    //es controla l'existència del fitxer d'acord amb el test
    if(file.exists()!=test){
        if(test){
            throw new GestioFitxersException("Error. el fitxer "
                + file.getAbsolutePath()+ " no existeix");
        }else{
            throw new GestioFitxersException("Error. el fitxer "
                + file.getAbsolutePath()+ " ja existeix");
        }
    }
}

private void TestejarAcces(File file) throws GestioFitxersException{
    //es comprova que existeixi el fitxer
    TestejarExistencia(file, true);
    //es controla que es tinguin permisos per llegir la carpeta
    if(!file.canRead()){
        throw new GestioFitxersException("Alerta. No es pot llegir "
```

```
            + file.getAbsolutePath() + ". No teniu prou permisos");
    }
}
```

Potser us preguntareu per què cal fer una còpia i una eliminació si reanomenant el fitxer podem aconseguir un efecte semblant. El problema és que aquesta afirmació només és certa si el trasllat es fa dins d'un mateix dispositiu, però si el trasllat implica diferents dispositius cal fer una còpia física del fitxer i una posterior eliminació de l'origen.

El mètode usat per copiar el contingut d'un fitxer (que hem anomenat *còpia*), té força semblança amb l'anterior (que havíem anomenat *moure*). Es diferencien en que la còpia no realitza el testeig per comprovar si el fitxer destí existeixi, sinó que es crea sempre. Per descomptat a la còpia no elimina mai el fitxer original.

```
@Override
public void copiar(File origen, File desti)
                                        throws GestioFitxersException{
    TestejarAcces(origen);
    crearSiCal(desti);
    try {
        utilitats.copiaFitxersDeBytesNio(origen, desti);
    } catch (IOException ex) {
        throw new GestioFitxersException("S'ha produit un error "
            + "d'entrada o sortida: '" + ex.getMessage() + "'", ex);
    }
    actualitza();
}

private void crearSiCal(File file) throws GestioFitxersException{
    if(!file.exists()){
        File pare = file.getParentFile();
        if(!pare.exists()){
            pare.mkdirs();
        }
        try {
            if(!file.createNewFile()){
               throw new GestioFitxersException("Error. No s'ha pogut"
                    + " crear " + file.getAbsolutePath() + ".");
            }
        } catch (IOException ex) {
            throw new GestioFitxersException("S'ha produit un error "
              + "d'entrada o sortida: '" + ex.getMessage() + "'", ex);
        }
    }
}
```

La utilitat `creaSiCal` assegura l'existència del fitxer, bé perquè ja hi era, bé perquè es crea en el moment de l'execució. Per tal que la creació es pugui dur a terme sense problemes,es comprova l'existència de la ruta, la qual també es crearia si fos necessari. Si, malgrat tot, la creació del fitxer no acabés amb èxit, es passaria a llançar una excepció per impedir que es continuï executant l'acció de còpia.

Seguint amb els requeriments de la interfície `GestioFitxersPlus`, implementarem dues utilitats més. La primera permet obtenir un objecte File corresponent al nom passat per paràmetre (el nom serà relatiu a la carpeta de treball del gestor). La segona utilitat permetrà obtenir un nom de fitxer, assegurant que es tracta d'un nom inexistent encara en el sistema. El nou nom tindrà com a prefix la cadena passada per paràmetre concatenada a un sufix numèric abans de l'extensió.

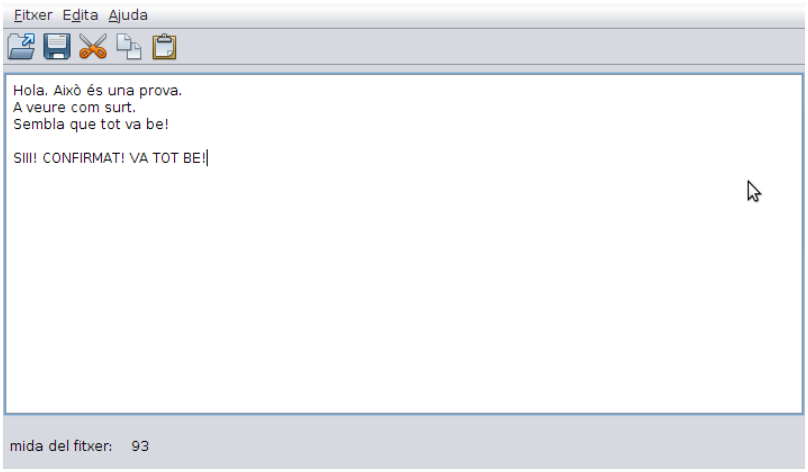
```java
@Override
public File getFile(String relativeName) {
    File ret = new File(carpetaDeTreball, relativeName);
    return ret;
}

@Override
public String getNouNom(String relativeName) {
    int cont=1;
    File file = new File(carpetaDeTreball, relativeName);
    String[] nom = new String[2];
    int indPunt = relativeName.lastIndexOf('.');
    nom[0] = relativeName.substring(0, indPunt);
    nom[1] = relativeName.substring(indPunt+1);
    while(file.exists()){
        if(nom[1].length()>0){
           file = new File(carpetaDeTreball, nom[0] + "("+ cont +")"
                + "." + nom[1]);
        }else{
           file = new File(carpetaDeTreball, nom[0] + "("+ cont +")");
        }
        cont++;
    }
    return file.getName();
}
```

2.4.2 Edició de fitxers de text

Podeu trobar la documentació de les interfícies i classes usades en aquest apartat a l'annex "Documentació API *GestioFitxersBase*. Paquet `ioc.dam.m6.exemples.gestiofitxers.contingut` de la secció "Annexos".

Per tal d'aconseguir editar els fitxers de text des del gestor de fitxers, caldrà disposar d'una interfície gràfica que contingui una àrea de text editable (figura 2.9). En aquest cas, també se us proporciona la interfície gràfica a la biblioteca del mòdul. Aquesta està basada en l'editor de text que l'empresa Sun Microsystems incorpora en els exemples de la versió estàndard del llenguatge Java. La interfície gràfica original ha estat modificada per adaptar-la a la nostra aplicació.

Figura 2.9. Interfície gràfica de l'editor de text que usarem en aquesta aplicació



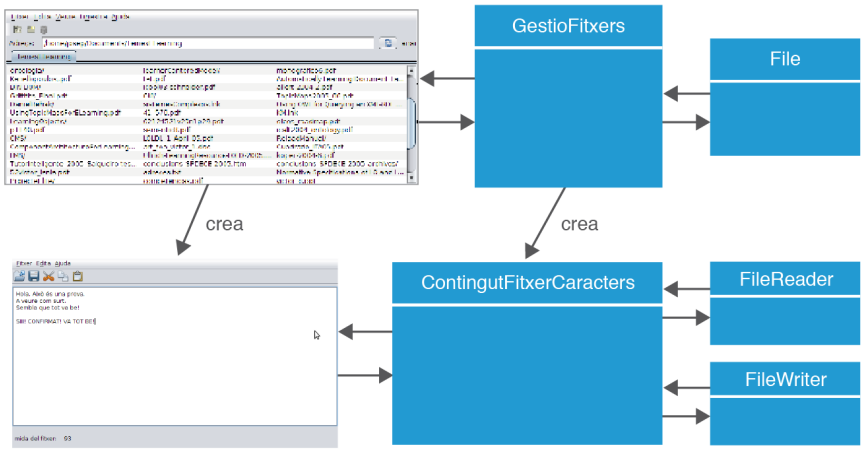
L'editor de text està preparat per treballar amb instàncies de la interfície anomenada `ContingutFitxerCaracters`, l'objectiu de la qual és fer d'intermediari entre el contingut real del fitxer i el text visible de la interfície gràfica. És a dir, fa d'*adaptador*.

El gestor de fitxers aconseguirà les instàncies de la interfície a partir del `GestioFitxersPlus`. Aquest serà l'encarregat de crear un objecte `ContingutFitxerCaracter` que gestioni el contingut del fitxer seleccionat. És a dir, farà de *Factory*.

La instància recent creada s'assignarà a la interfície gràfica d'edició i se li passarà el control (focus).

La interfície `ContingutFitxerCaracters` gestionarà el contingut dels fitxers com un tot (figura 2.10). És a dir, carregarà tot el contingut del fitxer en memòria o emmagatzemarà tot el contingut de la memòria al fitxer. Així doncs, la manipulació del text es farà sempre en memòria, i només el manipularà el fitxer en la càrrega o la gravació.

FIGURA 2.10. Relacions entre el gestor de fitxers i l'editor de fitxers de text



Com es pot veure, la finestra d'edició, la crearà la mateixa interfície gràfica de gestió de fitxers, mentre que GestioFitxers s'encarrega de crear les instàncies de ContingutFitxerCaracters.

Si consultem els mètodes de la interfície `ContingutFitxerCaracters` trobarem els següents:

- `void setFitxer (File fitxer) throws ContingutFitxerException`: associa al gestor de contingut el fitxer de text passat per paràmetre.
- `File getFitxer()`: obté el fitxer associat a aquest gestor de contingut.
- `void setText (String text) throws ContingutFitxerException`: assigna el text passat per paràmetre com a nou contingut del fitxer. El contingut romandrà en memòria fins que no s'executi el mètode salvar().
- `String getText() throws ContingutFitxerException`: obté el contingut existent en memòria corresponent al contingut del fitxer. Si el contingut no s'ha salvat podrien existir desfasaments entre ambdós.
- `void salvar() throws ContingutFitxerException`: guarda el contingut de la memòria al fitxer associat.

- `void carregar() throws ContingutFitxerException`: carrega en memòria el contingut del fitxer.
- `boolean estaSalvat()`: indica si el contingut de la memòria ja ha estat salvat i, per tant, es correspon amb el del fitxer.
- `boolean estaCarregat()`: indica si el fitxer ja ha estat carregat o si encara no s'ha executat aquesta acció.
- `long midaContingut() throws ContingutFitxerException`: obté la mida del contingut de la memòria.

Serà necessari un atribut per mantenir el fitxer associat, un per emmagatzemar el text en memòria, i dos per controlar les accions de carregar i salvar. A més, afegirem un instància de la classe `Utilitats` com a eina de suport.

```
public class ContingutFitxerCaractersImpl
                                    implements ContingutFitxerCaracters{
    private String text="";
    private boolean salvat = true;
    private boolean carregat = true;
    private File fitxer=null;
    private Utilitats utilitats = new Utilitats();
    ...
```

Implementem els constructors i els accessors...

```
public ContingutFitxerCaractersImpl() {
}

public ContingutFitxerCaractersImpl(File file) {
    setFitxer(file);
}

@Override
public void setText(String text) {
    this.text = text;
    //quan modifiquem al text perdem l'estatus de salvat.
    salvat=false;
}

@Override
public String getText() {
    return text;
}

@Override
public boolean estaSalvat() {
    return salvat;
}

@Override
public boolean estaCarregat() {
    return carregat;
}

@Override
public File getFitxer() {
    return fitxer;
}
```

Finalment, podem fer la implementació de l'operació de *carregar* i *salvar* usant la funcionalitat de la classe `Utilitats`. Usarem un `FileWrite` per salvar i un

`FileReader` per carregar. D'aquesta manera, usarem la codificació per defecte del sistema. Així, ens podem despreocupar de la codificació. Només caldrà tenir cura del llançament d'excepcions.

```
@Override
public void salvar() throws ContingutFitxerException {
    FileWriter writer = null;
    try {
        writer = new FileWriter(getFitxer());
        utilitats.copiarStringAWriter(text, writer);
    } catch (IOException ex) {
        throw new ContingutFitxerException(ex);
    } finally {
        utilitats.intentarTancar(writer);
    }
    salvat=true; //acabem de salvar i actualitzem l'estat
}

@Override
public void carregar() throws ContingutFitxerException {
    if(getFitxer().exists()){
        FileReader reader = null;
        try {
            reader = new FileReader(getFitxer());
            text = utilitats.copiarReaderAString(reader);
        } catch (IOException ex) {
            throw new ContingutFitxerException(ex);
        } finally {
            utilitats.intentarTancar(reader);
        }
    }else{
        text="";
    }
    carregat=true; //acabem de carregar i actualitzem l'estat
}
```

Si voleu provar l'aplicació cal implementar el mètode de `GestioFItxersPlusImpl` anomenat `instanciarContingutFitxerCaracters` per tal que, rebent un File, retorni una instància de`ContingutFitxerCaracters`.

```
@Override
public ContingutFitxerCaracters
                  instanciarContingutFitxerCaracters(File fitxer)
                                        throws GestioFitxersException{
    return new ContingutFitxerCaractersImpl(fitxer);
}
```

3. Persistència d'objectes en fitxers

A banda dels fitxers de text, sovint ens pot interessar emmagatzemar objectes sencers per poder-los recuperar en qualsevol moment. Bàsicament, disposem de tres maneres diferents d'aconseguir fer-los persistents. Podem seriar-los usant les eines específiques que Java té per convertir qualsevol objecte en una sèrie de bytes. Podem implementar algun format binari específic de conversió d'objectes a dades primitives que permeti l'emmagatzematge i la recuperació de les dades bàsiques. Finalment, podem convertir els objectes persistents en una jerarquia XML capaç d'emmagatzemar les dades bàsiques i les relacions establertes entre objectes.

3.1 Seriació d'objectes

La tècnica de la seriació és segurament la més senzilla de totes, però també a la vegada la més problemàtica. Java disposa d'un sistema genèric de seriació de qualsevol objecte, un sistema recursiu que es repeteix per cada objecte contingut a la instància que s'està seriant. Aquest procés s'atura en arribar als tipus primitius, els quals s'emmagatzemen com una sèrie de bytes. A banda dels tipus primitius, Java serialitza també força informació addicional o metadades específiques de cada classe (el nom de les classe, els noms dels atributs i molta més informació addicional). Gràcies a les metadades es fa possible automatitzar la seriació de forma genèrica amb garanties de recuperar un objecte tal com es va emmagatzemar.

Malauradament, aquest és un procediment específic de Java. És dir, no és possible recuperar els objectes seriats des de Java usant un altre llenguatge. D'altra banda, el fet d'emmagatzemar metadades pot arribar a comportar també problemes, tot i que usem sempre el llenguatge Java. La modificació d'una classe pot fer variar les seves metadades. Aquestes variacions poden donar problemes de recuperació d'instàncies que hagin estat guardades amb algunes versions anteriors a la modificació, impedint que l'objecte pugui ser recuperat.

Aquestes consideracions desestimen aquesta tècnica per emmagatzemar objectes de forma més o menys permanent. En canvi, la seva senzillesa la fa una perfecta candidata per a l'emmagatzematge temporal, per exemple dins la mateixa sessió.

Per tal que un objecte pugui ser seriat cal que la seva classe i tot el seu contingut implementin la interfície `Serializable`. Es tracta d'una interfície sense mètodes, perquè l'únic objectiu de la interfície és actuar de marcador per indicar a la màquina virtual quines classes es poden seriar i quines no.

Totes les classes equivalents als tipus bàsics ja implementen `Serializable`. També implementen aquesta interfície la classe `String` i tots els contenidors i

els objectes Array. Tot i això, la seriació de col·leccions depèn en darrer terme dels elements continguts. Si aquest són seriables, la col·lecció també ho serà.

En cas que la classe de l'objecte que s'intenti seriar, o les d'algun dels objectes que contingui, no implementessin la interfície `Serializable`, es llançaria una excepció de tipus `NotSerializableException`, impedint l'emmagatzematge.

L'Stream `ObjectInputStream` i `ObjectOutputStream` són decoradors que afegeixen a qualsevol altre Stream la capacitat de seriar qualsevol objecte `Serializable`. L'*stream* de sortida disposarà del mètode `writeObject`. L'*stream* d'entrada, en canvi, obtindrà l'objecte invocant el mètode de lectura `readObject`.

El mètode `readObject` només permet recuperar instàncies que siguin de la mateixa classe que la que es va emmagatzemar. En cas contrari, es llançaria una excepció de tipus `ClassCastExeception`. A més, cal que l'aplicació disposi del codi compilat de la classe; si no fos així, l'excepció llançada seria `ClassNotFoundException`.

3.1.1 Exemple d'implementació

Consulteu els annexos "Biblioteques de la unitat" i "Documentació API GestioFitxersBase". Hi trobareu el codi i la documentació del paquet `ioc.dam.m6.exemples.calculadora` en la secció "Annexos".

Vegem primer un exemple senzill. Suposem que desitgem guardar temporalment l'estat d'una calculadora representada per la classe `EstatCalculadora`. Aquesta classe tindrà un atribut enter representant la darrera operació demanada (*opreaciActiva*), un atribut de tipus *double* amb un valor del darrer número introduït (*valor*) i un altre, també *double,* on s'emmagatzemarà el resultat de les darreres operacions calculades (*resultat*).

```
1  public class EstatCalculadora implements Serializable, Cloneable{
2      double valor=0;
3      int operacioActiva=Constants.IGUAL;
4      double resultat=0;
5      ...
```

Podem implementar mètodes per emmagatzemar o recuperar una instància d'`EstatCalculadora` fent:

```
1  public void guardar(EstatCalculadora estat, String nomFitxer )
2                              throws EstatCalculadoraException{
3      try {
4          ObjectOutputStream out = new ObjectOutputStream(
5              new FileOutputStream(nomFitxer));
6          out.writeObject(estat);
7      } catch (IOException ex) {
8          throw new EstatCalculadoraException(ex);
9      }finally{
10         utilitats.intentarTancar(out);
11     }
12 }
13
14 public EstatCalculadora obtenir(String nomFitxer)
15                            throws EstatCalculadoraException{
16     EstatCalculadora estat = null;
17     try {
```

```
18          ObjectInputStream in = new ObjectInputStream(
19              new FileInputStream(nomFitxer));
20          estat = (EstatCalculadora) in.readObject();
21      } catch (ClassNotFoundException ex) {
22          throw new EstatCalculadoraException(ex);
23      } catch (IOException ex) {
24          throw new EstatCalculadoraException(ex);
25      }finally{
26          utilitats.intentarTancar(in);
27      }
28      return estat;
29  }
```

on `EstatCalculadoraException` seria una excepció especifica de l'aplicació.

El mètode `obtenir` recuperaria els diferents estats emmagatzemats en el mateix ordre en què els vàrem guardar. Imaginem, però, que volem implementar un sistema que permeti desfer les accions realitzades a la calculadora en l'ordre invers a com s'han realitzat. Normalment, això es fa amb una pila en memòria, però si preveiem que la pila pot créixer molt, ens pot interessar un cert suport persistent.

Per implementar aquest sistema en un fitxer caldrà emmagatzemar els diferents estats per on vagi passant la calculadora en un fitxer, un darrere l'altre, de forma semblant a com s'ha fet en el mètode *guardar*. Per recuperar-lo, caldrà anar a la darrera entrada del fitxer, recuperar l'objecte guardat i truncar el fitxer per tal de fer desaparèixer l'entrada recuperada.

La recuperació, però, ens planteja diversos problemes. En primer lloc, cal adonar-nos que no podem usar instàncies de `FileInputStream` per fer la lectura del darrer objecte, ja que necessitem avançar pel fitxer des del final cap a l'inici. És a dir, necessitem un `RandomAccesFile`. El segon problema que se'ns planteja és saber on comença el darrer objecte, o el que ve a ser el mateix, saber quants bytes ocupa l'objecte emmagatzemat. ObjectOutputStream no ens dóna aquesta informació, i necessitem algun mecanisme de càlcul. La solució és una mica enrevessada, però efectiva. Cal convertir un objecte a un vector de bytes i comptabilitzar la mida del vector.

La conversió la realitzarem escrivint l'objecte sobre un `ByteArrayOutputStrem` en lloc de fer-ho directament sobre el fitxer. Les instàncies de `ByteArrayOutputStrem` disposen d'un mètode anomenat `toByteArray` que retorna el seu *buffer* de bytes intern (un vector de bytes de la mida exacta que ocupen les dades escrites).

Els objectes `EstatCalculadora`seran tots de la mateixa mida, perquè només contenen atributs de tipus de dades primitives. No sempre passa això. Sovint els objectes contenen estructures dinàmiques que creixen en funció de les dades contingudes i fan que la seva mida no sigui fixa.

Sembla, doncs, un bona pràctica acompanyar la seriació dels objectes amb la seva pròpia mida per tal de poder navegar pel fitxer, d'objecte en objecte, quan faci falta. És el que farem a la nostra pila d'estats, emmagatzemarem la mida dels objectes al final per tal de saber quants bytes caldrà retrocedir per anar a l'inici de les dades de l'objecte.

Com que les instàncies de `RandomAccesFile`poden ser *Streams* d'entrada i sortida a la vegada, només crearem una única instància, que ens permetrà tant escriure com llegir en el fitxer.

Per gestionar el fitxer implementarem una classe que anomenarem `PilaEstatsPersistent`. Per defecte, aquesta classe usarà un fitxer anomenat *estats.stk*, però si fos necessari es podria passar una instància de `File` al constructor per usar un fitxer alternatiu.

Per assegurar que el fitxer es trobi obert cada cop que escrivim o llegim, obrirem el fitxer en el constructor i el mantindrem obert fins a la destrucció de l'objecte. Per aconseguir-ho, sobreescriurem el mètode `finalize`, des d'on farem el tancament. El mètode `finalize` es crida automàticament just abans que el *garbagecollector* destrueixi la instància de la memòria RAM.

Abans d'obrir el fitxer, assegurarem que aquest s'eliminarà en acabar l'execució gràcies a la invocació del mètode `deleteOnExit`. D'aquesta manera, cada execució treballarà sempre amb un fitxer nou.

```
1  public class PilaEstatsPersistent {
2      public static final String NOM_FITXER_PER_DEFECTE="estats.stk";
3      File file;
4      RandomAccessFile fin;
5      Utilitats utilitats = new Utilitats();
6
7      @Override
8      public void finalize() throws Throwable{
9          utilitats.intentarTancar(fin);
10         super.finalize();
11     }
12
13     public PilaEstatsPersistent() throws EstatCalculadoraException {
14         setFile(new File(NOM_FITXER_PER_DEFECTE));
15     }
16
17     public PilaEstatsPersistent(File file)
18                                 throws EstatCalculadoraException {
19         setFile(file);
20     }
21
22     private void setFile(File file) throws EstatCalculadoraException{
23         this.file=file;
24         try {
25             //assegurem que el fitxer s'eliminarà en acabar l'execució
26             //de l'aplicació on utilitzem aquesta pila.
27             this.file.deleteOnExit();
28             //Assegurem que abans d'obrir el fitxer existeixi
29             this.file.createNewFile();
30             //obrim el fitxer en mode lectura i escriptura
31             fin = new RandomAccessFile(file, "rw");
32         } catch (IOException ex) {
33             throw new EstatCalculadoraException(ex.getMessage(), ex);
34         }
35     }
36 ...
```

Per emmagatzemar cada estat al final del fitxer junt amb la mida implementarem el mètode `empilar`. Aquest escriu l'objecte en un `ByteArrayOutputStream` (via `ObjectOutputStream`), del qual obtindrem el vector de bytes que ens servirà per detectar la mida de l'objecte i per escriure els bytes al `RandomAccesFile`. Un cop emmagatzemat l'objecte, a continuació hi emmagatzemarem la mida.

```
public void empilar(EstatCalculadora estat)
                                    throws EstatCalculadoraException{
    try {
        ByteArrayOutputStream bOut=new ByteArrayOutputStream();
        ObjectOutputStream oOut = new ObjectOutputStream(bOut);
        oOut.writeObject(estat);
        byte[] bArray = bOut.toByteArray();
        //saltem al final del fitxer
        fin.seek(fin.length());
        //escrivim l'objecte a mode de vector de bytes.
        fin.write(bArray);
        //escrivim la mida de l'objecte just després d'aquest.
        fin.writeInt(bArray.length);
    } catch (IOException ex) {
        throw new EstatCalculadoraException(ex.getMessage(), ex);
    }
}
```

La lectura del darrer objecte emmagatzemat la realitzarem en el mètode desempilar, el qual, a més de llegir-lo, l'eliminarà del fitxer.

```
public EstatCalculadora desempilar() throws EstatCalculadoraException{
    EstatCalculadora estat = null;
    try {
        /* Saltem 4 bytes enrere del final del fitxer per poder llegir
         * la mida de l'objecte ja que un enter ocupa 4 bytes*/
        fin.seek(fin.length()- 4);
        //Llegim la mida de l'objecte.
        int length = fin.readInt();
        //Ens desplacem fins el començament de l'objecte a llegir
        fin.seek(file.length()- 4 -length);
        //Llegim l'objecte com una col·lecció de bytes.
        byte[] bArray = new byte[length];
        fin.readFully(bArray);
        /* Amb el vector de bytes, construïm un ByteArrayInputStream
         * encapsulat dins un ObjectInputStream.*/
        ObjectInputStream oIn = new ObjectInputStream(
                                new ByteArrayInputStream(bArray));
        //Instanciem l'objecte llegint-lo de l'Stream recent creat.
        estat = (EstatCalculadora) oIn.readObject();
        //Trunquem el fitxer per tal de fer desaparèixer l'objecte.
        fin.setLength(file.length()- 4 -length);
    } catch (IOException ex) {
        throw new EstatCalculadoraException(ex.getMessage(), ex);
    } catch (ClassNotFoundException ex) {
        throw new EstatCalculadoraException(ex.getMessage(), ex);
    }
    return estat;
}
```

Implementarem també el mètode `esBuit` per saber si el fitxer d'estats té algun estat per llegir o és buit. Per esbrinar-ho, farem servir el mètode `length` de la classe `RandomAccesFile`.

```
boolean esBuit() throws EstatCalculadoraException{
    try {
        return fin.length()==0;
    } catch (IOException ex) {
        throw new EstatCalculadoraException(ex.getMessage(), ex);
    }
}
```

Podeu comprovar el funcionament de `PilaEstatsPerssitent` amb el codi següent:

```java
public class ProvaPilaEstatPersistent {
    public static void main(String[] args) {
        try {
            PilaEstatsPersistent pila = new PilaEstatsPersistent();

            pila.empilar(new EstatCalculadora());
            pila.empilar(new EstatCalculadora(100, 1, 0));
            pila.empilar(new EstatCalculadora(10, 1, 100));
            pila.empilar(new EstatCalculadora(100, 1, 110));
            pila.empilar(new EstatCalculadora(0, 0, 210));

            while(!pila.esBuit()){
                EstatCalculadora estat = pila.desempilar();
                System.out.print("Valor: "
                                 + estat.getValorAcumulat());
                System.out.print(" Operacio: "
                                 + estat.getOperacioActiva());
                System.out.println(" Resultat:"
                                 + estat.getResultat());
            }
        } catch (EstatCalculadoraException ex) {
            ex.printStackTrace(System.err);
        }
    }
}
```

3.2 Fitxers amb formats binaris específics

Podeu trobar el model de classes necessàries per a aquesta implementació a la biblioteca `GestioFitxersBase`. Consulteu l'annex "Biblioteques del mòdul" i l'annex "Documentació API GestioFitxersBase", on trobareu informació sobre el paquet `ioc.dam.m6.exemples.dibuix`.

Si volem treballar amb fitxers que es puguin llegir des de diferents llenguatges de programació caldrà definir un format binari independent. La majoria de formats estàndards que coneixeu són independents del llenguatge i es poden llegir des d'aplicacions escrites en Java, C, Pascal, etc. Els fitxers JPG, PNG, PDF, VSD, AVI, MP3 i MPG són exemples de formats independents. Qualsevol aplicació escrita amb qualsevol llenguatge podrà llegir el format amb independència de l'aplicació que haguem utilitzat per escriure'l.

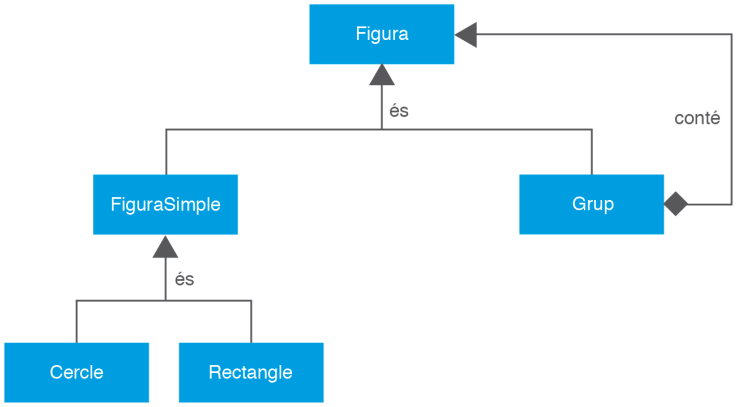
Es tracta d'una tècnica més laboriosa que l'anterior, perquè cal implementar mètodes específics que converteixin l'objecte a bytes agrupant-los d'acord amb el format, o a l'inrevés, mètodes que llegint els bytes en el format adequat, els converteixin en les respectives instàncies d'objectes que els han originat.

Imaginem que decidim inventar un format independent per aconseguir emmagatzemar dibuixos compostos de figures geomètriques. Per simplificar, només tindrem en compte dues figures geomètriques, el cercle i el rectangle. A més, les figures es poden agrupar creant composicions de diverses figures simples o agrupacions ja existents amb anterioritat, tal com podem observar a l'*esquema de la jerarquia de classes de les figures geomètriques* (figura 3.1). Totes les figures, les simples o les compostes, tenen un antecessor comú, la `Figura`. El `Cercle`i el `Rectangle`, a diferència del grup, són figures simples i deriven de la mateixa branca.

La classe `Figura` conté tres atributs: la posició, de tipus `Point` representant les coordenades x i y on s'haurà de dibuixar la figura, l'escala indicant l'ampliació o reducció que caldrà aplicar a la mida de la figura, i la rotació, mesurada en radians,

que és el gir que caldrà donar a la figura des de la seva posició original.

Figura 3.1. Jerarquia de classes per representar figures geomètriques d'un dibuix



La *FiguraSimple* afegeix dos atributs més, el color que tindrà la línia o contorn de la figura i el color interior de la mateixa. El `Cercle`, a més, afegeix un radi i el *Rectangle* una alçada i una amplada. El Grup conté una vector de figures. Al grup no se li pot definir cap color específic per a les seves figures, sinó que cada `FigureSimple`de la composició tindrà definit el seu propi color. En canvi, la posició, l'escala i la rotació sí que afecten totes les figures del grup, per això hereta directament de figura.

Totes les figures, a més, disposen d'un atribut de només lectura anomenat *tipusFigura* que indicarà el tipus que l'objecte representa (cercle, rectangle o grup). Es tracta d'un conjunt de valors enumerats amb quatre valors possibles (INDETERMINAT, CERCLE, RECTANGLE o GRUP). Tots els atributs de les classes disposen dels seus accessors de lectura i escriptura (get... i set...), excepte aquest darrer, que serà només un atribut de lectura.

```
1  public abstract class Figura {
2      private TipusFigura tipusFigura=TipusFigura.INDETERMINADA;
3      private Point posicio=null;
4      private int escala=1;
5      private float rotacio=0f;
6      ...
7  }
8
9  public abstract class FiguraSimple extends Figura {
10     private Color colorLinia;
11     private Color colorFigura;
12     ...
13 }
14
15 public class Cercle extends FiguraSimple {
16     private int radi=0;
17     ...
18 }
19
20 public class Rectangle extends FiguraSimple{
21     private int amplada;
22     private int alcada;
23     ...
24 }
25
26 public class Grup extends Figura{
```

```
    ArrayList<Figura> elements = new ArrayList<Figura>();
    ...
}
```

A continuació implementarem un sistema de conversió de dades per poder seriar les instàncies de les figures en dades primitives. Crearem la classe `CtrlPersistenciaFigura`, on definirem un conjunt de mètodes estàtics que ens ajudin a fer-ho. Cada classe de la jerarquia disposarà d'un mètode de seriació independent. D'aquesta manera, podrem anar ampliant la jerarquia sense gaires modificacions.

```
public class CtrlPersistenciaFigura {

...

}
```

Començarem per la classe `Figura`. Els mètodes `escriureFigura` i `llegirFigura` seqüencien només els atributs específics d'aquesta classe. Cal destacar que quan ens trobem amb atributs de tipus no primitius s'han de localitzar les equivalències adequades per poder fer la transformació en ambdós sentits. És el que passa amb l'atribut *posició*, que cal convertir-lo en una seqüència de dos enters (la coordenada *x* i la coordenada *y* del punt), o amb l'atribut *tipusFigura*, que en ser un tipus enumerat es pot convertir en un identificador enter (usant l'operació *ordinal*). Per realitzar la recuperació caldrà llegir exactament en el mateix ordre com es va emmagatzemar.

```
protected static void escriureFigura(Figura figura,
                                     DataOutputStream out)
                           throws PersistenciaDibuixException{
    try {
        out.writeInt(figura.getTipusFigura().ordinal());
        out.writeInt(figura.getPosicio().x);
        out.writeInt(figura.getPosicio().y);
        out.writeInt(figura.getEscala());
        out.writeFloat(figura.getRotacio());
    } catch (IOException ex) {
        throw new PersistenciaDibuixException(ex);
    }
}

protected static void llegirFigura(DataInputStream in, Figura figura)
                                   throws PersistenciaDibuixException{
    int x,y;
    try {
        x = in.readInt();
        y = in.readInt();
        figura.setPosicio(x, y);
        figura.setEscala(in.readInt());
        figura.setRotacio(in.readFloat());
    } catch (IOException ex) {
        throw new PersistenciaDibuixException(ex);
    }
}
```

Realitzarem les mateixes operacions amb els atributs de les instàncies que derivin de *FiguraSimple* fent el següent:

```
protected static void escriureFiguraSimple(FiguraSimple figura,
                                           DataOutputStream out)
```

```
                                throws PersistenciaDibuixException{
    try {
        //escriureFigura(figura, out);//
        out.writeInt(figura.getColorLinia().getRGB());
        out.writeInt(figura.getColorFigura().getRGB());
    } catch (IOException ex) {
        throw new PersistenciaDibuixException(ex);
    }
}

protected static void llegirFiguraSimple(DataInputStream in,
                                         FiguraSimple figura)
                              throws PersistenciaDibuixException{
    try {
        //llegirFigura(in, figura);//
        figura.setColorLinia(new Color(in.readInt()));
        figura.setColorFigura(new Color(in.readInt()));
    } catch (IOException ex) {
        throw new PersistenciaDibuixException(ex);
    }
}
```

Observeu la crida al mètode `escriureFigura/llegirFigura`. Fent aquestes crides assegurarem que qualsevol instància derivada de `FiguraSimple` seqüencií, abans dels atributs propis d'aquesta classe, els atributs propis de `Figura`.

Observeu també la seqüenciació dels atributs `Colorv` usant l'operació `getRGB`, que converteix qualsevol color en enter. Aquest valor servirà, durant la recuperació de la instància, per construir un color amb el mateix valor.

De forma semblant, anem implementant la seriació dels cercles i dels rectangles:

```
public static void escriure(Cercle figura, DataOutputStream out)
                            throws PersistenciaDibuixException{
    try {
        escriureFiguraSimple(figura, out);
        out.writeInt(figura.getRadi());
    } catch (IOException ex) {
        throw new PersistenciaDibuixException(ex);
    }
}

public static Cercle llegirCercle(DataInputStream in)
                        throws PersistenciaDibuixException{
    Cercle res=new Cercle();
    try {
        llegirFiguraSimple(in, res);
        res.setRadi(in.readInt());
    } catch (IOException ex) {
        throw new PersistenciaDibuixException(ex);
    }
    return res;
}

public static void escriure(Rectangle figura, DataOutputStream out)
                                throws PersistenciaDibuixException{
    try {
        escriureFiguraSimple(figura, out);
        out.writeInt(figura.getAmplada());
        out.writeInt(figura.getAlcada());
    } catch (IOException ex) {
        throw new PersistenciaDibuixException(ex);
    }
}

public static Rectangle llegirRectangle(DataInputStream in)
```

```
                            throws PersistenciaDibuixException{
    Rectangle res=new Rectangle();
    try {
        //llegirFiguraSimple(in, res);//
        res.setAmplada(in.readInt());
        res.setAlcada(in.readInt());
    } catch (IOException ex) {
        throw new PersistenciaDibuixException(ex);
    }
    return res;
}
```

També seriarem la classe `Grup`, de la qual destacarem que, per aconseguir una recuperació correcta, cal emmagatzemar tantes figures com componguin el grup. La seriació/deseriació de cada figura s'aconseguirà cridant la seva pròpia operació d'escriptura o lectura.

```
public static void escriure(Grup figura, DataOutputStream out)
                            throws PersistenciaDibuixException{
    try {
        int size= figura.size();
        escriureFigura(figura, out);
        out.writeInt(size); //quantitat de figures de l'agrupació
        for(int i=0; i<size; i++){
            escriure(figura.get(i), out);
        }
    } catch (IOException ex) {
        throw new PersistenciaDibuixException(ex);
    }
}

public static Grup llegirGrup(DataInputStream in)
                    throws PersistenciaDibuixException{
    Grup res=new Grup();
    try {
        llegirFigura(in, res);
        int size = in.readInt(); //Numero de figures a recuperar.
        for(int i=0; i<size; i++){
            res.add(llegir(in));
        }
    } catch (IOException ex) {
        throw new PersistenciaDibuixException(ex);
    }
    return res;
}
```

Finalment, dos mètodes genèrics ajuden a reconduir i trobar l'operació adequada per a cada figura:

```
public static Figura llegir(DataInputStream in)
                throws PersistenciaDibuixException{
    Figura res = null;
    try {
        int tipusFigura = in.readInt();
        if(tipusFigura == TipusFigura.CERCLE.ordinal()){
            res = llegirCercle(in);
        }else if(tipusFigura == TipusFigura.RECTANGLE.ordinal()){
            res = llegirRectangle(in);
        }else if(tipusFigura == TipusFigura.GRUP.ordinal()){
            res = llegirGrup(in);
        }
    } catch (IOException ex) {
        throw new PersistenciaDibuixException(ex);
    }
    return res;
}
```

```
18
19 public static void escriure(Figura figura, DataOutputStream out)
20                             throws PersistenciaDibuixException{
21     TipusFigura tipusFigura = figura.getTipusFigura();
22     if(tipusFigura == TipusFigura.CERCLE){
23         escriure((Cercle) figura, out);
24     }else if(tipusFigura == TipusFigura.RECTANGLE){
25         escriure((Rectangle) figura, out);
26     }else if(tipusFigura == TipusFigura.GRUP){
27         escriure((Grup) figura, out);
28     }
29 }
```

Abans de continuar, cal comentar que hauríem pogut crear els diferents mètodes d'escriptura i lectura en cada una de les classes pròpies del model. És a dir, hauríem pogut implementar un mètode per escriure a la classe `Figura`, un altre a la classe `FiguraSimple`, etc. El principal problema d'aquesta solució seria l'excessiva dependència entre el model (la jerarquia de figures) i el sistema d'emmagatzematge. Un model ha de ser independent de la forma com es decideixi emmagatzemar les seves instàncies. Per això, si es pot, és preferible independitzar ambdós sistemes creant una o diverses classes encarregades específicament de l'emmagatzematge.

Continuant amb l'exemple, per tal de poder acabar la implementació, necessitarem que el nostre model encapsuli les diferents figures en un sola instància que representi un dibuix. La classe `dibuix` contindrà la col·lecció de figures i permetrà realitzar un seguit d'accions sobre aquestes, com ara seleccionar figures, canviar-ne l'ordre de visualització, afegir noves figures al dibuix, etc.

Simplificarem aquestes accions fent que la nostra classe dibuix hereti d'ArrayList. Així:

```
1 public class Dibuix extends ArrayList<Figura>{
2     ...
3 }
```

Ara ja podem crear una classe que tingui com a funció associar un dibuix a un fitxer i sigui capaç d'emmagatzemar les figures que contingui o bé recuperar-lo i omplir el dibuix amb les figures emmagatzemades.

```
1  public class CtrlPersistenciaDibuix {
2      File file=null;
3      private Dibuix dibuix=null;
4      Utilitats utilitats=new Utilitats();
5
6      public CtrlPersistenciaDibuix() {
7          dibuix = new Dibuix();
8      }
9
10     public CtrlPersistenciaDibuix(Dibuix dibuix){
11         this.dibuix = dibuix;
12     }
13
14     public CtrlPersistenciaDibuix(Dibuix dibuix, File file){
15         this.file=file;
16         this.dibuix = dibuix;
17     }
18
19     public File getFile(){
20         return file;
```

```
    }

    public void setFile(File file) {
        this.file=file;
    }

    public Dibuix getDibuix() {
        return dibuix;
    }

    public void emmagatzemar() throws PersistenciaDibuixException{
        DataOutputStream fout=null;
        try {
            fout=new DataOutputStream(new FileOutputStream(file));
            for(Figura f: dibuix){
                CtrlPersistenciaFigura.escriure(f, fout);
            }
        } catch (FileNotFoundException ex) {
            throw new PersistenciaDibuixException(ex);
        }finally{
            utilitats.intentarTancar(fout);
        }
    }

    public void recuperar() throws PersistenciaDibuixException{
        DataInputStream fin=null;
        dibuix.clear();
        try {
            fin=new DataInputStream(new FileInputStream(file));
            while(fin.available()>0){
                dibuix.add(CtrlPersistenciaFigura.llegir(fin));
            }
        } catch (IOException ex) {
            throw new PersistenciaDibuixException(ex);
        }finally{
            utilitats.intentarTancar(fin);
        }
    }
}
```

Farem la verificació de la implementació amb una classe de prova:

```
public class ProvaFormatBinari {

    public static void main(String[] args) {
        File file = new File("prova");
        file.deleteOnExit();

        if(escriure(file).equals(llegir(file))){
            System.out.println("OK");
        }else{
            System.out.println("KO");
        }
    }

    private static Dibuix escriure(File file){
        Dibuix dibuix = new Dibuix();
        try {
            CtrlPersistenciaDibuix ctrlPersistenciaDibuix =
                new CtrlPersistenciaDibuix(dibuix, file);
            dibuix.add(new Rectangle(0, 5, 10, 10));
            dibuix.add(new Cercle(50, 32, 24));
            Grup grup = new Grup();
            grup.add(new Rectangle(1,1,100, 100));
            grup.add(new Rectangle(100, 100, 10, 10));
            grup.add(new Cercle(50, 55, 30));
            dibuix.add(grup);

            ctrlPersistenciaDibuix.emmagatzemar();
```

```
28          } catch (PersistenciaDibuixException ex) {
29              ex.printStackTrace(System.err);
30          }
31          return dibuix;
32      }
33
34      private static Dibuix llegir (File file){
35          Dibuix dibuix = new Dibuix();
36          try {
37              CtrlPersistenciaDibuix ctrlPersistenciaDibuix =
38                  new CtrlPersistenciaDibuix(dibuix, file);
39              ctrlPersistenciaDibuix.recuperar();
40          } catch (PersistenciaDibuixException ex) {
41              ex.printStackTrace(System.err);
42          }
43          return dibuix;
44      }
45  }
```

3.3 Fitxers amb formats XML

L'ús de formats binaris independents sobrepassa la barrera del llenguatge Java, permetent que altres llenguatges puguin llegir els fitxers que segueixin el format especificat. Tot i això, els formats binaris presenten encara certa incompatibilitat si els sistemes operatius sobre els quals es treballa fan servir diferents sistemes de representació de dades binàries. Com ja sabeu, no tots els sistemes representen les dades de la mateixa manera. N'hi ha que usen BCD per representar números, altres complement 2. Quan una dada ocupa més d'un byte, aquesta es pot llegir començant pel Byte menor (Little Endian) o bé pel major (Big Endian), etc.

És a dir, els formats binaris no garanteixen tampoc la compatibilitat de dades entre sistemes. Per això, quan es volen emmagatzemar dades que hagin de ser llegides per aplicacions executades en múltiples plataformes serà necessari recórrer a formats més estandarditzats, com ara els llenguatges de marques.

Els documents XML aconsegueixen estructurar la informació intercalant un seguit de marques anomenades etiquetes. En XML, les marques o etiquetes tenen certa similitud amb un contenidor d'informació. Així, una etiqueta pot contenir altres etiquetes o bé informació textual. D'aquesta manera, aconseguirem subdividir la informació estructurant-la de forma que pugui ser fàcilment interpretada.

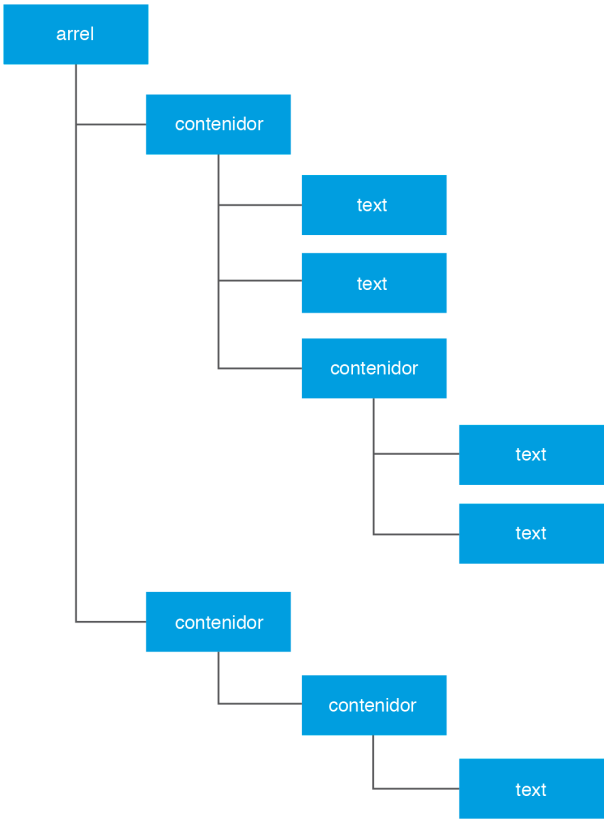
Com que tota la informació és textual, no existeix el problema de representar les dades de diferent manera. Qualsevol dada, ja sigui numèrica o booleana, caldrà transcriure-la en mode text, de manera que sigui quin sigui el sistema de representació de dades serà possible llegir i interpretar correctament la informació continguda en un fitxer XML.

És cert que els caràcters es poden escriure usant també diferents sistemes de codificació, però XML ofereix diverses tècniques per evitar que això sigui un problema. Per exemple, és possible incloure a la capçalera del fitxer quina codificació s'ha fet servir durant l'emmagatzematge, o també es poden escriure els caràcters de codi ASCII superior a 127, fent servir *entities de caràcter*, una

forma universal de codificar qualsevol símbol.

XML aconsegueix estructurar qualsevol tipus d'informació jeràrquica. Es pot establir certa similitud amb la forma com la informació s'emmagatzema en els objectes d'una aplicació i la forma com s'emmagatzemaria en un document XML. La informació, en les aplicacions orientades a objecte, s'estructura, agrupa i jerarquitza en classes, i en els documents XML s'estructura, organitza i jerarquitza en etiquetes contingudes unes dins les altres i atributs de les etiquetes (figura 3.2).

FIGURA 3.2. Les estructures arbòries són la forma més adient de representar informació jerarquitzada



Imaginem que volem representar les dades d'un dibuix com el de l'aparat anterior usant un format XML. No existeix una única solució, però cal que totes respectin la jerarquia del model. Sabem que totes les figures conformen un dibuix i que aquestes poden ser de tipus rectangles, cercles o grups de figures, i que cada una d'aquestes figures conté informació específica. Un possible format podria ser el següent:

```
<dibuix>
    <rectangle x='5' y='10' esc='1' rot='0.0' alc='100'
               amp='50' colli='0' colfig='325678'/>
    <grup x='10' y='25' esc='1' rot='0,0'>
        <cercle x='7' y='0' esc='1' ret='0.0' colli='0'
                colfig='325678' radi='50'/>
        <cercle x='50' y='50' esc='1' ret='0.0' colli='0'
                colfig='325678' radi='20'/>
    </grup>
</dibuix>
```

O també:

```
<dibuix>
    <rectangle>
        <figurasimple>
            <figura>
                <posicio>
                    <x>5</x>
                    <y>10</y>
                </posicio>
                <escala>1</escala>
                <rotacio>0.0</rotacio>
            </figura>
            <colorlinia>0</colorlinia>
            <colorfigura>325678</colorfigura>
        </figurasimple>
        <alcada>100</alcada>
        <amplada>50</amplada>
    </rectangle>
    <grup>
        <figura>
            <posicio>
                <x>10</x>
                <y>25</y>
            </posicio>
            <escala>1</escala>
            <rotacio>0.0</rotacio>
        </figura>
        <colleccio>
            <cercle>
                <figurasimple>
                    <figura>
                        <posicio>
                            <x>7</x>
                            <y>0</y>
                        </posicio>
                        <escala>1</escala>
                        <rotacio>0.0</rotacio>
                    </figura>
                    <colorlinia>0</colorlinia>
                    <colorfigura>325678</colorfigura>
                </figurasimple>
                <radi>50</radi>
            </cercle>
            <cercle>
                <figurasimple>
                    <figura>
                        <posicio>
                            <x>50</x>
                            <y>50</y>
                        </posicio>
                        <escala>1</escala>
                        <rotacio>0.0</rotacio>
                    </figura>
                    <colorlinia>0</colorlinia>
                    <colorfigura>325678</colorfigura>
                </figurasimple>
                <radi>20</radi>
            </cercle>
        </colleccio>
    </grup>
</dibuix>
```

3.4 'Parser' o analitzador XML

Un *Parser* XML és una classe que té per objectiu analitzar i classificar el contingut d'un arxiu XML extraient la informació continguda en cada una de les etiquetes i relacionant-la d'acord amb la seva posició dins la jerarquia.

3.4.1 Analitzadors seqüencials

Els analitzadors seqüencials, que permeten extreure el contingut a mida que es van descobrint les etiquetes d'obertura i tancament, s'anomenen analitzadors sintàctics. Són analitzadors molt ràpids, però presenten el problema que cada cop que es necessita accedir a una part del contingut cal rellegir tot el document de dalt a baix.

En Java, l'analitzador sintàctic més popular s'anomena SAX, que és l'acrònim de Simple API for XML. És una analitzador molt usat en diverses biblioteques de tractament de dades XML, però no sol usar-se en aplicacions finals.

Els **analitzadors sintàctics** són capaços d'aïllar les dades XML en una sola lectura seqüencial detectant les etiquetes d'obertura i tancament. Són molt ràpids, però han de llegir tot el document a cada consulta.

3.4.2 Analitzadors jeràrquics

Generalment, les aplicacions finals que els cal treballar amb dades XML solen usar analitzadors jeràrquics, perquè a més de realitzar una anàlisi seqüencial que els permet classificar el contingut, s'emmagatzemen a la memòria RAM seguint l'estructura jeràrquica detectada en el document. Això facilita molt les consultes que calgui repetir diverses vegades, atès que les estructures jeràrquiques de la memòria RAM tenen un rendiment d'accés parcial a les dades molt eficient.

Els **analitzadors jeràrquics** guarden totes les dades del'XML en memòria dins una estructura jeràrquica. Són ideals per a aplicacions que requereixin una consulta contínua de les dades.

El format de l'estructura on s'emmagatzema la informació a la memòria RAM ha estat especificat per l'organisme internacional W3C (World Wide Web Consortium) i s'acostuma a conèixer com a DOM (Document Object Model). És una estructura que HTML i javascript han popularitzat molt i es tracta d'una especificació que Java materialitza en forma d'interfícies. La principal s'anomena

Document i representa tot un document XML. En tractar-se d'una interfície, pot ser implementada per diverses classes.

Vegeu l'apartat "Fluxos eficients: Channels i Buffers", on es parla de les classes "factory".

L'estàndard W3C defineix l'especificació de la classe `DocumentBuilder` amb el propòsit de poder instanciar estructures DOM a partir d'un XML. La classe `DocumentBuilder` és una classe abstracta, i per tal que es pugui adaptar a les diferents plataformes, pot necessitar fonts de dades o requeriments diversos. Recordeu que les classes abstractes no es poden instanciar de forma directa. Per aquest motiu, el consorci W3 especifica també la classe `DocumentBuilderFactory`.

Les instruccions necessàries per llegir un fitxer XML i crear un objecte *Document* serien les següents:

```
1 DocumentBuilderFactory dbFactory =
2                          DocumentBuilderFactory.newInstance();
3 DocumentBuilder dBuilder = dbFactory.newDocumentBuilder();
4 Document doc = dBuilder.parse(new File("fitxer.xml"));
```

Podem reduir el nombre de variables concatenant instruccions.

```
1 Document doc = DocumentBuilderFactory.newInstance().newDocumentBuilder().parse(
      "fitxer.xml");
```

`DocumentBuilder`també instancia objectes `Document` buits que podran ser manipulats a posteriori.

```
1 DocumentBuilderFactory dbfac = DocumentBuilderFactory.newInstance();
2 DocumentBuilder docBuilder = dbfac.newDocumentBuilder();
3 Document doc = docBuilder.newDocument();
```

O bé,

```
1 Document doc = DocumentBuilderFactory.newInstance().newDocumentBuilder().
      newDocument();
```

L'escriptura de la informació continguda al DOM es pot seqüenciar en forma de text fent servir una altra utilitat de Java anomenada *Transformer*. Es tracta d'una utilitat que permet realitzar fàcilment conversions entre diferents representacions d'informació jeràrquica. És capaç, per exemple, de passar la informació continguda en un objecte Document a un fitxer de text en format XML. També seria capaç de fer l'operació inversa, però el mateix *DocumentBuilder* ja s'encarrega d'això.

Transformer és també una classe abstracta i requereix d'una *factory* per poder ser instanciada. La classe `Transformer` pot treballar amb multitud de contenidors d'informació perquè en realitat treballa amb un parell de tipus adaptadors (classes que fan compatibles jerarquies diferents) que s'anomenen `Source`i `Result`. Les classes que implementin aquestes interfícies s'encarregaran de fer compatible un tipus de contenidor específic al requeriment de la classe `Transformer`. Així, disposem de les classes DOMSource, SAXSource o StreamSorce com a adaptadors del contenidor de la font d'informació (DOM, SAX o Stream respectivament). DOMResult, SAXResult o StreamResult són els adaptadors equivalents del contenidor destí.

El codi bàsic per realitzar una transformació de DOM a fitxer de text XML seria el següent:

```
//Creació d'una instancia Transformer
Transformer trans = TransformerFactory.newInstance().newTransformer();

//Creació dels adaptadors Source i Results a partir d'un Document
//i un File.
StreamResult result = new StreamResult(file);
DOMSource source = new DOMSource(doc);
trans.transform(source, result);
```

Per tal de rebaixar la complexitat, crearem una classe que anomenarem `XmlCtrlDom`amb utilitats genèriques que ens simplifiquin el traspàs de fitxers XML a DOM o viceversa.

```
public class XmlCtrlDom {
    public static Document instanciarDocument()
        throws ParserConfigurationException{
        Document doc=null;
        doc = DocumentBuilderFactory.newInstance().newDocumentBuilder().
            newDocument();
        return doc;
    }

    public static void escriureDocumentATextXml(Document doc, File file)
      throws TransformerException {
        Transformer trans = TransformerFactory.newInstance().newTransformer();
        trans.setOutputProperty(OutputKeys.INDENT, "yes");

        StreamResult result = new StreamResult(file);
        DOMSource source = new DOMSource(doc);
        trans.transform(source, result);
      }

    public static Document instanciarDocument(File fXmlFile)
                               throws ParserConfigurationException,
                                      SAXException,
                                      IOException{
        Document doc=null;
        doc = DocumentBuilderFactory.newInstance().newDocumentBuilder().parse(
            fXmlFile);
        doc.getDocumentElement().normalize();
        return doc;
    }

    ...
}
```

L'estructura DOM

L'estructura DOM pren la forma d'un arbre, on cada part del'XML s'hi trobarà representada en forma de node. En funció de la posició en el document XML, parlarem de diferents tipus de nodes. El node principal que representa tot l'XML sencer s'anomena **document**, i les diverses etiquetes, inclosa l'etiqueta arrel, es coneixen com a nodes **element**. El contingut textual d'una etiqueta s'instancia com a node de tipus `TextElement` i els atributs com a nodes de tipus `Atribute`. Cada node específic disposa de mètodes per accedir a les seves dades concretes (nom, valor, nodes fills, node pare, etc.).

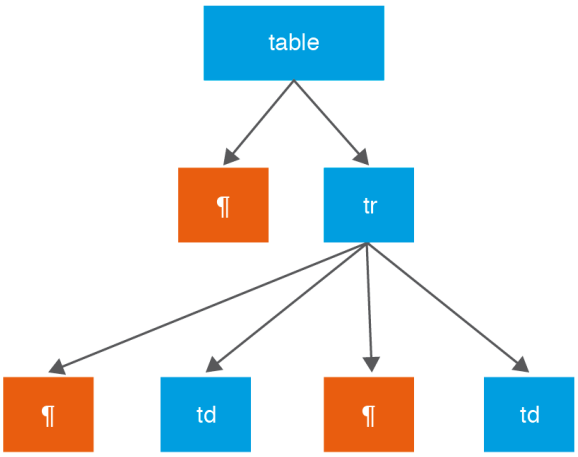
El DOM resultant obtingut des d'un XML acaba sent un còpia exacta del fitxer, però disposat de diferent manera. Tant al'XML com al DOM hi haurà informació no visible, com ara els retorns de carro, que cal tenir en compte per tal de saber processar correctament el contingut i evitar sorpreses poc comprensibles.

Imaginem que disposem d'un document XML amb el següent contingut:

```
<table>
    <tr>
        <td></td>
        <td></td>
    </tr>
</table>
```

A la figura 3.3 es mostra la representació que l'objecte DOM tindria, un cop copiat en memòria. Cal destacar que l'element *table* tindrà dos fills. En un s'hi guardarà el retorn de carro que situa l'etiqueta *tr* a la següent línia, a l'altre hi trobarem l'etiqueta *tr*. El mateix passa amb els fills de *tr*, abans de cada node *td* trobarem un retorn de carro.

Figura 3.3. Representació d'un objecte DOM amb retorns de carro

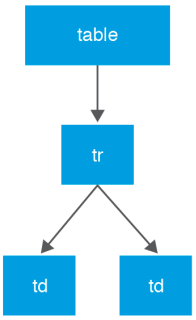


En canvi, si haguéssim partit d'un XML equivalent però sense retorns de carro, el resultat hauria estat també diferent.

```
<table><tr><td></td><td></td></tr></table>
```

El document XML anterior, sense retorns de carro, donaria la representació de l'objecte DOM que podeu observar en la figura 3.4.

Figura 3.4. Representació d'un objecte DOM sense retorns de carro



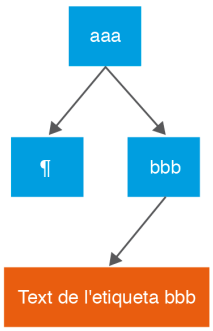
L'absència de retorns de carro en el fitxer implica també l'absència de nodes contenint els retorns de carro en l'estructura DOM.

Un altre aspecte a tenir en compte sobre el mapatge de fitxers XML és que el contingut textual de les etiquetes es plasma en el DOM com un node fill de l'etiqueta contenidora. És a dir, per obtenir el text d'una etiqueta cal obtenir el primer fill d'aquesta.

```
<aaa>
    <bbb>
        text de l'etiqueta bbb
    </bbb>
</aaa>
```

La figura 3.5 il·lustra la representació DOM d'una etiqueta que contingui text.

Figura 3.5. Representació DOM d'una etiqueta amb text



La interfície `Document` contempla un conjunt de mètodes per seleccionar diferents parts de l'arbre a partir del nom de l'etiqueta o d'un atribut identificador. Les parts de l'arbre es retornen com a objectes `Element`, els quals representen un node i tots els seus fills. D'aquesta manera, podrem anar explorant parts de l'arbre sense necessitat d'haver de passar per tots el nodes.

Per tal de facilitar l'obtenció del contingut d'un `Element` ampliarem les utilitats de la classe `XmlCtrlDom` afegint dos mètodes més.

```
public static String getValorEtiqueta(String etiqueta,
                                       Element element) {
    Node nValue = element.getElementsByTagName(etiqueta).item(0);
    return nValue.getChildNodes().item(0).getNodeValue();
}

public static Element getElementEtiqueta(String etiqueta,
                                         Element element) {
    return (Element) element.getElementsByTagName(etiqueta).item(0);
}
```

El primer rep el nom de l'etiqueta i l'element (o node parcial de l'arbre) a partir del qual es desitja fer la cerca. Fent servir el mètode `getElementsByTagName`, aconseguirem tots els nodes que tinguin per nom el valor del paràmetre etiqueta. És a dir, ens retornarà una col·lecció de nodes. Si només existeix un únic node amb el nom del paràmetre, aquest ocuparà la primera posició de la llista. Per això hi accedim amb el mètode `item`, indicant que ens interessa el primer element (posició zero).

El segon mètode és molt semblant al primer, però en comptes de recuperar el text, obtindrem el node amb tots els fills que tingui. És útil per aplicar a nodes intermedis (no textuals).

Els objectes Element disposen de mètodes per afegir nous fills (`appendChild`) o assignar el valor a un atribut (`setAtributte`). També permeten la consulta del valor dels atributs (`getAttributte`) o la navegació pels nodes de l'arbre (`getParentNode`, per obtenir el pare; `getFirstChild/getLastChild`, per obtenir el primer/darrer fill, o `getNextSibling` per navegar de germà en germà).

L'objecte `Document` farà de *factory* per a qualsevol node del document. La creació de nous elements (etiquetes) implicarà l'ús de `createElement`. Si volem crear contingut textual caldrà cridar el mètode `createTextNode`, i si el que volem són comentaris cridarem `createComment`.

La creació de nodes no implica la ubicació d'aquest dins l'arbre. És a dir, a més de crear-los caldrà assignar-los a un pare usant `appendChild`.

3.4.3 Implementació d'exemple

L'exemple conductor que ens permetrà mostrar l'ús de la biblioteca DOM de Java serà també la implementació d'una eina que gestioni l'emmagatzematge d'un dibuix de figures geomètriques, podeu comparar aquesta implementació amb la realitzada a l'apartat "Fitxers amb formats binaris específics".

El format XML que representarà el dibuix serà el següent: l'element arrel serà *dibuix*, el qual contindrà una llista de figures etiquetades segons el seu tipus com *rectangles*, *cercles* o *grups*.

```
dibuix ((rectangle | cercle | grup)*)
```

Els rectangles es definiran per un element de tipus *figurasimple*, per *l'alçada* i per *l'amplada*:

```
rectangle (figurasimple, alcada, amplada)
```

Els cercles, en canvi, contindran també un element de tipus *figurasimple*, però a continuació només disposaran de l'element *radi*.

```
cercle (figurasimple, radi)
```

El grup contindrà només dos elements, un de tipus *figura* i una altre de tipus *colleccio*.

```
cercle (figura, colleccio)
```

La *figurasimple* serà també un element compost, mentre que *alcada* i *amplada* seran elements de tipus text contenint els valors numèrics corresponents al rectangle representat.

L'element *figurasimple* es compondrà de dos elements simples (textuals) anomenats *colorlinia* i *colorfigura.*

```
figurasimple (colorlinia, colorfigura)
```

Els elements que composin *figura* seran *posicio*, *escala* i *rotacio*. A la vegada, *posicio* estarà compost per dos elements textuals que etiquetarem amb *x* i *y*. En canvi, *escala* i *rotació* seran també elements textuals.

```
figura (posicio, escala, rotacio)
posicio (x, y)
```

Finalment, l'element *colleccio* dins l'etiqueta *grup* serà un element compost d'una llista de figures de forma semblant a com es troben definides a l'element *dibuix*.

```
colleccio ((rectangle | cercle | grup)*)
```

El procés de transferència d'informació entre objectes i document XML serà similar a la forma com hem implementat el format binari, una classe que disposi d'un conjunt de mètodes, cada un d'ells especialitzats en la conversió d'una de les classes de la jerarquia de les figures.

En aquest cas, la classe que anomenarem `XmlCtrlFigura` heretarà de `XmlCtrlDom` per poder reutilitzar les utilitats ja implementades.

```
public class XmlCtrlFigura extends XmlCtrlDom{
    //Constants amb els noms de les etiquetes
    static final String ET_POSICIO="posicio";
    ...
    static final String ET_RADI="radi";
```

Començarem implementant l'escriptura de la classe `Figura`.

```
protected static void escriureFigura(Figura figura,
                                     Document doc,
                                     Element elemFigura){
        Element elmPosicio = doc.createElement(ET_POSICIO);
        Element nouElement = doc.createElement(ET_X);
        elmPosicio.appendChild(nouElement);
        nouElement.appendChild(doc.createTextNode(String.
            valueOf(figura.getPosicio().x)));
        nouElement = doc.createElement(ET_Y);
        elmPosicio.appendChild(nouElement);
        nouElement.appendChild(doc.createTextNode(String.
            valueOf(figura.getPosicio().y)));
        elemFigura.appendChild(elmPosicio);
        nouElement = doc.createElement(ET_ESCALA);
        nouElement.appendChild(doc.createTextNode(String.
            valueOf(figura.getEscala())));
        elemFigura.appendChild(nouElement);
        nouElement = doc.createElement(ET_ROTACIO);
        nouElement.appendChild(doc.createTextNode(String.
            valueOf(figura.getRotacio())));
        elemFigura.appendChild(nouElement);
    }
```

Del codi anterior destaquem que, per crear un element, necessitem un objecte `Document` que l'instanciï. Per això caldrà anar passant-lo als diferents mètodes que ho necessitin. L'element es crea invocant `createElement`i passant per paràmetre el nom de l'etiqueta XML associada. Si l'element és de text, cal crear-lo com un node textual fent servir `createTextNode` i passant per paràmetre el contingut textual. Els elements creats s'han d'afegir a l'estructura DOM com a fills de l'element que l'hagi de contenir (el qual haurem passat també per paràmetre juntament amb el *Document*). Usarem `appendChild`.

Els mètodes que traspassin informació de l'XML als objectes tindran la forma següent:

```
protected static void llegirFigura(Element elemFigura,
                                   Figura figura){
        int x = Integer.parseInt(getValorEtiqueta(ET_X, elemFigura));
        int y = Integer.parseInt(getValorEtiqueta(ET_Y, elemFigura));
        figura.setPosicio(x, y);
        int valorI = Integer.parseInt(getValorEtiqueta(ET_ESCALA,
                        elemFigura));
        figura.setEscala(valorI);
        float valorF = Float.parseFloat(getValorEtiqueta(ET_ROTACIO,
                        elemFigura));
        figura.setRotacio(valorF);
    }
```

Usarem les operacions implementades de `getValorEtiqueta` i de `getElementEtiqueta`, segons el cas, quan esperem un valor o un node amb fills respectivament.

De forma semblant, implementarem `escriureFiguraSimple`i `llegirFigurasSimple`:

```
protected static void escriureFiguraSimple(FiguraSimple figura,
      Document doc, Element elementFiguraSimple){
      Element elemFigura = doc.createElement(ET_FIGURA);
      escriureFigura(figura, doc, elemFigura);
      elementFiguraSimple.appendChild(elemFigura);
      Element nouElement = doc.createElement(ET_COLOR_LIN);
```

```
      nouElement.appendChild(doc.createTextNode(String.valueOf(figura.
          getColorLinia().getRGB())));
      elemFigura.appendChild(nouElement);
      nouElement = doc.createElement(ET_COLOR_FIG);
      nouElement.appendChild(doc.createTextNode(String.valueOf(figura.
          getColorFigura().getRGB())));
      elemFigura.appendChild(nouElement);
    }

    protected static void llegirFiguraSimple(Element elemFiguraSimple,
                                             FiguraSimple figura){
        Element elemFigura = getElementEtiqueta(ET_FIGURA,
                                                elemFiguraSimple);
        llegirFigura(elemFigura, figura);
        int valorI = Integer.parseInt(getValorEtiqueta(ET_COLOR_LIN,
                                                elemFiguraSimple));
        figura.setColorLinia(new Color(valorI));
        valorI = Integer.parseInt(getValorEtiqueta(ET_COLOR_FIG,
                                                elemFiguraSimple));
        figura.setColorFigura(new Color(valorI));
    }
```

L'escriptura i lectura dels objectes `Cercle`, `Rectangle` o `Grup` segueixen un patró semblant i els deixarem com a exercici a implementar, a excepció de la lectura del grup, ja que voldria parar atenció al tractament que cal donar a un node contenidor d'una llista d'elements dels quals en desconeixem el tipus. És el cas de l'etiqueta *colleccio* de l'element *grup* que conté totes les figures associades, però a priori desconeixem si es tracta de rectangles, cercles o altres grups.

Per analitzar la col·lecció continguda caldrà, en primer lloc, obtenir l'element (via `getElementEtiqueta`). Seguidament, recollirem tots els seus fills amb el mètode `getChildNodes`. Aquest mètode retorna un NodeList o col·lecció de tots els nodes fill. Recorrerem tots els fills usant un bucle. El que volem fer és recollir cada un dels fills (un cercle, un rectangle o un altre grup i fer la crida corresponent per crear la figura que representi). El problema és que no podem estar segurs que tots els nodes siguin figures. Recordeu que segons com estigui escrit l'XML, podem trobar intercalats nodes que continguin retorns de carro i que no ens interessa tractar. Per distingir els nodes tractables dels que no ho són, usarem el mètode `getNodeType`. Tots els nodes d'un DOM estan marcats amb un valor numèric que indica el tipus de node i el que conté. Ens interessa que es tracti d'un node de tipus Node.ELEMENT_NODE. És a dir, un node que conté altres nodes (ja que es tracta de figures).

```
public static Grup llegirGrup(Element element){
        Grup res=new Grup();
        Element elemFigura = getElementEtiqueta(ET_FIGURA, element);
        llegirFigura(elemFigura, res);
        Element elemColleccio = getElementEtiqueta(ET_COLLECCIO,
                                                   element);
        Noderetre_us_elsList nList = elemColleccio.getChildNodes();
        for(int i=0; i<nList.getLength(); i++){
            if(nList.item(i).getNodeType()==Node.ELEMENT_NODE){
                res.add(llegir((Element) nList.item(i)));
            }
        }
        return res;
    }
```

Els dos darrers mètodes que segueixen aconseguiran seleccionar l'operació correcta segons el tipus de figura que s'estigui escrivint o construint.

```
public static Figura llegir(Element element){
        Figura res = null;
        if(element.getTagName().equalsIgnoreCase(ET_CERCLE)){
            res = llegirCercle(element);
        }else if(element.getTagName().equalsIgnoreCase(ET_RECTANGLE)){
            res = llegirRectangle(element);
        }else if(element.getTagName().equalsIgnoreCase(ET_GRUP)){
            res = llegirGrup(element);
        }
        return res;
    }

    public static void escriure(Figura figura,
                                Document out,
                                Element element){
        Element elemFigura;
        TipusFigura tipusFigura = figura.getTipusFigura();
        if(tipusFigura == TipusFigura.CERCLE){
            elemFigura = out.createElement(ET_CERCLE);
            escriureCercle((Cercle) figura, out, elemFigura);
            element.appendChild(elemFigura);
        }else if(tipusFigura == TipusFigura.RECTANGLE){
            elemFigura = out.createElement(ET_RECTANGLE);
            escriureRectangle((Rectangle) figura, out, elemFigura);
            element.appendChild(elemFigura);
        }else if(tipusFigura == TipusFigura.GRUP){
            elemFigura = out.createElement(ET_GRUP);
            escriureGrup((Grup) figura, out, elemFigura);
            element.appendChild(elemFigura);
        }
    }
```

Ara només quedaria implementar la classe `XmlCtrlDibuix` per vinclar un *dibuix* a un fitxer XML. A banda dels constructors i els accessors, caldrà un mètode que llegeixi i escrigui l'objecte *Dibuix* a una estructura DOM i, finalment, dos mètodes més que emmagatzemin el DOM en un fitxer XML o que recuperin el contingut del fitxer i el buidin en un DOM. Respectivament, els anomenarem `emmagatzemar` i `recuperar`.

```
public class XmlCtrlDibuix extends XmlCtrlDom{
    static final String ET_DIBUIX="dibuix";

    File file=null;
    private Dibuix dibuix=null;

    public XmlCtrlDibuix() {
        dibuix = new Dibuix();
    }

    public XmlCtrlDibuix(Dibuix dibuix){
        this.dibuix = dibuix;
    }

    public XmlCtrlDibuix(Dibuix dibuix, File file){
        this.file=file;
        this.dibuix = dibuix;
    }

    public File getFile(){
        return file;
    }

    public void setFile(File file) {
```

```
25          this.file=file;
26      }
27
28      public Dibuix getDibuix() {
29          return dibuix;
30      }
31
32      private void escriure(Document doc){
33          Element arrel = doc.createElement(ET_DIBUIX);
34          doc.appendChild(arrel);
35          for(Figura f: dibuix){
36              XmlCtrlFigura.escriure(f, doc, arrel);
37          }
38      }
39
40      private void llegir(Document doc){
41          Element arrel = doc.getDocumentElement();
42          NodeList nList = arrel.getChildNodes();
43          dibuix.clear();
44          for(int i=0; i<nList.getLength(); i++){
45              if(nList.item(i).getNodeType()==Node.ELEMENT_NODE){
46                  dibuix.add(XmlCtrlFigura.llegir(
47                                          (Element) nList.item(i)));
48              }
49          }
50      }
51
52      public void emmagatzemar() throws xmlDibuixException{
53          try {
54              Document doc=null;
55              doc=instanciarDocument();
56              escriure(doc);
57              escriureDocumentATextXml(doc, file);
58          } catch (ParserConfigurationException ex) {
59              throw new xmlDibuixException(ex);
60          } catch (TransformerException ex) {
61              throw new xmlDibuixException(ex);
62          }
63      }
64
65      public void recuperar() throws xmlDibuixException {
66          try {
67              Document doc=null;
68              doc=instanciarDocument(file);
69              llegir(doc);
70          } catch (ParserConfigurationException ex) {
71              throw new xmlDibuixException(ex);
72          } catch (SAXException ex) {
73              throw new xmlDibuixException(ex);
74          } catch (IOException ex) {
75              throw new xmlDibuixException(ex);
76          }
77       }
78  }
```

3.5 Binding

El *Binding* és una tècnica que consisteix a vincular classes Java amb formats específics d'emmagatzematge de manera automatitzada. Sovint un conjunt idèntic de classes pot donar lloc a múltiples esquemes XML. Això significa que normalment l'automatització de l'escriptura de dades en format XML, que en termes tècnics s'anomena *marshalling*, requereix de certa ajuda per tal de decidir quin format

s'automatitzarà d'entre els molts possibles. D'això se'n diu *mapar*, perquè ve a ser com si configuréssim un mapa on s'indiquen els vincles entre les classes i els seus atributs, i els elements i atributs XML.

En Java existeixen diverses biblioteques per gestionar el *binding*, com per exemple *JAXB*, *JiBX*, *XMLBinding*, etc. Des de la versió 6.0 s'ha incorporat en el JDK estàndard *JAXB,* unapotent biblioteca.

3.5.1 Configuració amb anotacions

JAXB utilitza *Anotacions* per aconseguir la informació extra necessària per *mapar* el *binding*. Les *Anotacions* són unes interfícies o classes de Java molt especials. Serveixen per associar informació i funcionalitat als objectes sense interferir en l'estructura del model de dades. Abans que apareguessin les *Anotacions* era necessari fer servir l'herència per poder afegir funcionalitat a una classe sense haver de codificar-la. L'herència, però, interfereix directament en el model de dades, ja que en Java no és possible l'herència múltiple. Si la classe `Rectangle`, per exemple, en correspondència al model de dades, cal que hereti de `FiguraSimple`, però a la vegada necessitem afegir a la classe rectangle una funcionalitat extra, si no fem servir *Anotacions* serà necessari escriure codi extra o fins i tot haver de modificar el model final de dades per aconseguir ambdós objectius.

Si, per contra, fem servir Anotacions, els objectes disposaran d'informació o de funcionalitat extra sense que el model de dades quedi modificat, ja que les *Anotacions* no són visibles des dels objectes, malgrat que sí són accessibles via reflexió. Les *Anotacions* poden associar-se a un paquet, a una classe, a un atribut o fins i tot a un paràmetre. Aquestes classes especials es declaren en el codi de l'aplicació anteposant el símbol @ al nom de l'*Anotació*. Quan el compilador de Java detecta una *Anotació* crea una instància i la injecta dins l'element estructural afectat (paquet, classe, mètode, atribut, etc.). Això fa que aquestes no apareguin com a atributs o mètodes propis de l'objecte, i per això diem que no interacciona amb el model de dades, però les aplicacions que ho necessitin poden obtenir la instància injectada i fer-la servir.

Les *Anotacions* poden declarar-se amb paràmetres o sense. En cas que tinguin paràmetres, aquests poden ser altres Anotacions, valors constants o valors literals, de manera que estiguin disponibles en temps de compilació (que és quan el compilador realitza la injecció).

Abans de començar a estudiar algunes de les *Anotacions* de JAXB, veurem que aquesta biblioteca és capaç també de generar, de forma automàtica, les classes necessàries per suportar la informació descrita per un Schema XML, el que venim anomenant model de l'aplicació. Les classes generades es creen amb les *Anotacions* necessàries per establir correctament els vincles entre els atributs de les classes i els elements XML.

3.5.2 Generació automàtica del model de dades

En XML, els *Schema* són documents XML que permeten definir l'estructura d'altres documents. L'estudi dels *Schema* cau per complet fora dels objectius d'aquest mòdul, però com que són una peça important en el *Binding* farem un petit repàs com a recordatori dels conceptes més importants. Els *Schema* permeten definir tipus de dades que es classifiquen en tipus de dades simples i tipus de dades complexes. Els tipus de dades simples són valors unitaris i no es poden descompondre perquè perdrien totalment el seu sentit. Generalment, es corresponen amb tipus de dades primitius, però s'amplien amb tipus específics com *dates*, *text* o classes que representin conceptualment el mateix que els valors primitius (classe `Integer`, `BigInteger`, `Float`, etc.).

Els tipus complexos es defineixen com una composició d'altres tipus (ja siguin simples o complexes), els quals aporten un sentit extra en ser tractats en conjunt. El símil per excel·lència en termes de Java són les classes, i per especificar un tipus complex es defineixen elements, cadascun dels quals representaria un dels tipus en què es descompondria el tipus complex que s'estigui definint. És a dir, podem establir una correspondència entre els elements d'un tipus complex i els atributs d'una classe. El problema és que, al mateix nivell que els elements, es poden definir també atributs XML amb una correspondència també directa amb atributs de les classes Java. Els elements XML que configurin un tipus complex es poden definir com una seqüència ordenada d'elements (*sequence*), com una tria opcional (*choice*) o bé com una combinació d'ambdues.

XMLTool4NetBeans

XMLTool4NetBeans es distribueix a la versió 7.1 de l'IDE. Si teniu una versió anterior caldrà que us l'actualitzeu usant l'eina pròpia de l'IDE o baixant-vos-el directament del portal de plugins de NetBeans (http://plugins.netbeans.org/PluginPortal/) i cercant pel nom, o accedint directament a la pàgina http://plugins.netbeans.org/plugin/40292/xmltools4netbeans. Podeu trobar informació d'ús a la pàgina de l'autor: http://blogs.oracle.com/geertjan/entry/xml_schema_editor_in_netbeans.

NetBeans disposa d'un *plugin*, anomenat **XMLTools4NetBeans**, on podem trobar un conjunt d'eines XML, entre les quals destaquem un editor d'esquemes molt potent.

El format XML que hem triat és un format còmode per fer el tractament de dades usant un DOM, ja que les dades dels diferents nivells de la jerarquia d'una figura queden totalment encapsulades en nodes independents. Aquest format, però, presenta certa dificultat d'interpretació als ulls humans en llegir el contingut XML generat, ja que les dades de figura o figura simple es tracten com a components interns de les figures específiques (`Rectangle`, `Cercle` o `Grup`), més que no pas tractar-se com a contenidors genèrics que continguin casos específics.

Aquí, en tractar-se d'un procediment automàtic de transferència de dades, podem deixar de banda l'elecció del tipus d'estructura en funció del tractament i centrar-nos més en la part conceptual. Dissenyarem un format senzill més fàcilment interpretable en què no cal plasmar la jerarquia. D'aquesta manera, independitzem el format XML de la implementació Java finalment realitzada. Així, si algun dia decidim reestructurar la jerarquia, podem continuar amb el mateix format.

Podeu fer servir el *plugin* XMLTools4NetBeans per obtenir un *Schema* que ens defineixi un format per emmagatzemar les figures del dibuix definides a l'apartat "Implementació d'un exemple" de la Secció "Parser o analitzador sintàctic".

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
3     targetNamespace="http://xml.netbeans.org/schema/dibuix"
4     xmlns:tns="http://xml.netbeans.org/schema/dibuix"
5     elementFormDefault="qualified">
6     <!--Tipus Punt per definir les coordenades
```

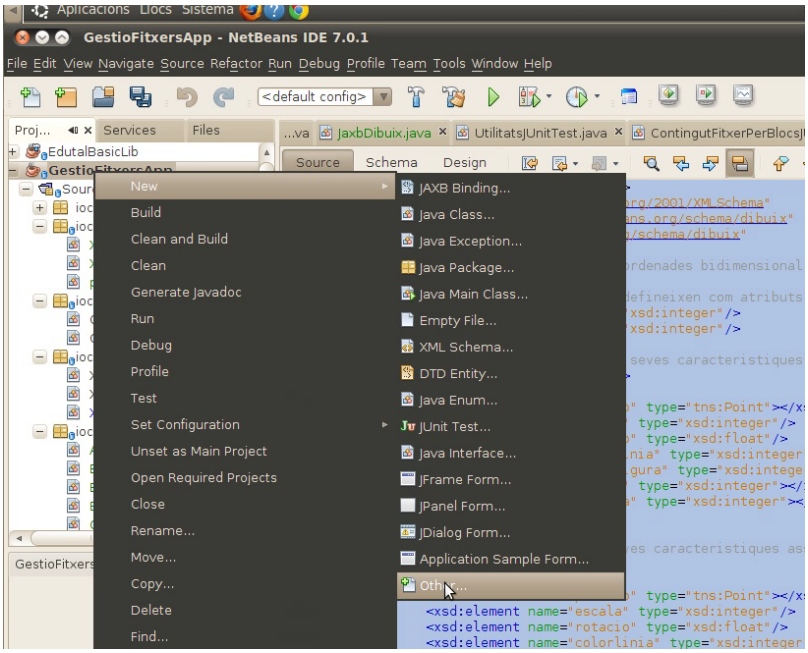
```
        bidimensional-->
    <xsd:complexType name="Point">
        <!--Les coordenades x i y és defineixen com atributs,
            no pas com elements-->
        <xsd:attribute name="x" type="xsd:integer"/>
        <xsd:attribute name="y" type="xsd:integer"/>
    </xsd:complexType>
    <!--Tipus Rectangle amb totes les seves caracteristiques
        associades-->
    <xsd:complexType name="Rectangle">
        <xsd:sequence>
            <xsd:element name="posicio" type="tns:Point"/>
            <xsd:element name="escala" type="xsd:integer"/>
            <xsd:element name="rotacio" type="xsd:float"/>
            <xsd:element name="colorlinia"
                                type="xsd:integer"/>
            <xsd:element name="colorfigura"
                                type="xsd:integer"/>
            <xsd:element name="alcada"
                                type="xsd:integer"/>
            <xsd:element name="amplada"
                                type="xsd:integer"/>
        </xsd:sequence>
    </xsd:complexType>
    <!--Tipus Cercle amb totes les seves caracteristiques
        associades-->
    <xsd:complexType name="Cercle">
        <xsd:sequence>
            <xsd:element name="posicio" type="tns:Point"/>
            <xsd:element name="escala" type="xsd:integer"/>
            <xsd:element name="rotacio" type="xsd:float"/>
            <xsd:element name="colorlinia"
                                type="xsd:integer"/>
            <xsd:element name="colorfigura"
                                type="xsd:integer"/>
            <xsd:element name="radi" type="xsd:integer"/>
        </xsd:sequence>
    </xsd:complexType>
    <!--Tipus que representarà una llista de figures. La
        llista podrà estar definida sense cap figura (buida)
        o be contenir un nombre indeterminat de figures-->
    <xsd:complexType name="LlistaFigures">
         <xsd:choice maxOccurs="unbounded" minOccurs="0">
             <xsd:element name="rectangle"
                                type="tns:Rectangle"/>
             <xsd:element name="cercle" type="tns:Cercle"/>
             <xsd:element name="grup" type="tns:Grup"/>
        </xsd:choice>
    </xsd:complexType>
    <!--Tipus Grup amb totes les seves caracteristiques
        associades-->
    <xsd:complexType name="Grup">
        <xsd:sequence>
            <xsd:element name="posicio" type="tns:Point"/>
            <xsd:element name="escala" type="xsd:integer"/>
            <xsd:element name="rotacio" type="xsd:float"/>
            <!--L'element col·lecció es defineix com una
                llista de figures-->
            <xsd:element name="colleccio"
                                type="tns:LlistaFigures"/>
        </xsd:sequence>
    </xsd:complexType>
    <!--L'element arrel cal definir-lo sense especificar
        l'atribut tipus per tal que JAXB el detecti com
        a tal. El tipus caldrà definir-lo com un element
        complex amb un únic component o la llista de
        figures-->
    <xsd:element name="dibuix">
        <xsd:complexType>
            <xsd:complexContent>
```

```
                <xsd:extension xmlns:tns=
                  "http://xml.netbeans.org/schema/dibuix"
                  base="tns:LlistaFigures"/>
            </xsd:complexContent>
        </xsd:complexType>
    </xsd:element>
</xsd:schema>
```

Un esquema com aquest validaria documents XML semblants a:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<dibuix xmlns="http://xml.netbeans.org/schema/dibuix">
    <rectangle>
        <posicio x="0" y="5"/>
        <escala>1</escala>
        <rotacio>0.0</rotacio>
        <colorLinia>-16777216</colorLinia>
        <colorFigura>-1</colorFigura>
        <alcada>10</alcada>
        <amplada>10</amplada>
    </rectangle>

    <cercle>
        <posicio x="50" y="32"/>
        <escala>1</escala>
        <rotacio>0.0</rotacio>
        <colorLinia>-16777216</colorLinia>
        <colorFigura>-1</colorFigura>
        <radi>24</radi>
    </cercle>

    <grup>
        <posicio x="0" y="0"/>
        <escala>1</escala>
        <rotacio>0.0</rotacio>
        <colleccio>
            <rectangle>
                <posicio x="1" y="1"/>
                <escala>1</escala>
                <rotacio>0.0</rotacio>
                <colorLinia>-16777216</colorLinia>
                <colorFigura>-1</colorFigura>
                <alcada>100</alcada>
                <amplada>100</amplada>
            </rectangle>
            <rectangle>
                <posicio x="100" y="100"/>
                <escala>1</escala>
                <rotacio>0.0</rotacio>
                <colorLinia>-16777216</colorLinia>
                <colorFigura>-1</colorFigura>
                <alcada>10</alcada>
                <amplada>10</amplada>
            </rectangle>
            <cercle>
                <posicio x="50" y="55"/>
                <escala>1</escala>
                <rotacio>0.0</rotacio>
                <colorLinia>-16777216</colorLinia>
                <colorFigura>-1</colorFigura>
                <radi>30</radi>
            </cercle>
        </colleccio>
    </grup>
</dibuix>
```

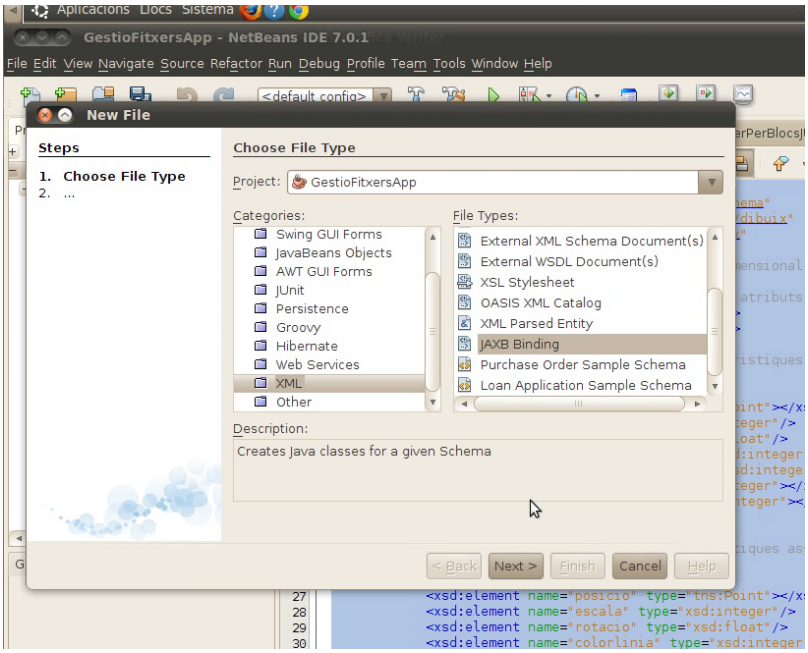
FIGURA 3.6. Menú per activar el Wizard de JAXB



JAXB és capaç de generar el model de dades automàticament (classes `Rectangle`, `Cercle`, `Grup` o `Dibuix`) realitzant un tractament anomenat compilació JAXB a partir de l'esquema definit més amunt. Aquest procés es pot realitzar usant comandes de consola o fent servir amb l'ajuda de l'IDE, *JAXBWizard* (figura 3.6). Sobre el projecte, cliqueu el botó dret del ratolí, seleccioneu el menú *New* i escolliu l'opció *Others*.

Això us obrirà un quadre de diàleg on caldrà que cerqueu l'opció *XML* a l'esquerra i el tipus *JAXB Binding* a la dreta (figura 3.7).

FIGURA 3.7. Quadre de diàleg per escollir la creació dels fitxers de configuració JAXB

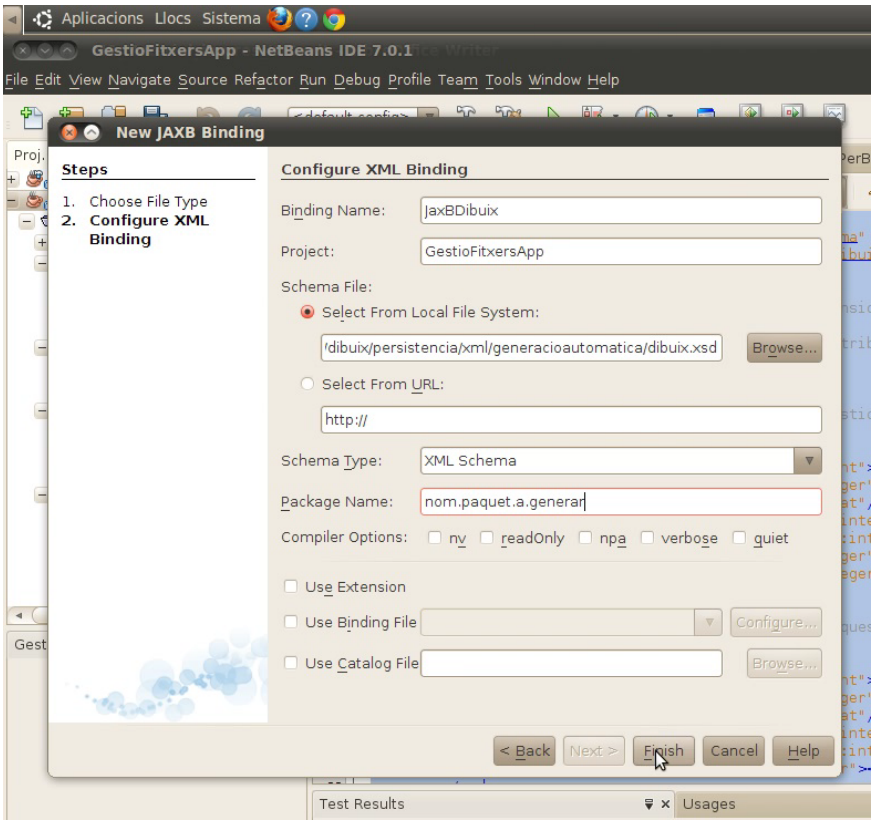


Cliqueu *Next* un cop hagueu fet la selecció correcta. D'aquesta manera, se us obrirà el darrer quadre de diàleg on caldrà que introduïu el nom de la configuració (ja que podeu generar-ne més d'una) i escolliu el fitxer esquema a partir del qual voleu fer la generació. Si cal, podeu navegar pel sistema de fitxers. Un cop escollit l'esquema, cal que introduïu el nom del paquet on desitgeu ubicar les classes generades, i finalment polseu *Finish*.

Aquest procés crearà les classes del model (les figures i el dibuix), més una classe anomenada `ObjectFactory`, la qual tindrà la funció d'instanciar els objectes de cada una de les classes del model durant el procés d'hidratació dels objectes o de traspàs de la informació des de l'XML. Si mireu el contingut, veureu també un altre fitxer anomenat *package-info.java*. No es tracta pas d'una classe, sinó d'un tipus de fitxer especial introduït a partir de la versió 5.0 de Java que permet recollir informació diversa sobre un paquet concret. Entre d'altres, admet una descripció textual sobre el propòsit general del fitxer, comentaris específics del paquet per afegir al *javadoc* i, el més important, les anotacions a nivell de paquet.

Per defecte, JAXB crearà una única *Anotació* a nivell de paquet especificant l'espai de noms que figuri al document *Schema* utilitzat per a la generació just abans de la sentència Java que identifica un paquet (figura 3.8).

FIGURA 3.8. Quadre de diàleg per configurar la generació automàtica de les classes del model a partir d'un Schema




```
1 @javax.xml.bind.annotation.XmlSchema(
2   namespace = "http://xml.netbeans.org/schema/dibuix",
3   elementFormDefault = javax.xml.bind.annotation.XmlNsForm.QUALIFIED)
4 package ioc.dam.m6.exemples.dibuix.persistencia.xml.generacioautomatica.schema;
```

Podeu observar el símbol @ com a element identificador de les *Anotacions*. Aquesta anotació, concretament, conté dos paràmetres, l'espai de noms de l'esquema i la indicació que les etiquetes descrites a l'*Schema* són qualificades i, per tant, pertanyents a l'espai de noms indicat.

La informació associada al paquet, junt amb la classe `ObjectFactory`, servirà a JAXB per crear el context de treball: una instància de *JAXContext* que usarà aquesta informació per crear uns objectes `Marshaller` o `UnMarshaller` específics per treballar amb el nostre model. La instrucció per crear el context és la següent:

```
1 JAXBContext context = JAXBContext.newInstance(nomPaquet);
```

L'objecte `Marshaller` l'instancia el context, fent:

```
1 Marshaller marshaller = context.createMarshaller();
```

Per realitzar transvasament d'informació del model al document XML serà necessari l'objecte del model corresponent a l'objecte contenidor principal de tota la informació (és a dir, l'equivalent al qual serà l'arrel del document XML) i un OutputStream associat al fitxer d'emmagatzematge.

```
1 marshaller.marshal(objecteRoot, fitxerOutputStream);
```

En el nostre model de figures, l'objecteRoot seria el dibuix.

La instància *UnMarshaller* també la crearà el context:

```
1 Unmarshaller unmarshaller = context.createUnmarshaller();
```

Per obtenir un objecte contenidor a partir de les dades emmagatzemades en un XML caldrà passar per paràmetre un `FileInputStream` del fitxer. El mètode `unmarshal`gestionarà tota la creació d'objectes necessaris per realitzar el transvasament d'informació cap al model. La instanciació dels objectes es realitzarà fent servir la classe `ObjectFactory`. El mètode `unmarshal`retornarà un objecte corresponent al contenidor principal i l'equivalent a l'arrel del document XML.

```
1 objectRoot = unmarshaller.unmarshal(fitxerInputStream);
```

Vigileu que les instàncies siguin realment de les classes generades per JAXB i no pas les usades a l'apartat "Implementació d'un exemple" de la Secció "Parser o analitzador sintàctic" i que formen part de la biblioteca del mòdul. Recordeu que, de moment, el vincle s'ha establert amb les classes generades de forma automàtica.

Podeu fer la prova instanciant un objecte Dibuix al qual li afegiu unes quantes figures.

3.5.3 Ús de JAXB amb un model de disseny propi

Normalment, el models de dades solen ser la peça clau de les aplicacions, i generarlos automàticament a partir d'un Schema no és, de ben segur, la millor manera d'optimitzar-lo. Es pot modificar l'Schema per aconseguir millors resultats, però a vegades és preferible plantejar les coses al revés. És a dir, partint de les classes, fem

les anotacions oportunes per aconseguir generar XML que validin un determinat Schema.

Segurament, la generació automàtica pot ser útil quan sigui necessari disposar ràpidament d'un model de classes Java a partir d'un model XML existent. Però si el punt de partida són les classes Java, la solució passa per escriure notacions sobre el model o sobre alguna classe derivada.

Cal que entengueu la derivació com a herència.

Si partim d'un model dissenyat totalment per nosaltres caldrà crear, a banda de les classes del model, la classe `ObjectFactory`i el fitxer *package-info.java*. L'`ObjectFactory`tindrà almenys un mètode de creació per a cada classe del model. Si fos necessari, una classe podria disposar de més d'un mètode d'instanciació, de la mateixa manera que fos necessari tenir diversos constructors.

Si, en canvi, partíssim d'un model derivat d'una altre al qual no hi tinguéssim accés, perquè fos un biblioteca de tercers, perquè no disposéssim de les fonts o per qualsevol altre motiu, només podríem anotar les classes derivades. Com ja hem vist, les *Anotacions* tenen diferents àmbits d'influència segons el tipus. Hi ha anotacions de paquets que afecten totes les seves classes, però hi ha també anotacions de classes especifiques, el rang de les quals afectaria només a la classe on s'ha definit i als seus components, mètodes, atributs, etc.

Un problema que trobarem quan treballem amb herència usant JAXB és que les *Anotacions* de classes afecten només la classe específica on s'han escrit les anotacions, i no serveixen per a les superclasses. No és possible, per exemple, anotar en la classe `Cercle` un atribut de `Figura`. Si necessitem anotar l'atribut caldrà fer-ho dins la mateixa classe `Figura`. Però si no hi tinguéssim accés, existeixen diverses solucions. Aquí plantejarem la més laboriosa, però també la que permet un major grau de configuració. La tècnica consisteix, d'una banda, a evitar que JAXB faci cap exploració de les classes que no puguem anotar. De l'altra, a dotar a les classes derivades d'accessors (mètodes `get`i `set`) propis per accedir als atributs heretats.

És possible optar per solucions intermèdies si la seriació per defecte que fa JAXB ja ens anés bé. En aquest cas, només caldria derivar la classe que representi l'etiqueta arrel per poder anotar-la com a tal.

Anotacions bàsiques

Per introduir les anotacions bàsiques de JAXB continuarem amb l'exemple del dibuix de figures geomètriques. Implementarem dues versions: en una, crearem el model des de zero i escriurem *anotacions* en qualsevol de les classes creades. A la segona versió, partirem del model que trobem a la biblioteca de manera que per anotar-lo ens calgui crear un model derivat. Farem les implementacions en paral·lel, i així copsarem millor les diferències.

Preparació dels paquets del model

Pel que fa a la implementació des de zero, en primer lloc caldrà crear el paquet. Per exemple:

```
ioc.dam.m6.exemples.dibuix.persistencia.xml.model
```

Per al model derivat, el paquet podria ser:

```
ioc.dam.m6.exemples.dibuix.persistencia.xml.modelderivat
```

També caldrà evitar que JAXB explori les classes originals de la biblioteca impedint-ne la seriació a l'XML. En el nostre projecte crearem un paquet amb el mateix nom que el paquet on es trobi el model original (el de la biblioteca). Un cop creat, hi ubicarem únicament un fitxer *package-info.java,* en el qual anotarem la deshabilitació dels accessors fent servir `@XmlAccessorType`, al qual li passarem per paràmetre el valor de la constant `@XmlAccessType.NONE`, la qual indicarà que les classes del paquet associat no seran accessibles des de JAXB.

```
@javax.xml.bind.annotation.XmlAccessorType(
  value= javax.xml.bind.annotation.XmlAccessType.NONE)
package ioc.dam.m6.exemples.dibuix;
```

Podreu crear un fitxer *package-info.java* clicant *new File*, escollint la categoria Java i seleccionant el tipus de fitxer *Java Package Info*.

Per a la creació del model, en la versió de la implementació total (des de zero) caldrà crear les mateixes classes (`Cercle`, `Rectangle`, `Grup`, `Figura`, `Dibuix`, etc.) que el model de la biblioteca.

Per al model derivat, caldrà implementar `XmlDibuix`, `XmlCercle`, `XmlRectangl` i `XmlGrup` heretant de les classes corresponents i implementant els accessors adequats. Veiem com a exemple `XmlCercle`:

```
public class XmlCercle extends Cercle {

    public XmlCercle() {
    }

    public XmlCercle(int x, int y, int radi) {
        super(x, y, radi);
    }

    @Override
    public Point getPosicio() {
        return super.getPosicio();
    }

    @Override
    public void setPosicio(Point posicio) {
        super.setPosicio(posicio);
    }

    @Override
    public Color getColorFigura() {
        return super.getColorFigura();
    }

    @Override
    public void setColorFigura(Color colorFigura) {
        super.setColorFigura(colorFigura);
    }

    @Override
    public Color getColorLinia() {
        return super.getColorLinia();
    }

```

```
35      @Override
36      public void setColorLinia(Color colorLinia) {
37          super.setColorLinia(colorLinia);
38      }
39
40      @Override
41      public int getEscala() {
42          return super.getEscala();
43      }
44
45      @Override
46      public void setEscala(int escala) {
47          super.setEscala(escala);
48      }
49
50      @Override
51      public float getRotacio() {
52          return super.getRotacio();
53      }
54
55      @Override
56      public void setRotacio(float rotacio) {
57          super.setRotacio(rotacio);
58      }
59
60      @Override
61      public int getRadi() {
62          return super.getRadi();
63      }
64
65      @Override
66      public void setRadi(int radi) {
67          super.setRadi(radi);
68      }
69  }
```

De forma semblant, caldrà implementar `XmlRectagle` i `XmlGrup`. D'aquesta darrera cal comentar que l'atribut `elements`necessita també accessors per tal que JAXB hi pugui accedir.

```
1   public class XmlGrup extends Grup{
2       public ArrayList<Figura> getElements(){
3           return elements;
4       }
5
6       public void setElements(ArrayList<Figura> list){
7           elements=list;
8       }
9
10      ...
11  }
```

Finalment, a la `classeXmlDibuix` hi afegirem un nou mètode per copiar la llista de figures d'un dibuix a un altre.

```
1   public class XmlDibuix extends Dibuix {
2       public XmlDibuix() {
3       }
4
5       public List<Figura> getFigures(){
6           return this;
7       }
8
9       public void copiarDesDe(Dibuix dibuix){
10          this.clear();
11          this.addAll(dibuix);
```

```
    }
}
```

Per a ambdues versions, i amb l'objectiu que JAXB reconegui les classes que ha de manipular i *mapar*, caldrà crear un arxiu *package-info.java* i un `ObjectFactory`a cada paquet.

Al fitxer amb la informació del paquet hi escriurem l'anotació `@XmlSchema`, que ens permetrà definir un espai de noms (http://xml.ioc.edu/schema/dibuix, per exemple) i indicar si els noms són o no qualificats. També indicarem de forma exclusiva en la versió del model derivat, que l'accés a les dades vinculades amb l'XML es realitzarà per mitjà d'accessors (mètodes `getiset`) usant el valor de la constant *XmlAccessType.PROPERTY*, mentre que el tipus d'accés per al model propi serà directe als atributs. Usarem el valor `XmlAccessType.FIELD`.

El contingut del *package-info.java* del model derivat és el següent:

```
@javax.xml.bind.annotation.XmlSchema(
  namespace ="http://xml.ioc.edu/schema/dibuix",
  elementFormDefault = javax.xml.bind.annotation
                                    .XmlNsForm.QUALIFIED)
@javax.xml.bind.annotation.XmlAccessorType(
    value= javax.xml.bind.annotation
                               .XmlAccessType.PROPERTY)
package ioc.dam.m6.exemples.dibuix.persistencia.xml.modelderivat;
```

I el del *package-info.java* del model propi, aquest:

```
@javax.xml.bind.annotation.XmlSchema(
    namespace = "http://xml.ioc.edu/schema/dibuix",
    elementFormDefault = javax.xml.bind
                            .annotation.XmlNsForm.QUALIFIED)
@javax.xml.bind.annotation.XmlAccessorType(
    value=javax.xml.bind.annotation.XmlAccessType.FIELD)
package ioc.dam.m6.exemples.dibuix.persistencia.xml.model;
```

Seguidament, caldrà crear i adaptar les classes *factory* que per defecte usa JAXB. Crearem en cada paquet una classe `ObjectFactory` que anotarem amb `@XmlRegistry`. Aquesta anotació indica a JAXB que es tracta d'una classe instanciadora i que, per tant, analitzant els seus mètodes es poden obtenir totes les classes usades durant la vinculació.

En el paquet del model propi, la classe `ObjectFactory` tindrà mètodes per generar rectangles, cercles, grups o dibuixos. Els mètodes instanciadors es compondran del prefix "*create*" i d'un sufix amb el nom que l'XML usarà com a referència d'aquesta classe:

```
@XmlRegistry
public class ObjectFactory {

    public ObjectFactory() {
    }

    public Grup createGrup() {
        return new Grup();
    }

    public Rectangle createRectangle() {
```

```
12          return new Rectangle();
13      }
14
15      public Rectangle createRectangle(int x, int y,
16                                       int amplada, int alcada) {
17          return new Rectangle(x, y, amplada, alcada);
18      }
19
20      public Cercle createCercle() {
21          return new Cercle();
22      }
23
24      public Cercle createCercle(int x, int y, int radi) {
25          return new Cercle(x, y, radi);
26      }
27
28      public Dibuix createDibuix() {
29          return new Dibuix();
30      }
31  }
```

Per facilitar la codificació de crear objecte de tipus rectangle i cercle, afegirem dos mètodes més que acceptin paràmetres d'inicialització. Aquest mètodes no podran substituir els instanciadors per defecte, ja que JAXB usa sempre instanciadors sense paràmetres.

De forma semblant, en el paquet del model derivat:

```
1   @XmlRegistry
2   public class ObjectFactory {
3
4       public ObjectFactory() {
5       }
6
7       public XmlGrup createGrup() {
8           return new XmlGrup();
9       }
10
11      public XmlRectangle createRectangle() {
12          return new XmlRectangle();
13      }
14
15      public XmlRectangle createRectangle(int x, int y, int amplada,
16          int alcada) {
17          return new XmlRectangle(x, y, amplada, alcada);
18      }
19
20      public XmlCercle createCercle() {
21          return new XmlCercle();
22      }
23
24      public XmlCercle createCercle(int x, int y, int radi) {
25          return new XmlCercle(x, y, radi);
26      }
27
28      public XmlDibuix createDibuix() {
29          return new XmlDibuix();
30      }
31  }
```

Anotacions de classe

L'anotació `@XmlElementRoot` serveix per indicar quina classe o quines classes són candidates a usar-se com a node arrel del document XML. Les instàncies d'aquestes classes poden accedir de manera directa o indirecta a tota la informació que s'emmagatzemarà al'XML. Cal recordar que XML no admet documents amb arrels múltiples i, per tant, no és possible seriar les dades a partir de dos o més objectes. En el nostre cas, la classe `arrel` correspon a `Dibuix`. Afegiu sobre la declaració de la classe l'`Anotació`.

```
@XmlRootElement(name="dibuix")
public class Dibuix extends ArrayList<Figura>{
    ...
```

@XmlElementRoot accepta el paràmetre `nom`, el qual indica el nom que tindrà l'etiqueta arrel en el document XML. Feu el mateix a la classe `XmlDibuix`.

Emmagatzemeu els canvis i obriu la classe `Figura`. Afegirem una notació per indicar l'ordre en què voldrem escriure els seus atributs.

```
@XmlType(propOrder={"posicio","escala","rotacio"})
public abstract class Figura {
    ...
```

L'Anotació `@XmlType` permet, entre d'altres coses, especificar l'ordre en què s'escriuran els seus camps. Observeu a la sintaxi com les Anotacions reben un conjunt de dades: el conjunt de dades es tanca entre claus, i per separar cada una de les dades s'intercala una coma.

Aquesta *Anotació* és opcional. De fet, si no la poséssim, tant el *marshaller* com l'*unmarshaller* funcionarien sense cap problema. De tota manera, es tracta d'una *anotació* important perquè sovint els XML han de complir DTD o Schemes força restrictius que requereixen un ordre determinat de les etiquetes.

Farem quelcom semblant a la classe `FiguraSimple` i `Rectangle`.

```
@XmlType(propOrder = {"colorLinia","colorFigura"})
public class FiguraSimple extends Figura{
    ...

@XmlType(propOrder = {"alcada","amplada"})
public class Rectangle extends FiguraSimple{
    ...
```

No és necessari definir l'ordre dels atributs de la classe `Cercle` ni la classe `Grup`, perquè només en tenen un.

Les operacions equivalents al paquet del model derivat haurien de poder definir el mateix ordre, però com que no tenim accés a les classes derivades caldrà fer les anotacions a la pròpia classe, que és on hi trobarem els accessors (`getsi sets`). A més, caldrà indicar que el tipus XML que aquesta classe representa no coincideix amb el seu nom.

```
@XmlType(name="Rectangle",
      propOrder={"posicio","rotacio","escala","colorFigura",
                 "colorLinia", "alcada","amplada"})
public class XmlRectangle extends Rectangle{
    ...
```

Cal realitzar la mateixa operació amb totes i cada una de les classes derivades:

```
@XmlType(name="Cercle",
      propOrder={"posicio","rotacio","escala", "colorFigura",
                 "colorLinia","radi"})
public class XmlCercle extends Cercle {
    ...

@XmlType(name="Grup",
      propOrder={"posicio","rotacio","escala","elements"})
public class XmlGrup extends Grup{
    ...
```

Anotacions de camps, mètodes o accessors

De vegades caldrà matisar vincles específics pels elements de les classes. Podem, per exemple, evitar que el valor d'un determinat camp es traspassi a l'XML. Això ho aconseguirem marcant-lo amb la notació `@XmlTransient`. És el que farem amb el camp TipusFigura de la classe `Figura` del model propi.

```
@XmlType(propOrder={"posicio","escala","rotacio"})
public abstract class Figura {
    @XmlTransient
    private TipusFigura tipusFigura=TipusFigura.INDETERMINADA;
    private Point posicio=null;
    ...
```

En el model derivat no cal fer-ho, perquè ja hem impedit l'accés a nivell del paquet. Malgrat tot, caldrà definir altres anotacions. Com que no hi ha accés directe als camps, JAXB no sabrà quin nom posar a les etiquetes de l'XML i, per defecte, JAXB usarà sempre el nom del camp associat. En el model derivat, caldrà indicar a l'Anotació el nom desitjat a l'etiqueta. Les Anotacions s'han de fer només a un dels dos accessors, i per convenció solen posar-se en els mètodes de lectura (`get`), però s'accepten a qualsevol dels dos.

Les propietats comunes caldrà repetir-les a totes les classes:

```
...

    @XmlElement(name="posicio")
    @Override
    public Point getPosicio() {
        return super.getPosicio();
    }
    ...

    @XmlElement(name="escala")
    @Override
    public int getEscala() {
        return super.getEscala();
    }
    ...
```

```
16
17      @XmlElement(name="rotacio")
18      @Override
19      public float getRotacio() {
20          return super.getRotacio();
21      }
22      ...
23
24      @XmlElement(name="colorFigura")
25      @Override
26      public Color getColorFigura() {
27          return super.getColorFigura();
28      }
29      ...
30
31      @XmlElement(name="colorLinia")
32      @Override
33      public Color getColorLinia() {
34          return super.getColorLinia();
35      }
36      ...
```

I les específiques, a les classes corresponents:

```
@XmlElement(name="alcada")
    @Override
    public int getAlcada() {
        return super.getAlcada();
    }
    ...

    @XmlElement(name="amplada")
    @Override
    public int getAmplada() {
        return super.getAmplada();
    }
    ...
```

El tractament de col·leccions també s'especifica a nivell d'element o de propietat. Per defecte, les col·leccions Java es plasmaran al'XML com una seqüència d'elements descendents directes de la classe contenidora. És a dir, en el cas dels *grups*, que contenen una llista de *figures*, si no s'indica res més, la seqüència de figures es penjarà directament de l'etiqueta *grup*. No és això el que volem. De fet, es vol fer dependre la seqüència de *figures* d'un element extra anomenat *colleccio*, el qual formaria part de l'element grup i contindria les *figures* de forma directa. JAXB disposa de l'*Anotació* `@XmlElementWrapper` per indicar la necessitat d'una etiqueta contenidora extra.

D'altra banda, els elements de la llista de figures a vegades seran rectangles, d'altres cercles i d'altres grups. La manera d'associar un tipus de dada amb una etiqueta es pot especificar aquí usant l'Anotació `@XmlElements`, a la qual se li indicarà usant una col·lecció de `@XmlElement` quines classes s'associaran a quines etiquetes.

```
public class Grup extends Figura{
    @XmlElementWrapper(name="colleccio", required=true)
    @XmlElements({
        @XmlElement(name="rectangle",type=Rectangle.class),
        @XmlElement(name="cercle",type=Cercle.class),
        @XmlElement(name="grup",type=Grup.class)
    })
    ArrayList<Figura> elements = new ArrayList<Figura>();
    ...
```

El mateix cal fer a la classe `XmlGrup`, però associant les classes corresponents.

```
@XmlElementWrapper(name="colleccio", required=true)
    @XmlElements({
        @XmlElement(name="rectangle",type=XmlRectangle.class),
        @XmlElement(name="cercle",type=XmlCercle.class),
        @XmlElement(name="grup",type=XmlGrup.class)
    })
    public ArrayList<Figura> getElements(){
        return elements;
    }
```

La classe `Dibuix` és també una llista de característiques molt similars a la classe `Grup`. Caldrà, doncs, una anotació semblant. Abans, però, haurem de salvar un problema: l'Anotació `@XmlElements` només pot ser definida a nivell de mètode o d'atribut i no pas de classe. Com que `Dibuix` hereta d'`ArrayList`, no disposa de cap atribut contenidor, la solució passa per crear un accessor de lectura on es pugui especificar la notació:

```
@XmlRootElement(name="dibuix")
public class Dibuix extends ArrayList<Figura>{
    @XmlElements({
        @XmlElement(name="rectangle",type=Rectangle.class),
        @XmlElement(name="cercle",type=Cercle.class),
        @XmlElement(name="grup",type=Grup.class)
    })
    public ArrayList<Figura> getFigures(){
        return this;
    }
    ...
```

Cal, també, realitzar una anotació idèntica a la classe `XmlDibuix`, substituint les classes originals per les derivades corresponents.

```
@XmlElements({
    @XmlElement(name="rectangle",type=XmlRectangle.class),
    @XmlElement(name="cercle",type=XmlCercle.class),
    @XmlElement(name="grup",type=XmlGrup.class)
})
public List<Figura> getFigures(){
    return this;
}
```

Rectificació de les anotacions a nivell de paquet

Malauradament, un cop s'han realitzat totes aquestes modificacions, si fem la prova intentant seriar una instància de Dibuix ens saltarà una excepció. Això és degut al fet que el nostre model utilitza dues classes pròpies de Java que JAXB no sap *mapar*. Ens referim a `Point` i a `Color`. JAXB necessita que totes les classes del model tinguin un constructor per defecte (constructor sense paràmetres) i que els atributs que calgui seriar tinguin accessors de lectura i escriptura (`get` i `set`). La classe `Color` no disposa de constructor per defecte, i la classe `Point` té un accessor de tipus *double* corresponent a un atribut de tipus enter i un accessor anomenat `getLocation` que no es correspon a cap atribut.

Per solucionar aquests problemes, JAXB disposa d'unes anotacions que permeten declarar classes o tipus alternatius, així com la forma de passar d'un tipus a un

altre. Ens referim a `@XmlJavaTypeAdapter` i a la classe `XmlAdapter`. La primera és l'Anotació amb la qual s'informarà de quines classes han de fer d'adaptadors. La segona és una classe abstracta i parametritzada amb els tipus que caldrà intercanviar, d'on qualsevol adaptador haurà de derivar.

Per exemple, en passar al'XML, volem que les figures escriguin el color com si fos un valor numèric. Crearem una Classe que hereti de `XmlAdapter`, que haurà d'implementar els seus dos mètodes abstractes anomenats `unmarshal` i `marshal`. El mètode `unmarshal`permet convertir el valor obtingut des del'XML a un valor de tipus vàlid en el llenguatge Java. En el nostre cas, rebrà un enter per paràmetre i el transformarà a `Color`.

```
public class ColorAdapter extends XmlAdapter<Integer, Color> {

    @Override
    public Color unmarshal(Integer v) throws Exception {
        return new Color(v.intValue());
    }

    @Override
    public Integer marshal(Color v) throws Exception {
        return new Integer(v.getRGB());
    }

}
```

El mètode `marshal` fa l'operació inversa: rep per paràmetre el valor provinent de la classe Java i el converteix en un valor adequat per emmagatzemar al'XML.

Farem el mateix amb la classe `Point`. La diferència és, però, que els punts no poden transformar-se en un tipus simple, ja que sempre es necessiten dues coordenades. La classe alternativa la crearem nosaltres mateixos assegurant-nos que compleix els requisits de JAXB:

```
public class Posicio implements Serializable{
    private int coordenadaX;
    private int coordenadaY;
    public Posicio() {
    }

    public Posicio(Point p){
        this.coordenadaX=p.x;
        this.coordenadaY=p.y;
    }

    @XmlAttribute(name="x")
    public int getCoordenadaX() {
        return coordenadaX;
    }

    public void setCoordenadaX(int coordenadaX) {
        this.coordenadaX = coordenadaX;
    }

    @XmlAttribute(name="y")
    public int getCoordenadaY() {
        return coordenadaY;
    }

    public void setCoordenadaY(int coordenadaY) {
        this.coordenadaY = coordenadaY;
    }
}
```

També implementarem el seu adaptador:

```
public class PointAdapter extends XmlAdapter<Posicio, Point> {

    @Override
    public Point unmarshal(Posicio v) throws Exception {
        return new Point(v.getCoordenadaX(), v.getCoordenadaY());
    }

    @Override
    public Posicio marshal(Point v) throws Exception {
        return new Posicio(v);
    }

}
```

Finalment, per tal que els canvis tinguin efecte, caldrà declarar-ho en forma d'Anotacions. Això ho farem a nivell de paquet. La notació rep dos paràmetres: el paràmetre *type* identifica la classe incompatible, i el paràmetre *value* especificarà per quina classe caldrà substituir-la. Per tal que es puguin definir tants adaptadors com sigui necessari, aquests es passaran per paràmetre de l'anotació `XmlJavaTypeAdapters`.

```
@javax.xml.bind.annotation.XmlSchema(
   namespace = "http://xml.netbeans.org/schema/dibuix",
   elementFormDefault = javax.xml.bind.annotation.XmlNsForm.QUALIFIED)

@javax.xml.bind.annotation.adapters.XmlJavaTypeAdapters({
    @javax.xml.bind.annotation.adapters.XmlJavaTypeAdapter(
        value=ioc.dam.m6.exemples.dibuix.persistencia.xml.binding.
            ColorAdapter.class,
        type=java.awt.Color.class),
    @javax.xml.bind.annotation.adapters.XmlJavaTypeAdapter(
        value=ioc.dam.m6.exemples.dibuix.persistencia.xml.binding.
            PointAdapter.class,
        type=java.awt.Point.class)
})

package ioc.dam.m6.exemples.dibuix.persistencia.xml.binding.model;
```

Implementació del controlador

Tal com hem fet en anteriors ocasions, crearem una classe `Controlador` que aglutini un conjunt d'utilitats per facilitar les crides a la seriació i deseriació en format XML usant les eines JAXB. Bàsicament, la Utilitat que construirem ha de facilitar la creació d'un context específic, així com l'escriptura des del model al'XML i la lectura des del'XML al model.

Aquesta classe l'anomenarem `BindingCtrl`. Contindrà una instància d'Utilitats, un `File` referenciant el fitxer d'emmagatzematge XML i una cadena representant el nom del paquet on es trobaran les classes del model de dades. Crearem un mètode *init* que inicialitzarà el controlador creant un JAXBContext. També disposarà d'utilitats per realitzar la seriació/deseriació a l'XML.

```
public class BindingCtrl {
    protected Utilitats utilitats = new Utilitats();
    protected File file = null;
    protected boolean init = false;
    protected JAXBContext context;
```

```
    protected String nomPaquet=null;

    public BindingCtrl() {
    }

    public BindingCtrl(String ctx) {
        nomPaquet = ctx;
    }

    public File getFile() {
        return file;
    }

    public void init() throws JAXBException {
        if(nomPaquet!=null){
            context = JAXBContext.newInstance(nomPaquet);
        }else{
            context = JAXBContext.newInstance(classFactory);
        }
        init=true;
    }

    public void setFile(File file) {
        this.file = file;
    }

    protected void escriu(Object object) throws BindingCtrlException{
        FileOutputStream out = null;
        try {
            if(!init){
                init();
            }
            Marshaller marshaller = context.createMarshaller();
            marshaller.setProperty(Marshaller.JAXB_ENCODING, "UTF-8");
            marshaller.setProperty(Marshaller.JAXB_FORMATTED_OUTPUT,
                                   true);
            out = new FileOutputStream(file);
            marshaller.marshal(object, out);
        } catch (FileNotFoundException ex) {
            throw new BindingCtrlException(ex);
        } catch (JAXBException ex) {
            throw new BindingCtrlException(ex);
        } finally {
            utilitats.intentarTancar(out);
        }
    }

    protected Object llegeix() throws BindingCtrlException {
        Object object = null;
        FileInputStream in=null;
        try {
            if(!init){
                init();
            }
            Unmarshaller unmarshaller = context.createUnmarshaller();
            in = new FileInputStream(file);
            object = unmarshaller.unmarshal(in);
        } catch (FileNotFoundException ex) {
            throw new BindingCtrlException(ex);
        } catch (JAXBException ex) {
            throw new BindingCtrlException(ex);
        } finally {
            utilitats.intentarTancar(in);
        }
        return object;
    }
}
```

A partir d'aquesta classe, per herència podem crear controladors específics per cada model concret. Per exemple, la del model derivat s'implementarà fent el següent:

```
public class ModelDerivatCtrlDibuix extends BindingCtrl{
    Dibuix root;

    public ModelDerivatCtrlDibuix(){
        this(new Dibuix());
    }

    public ModelDerivatCtrlDibuix(Dibuix dibuix){
        super(dibuix.getClass().getName().substring(0,
                    dibuix.getClass().getName().lastIndexOf(".")));
        this.root = dibuix;
    }

    public ModelDerivatCtrlDibuix(Dibuix dibuix, File file){
        this(dibuix);
        this.file=file;
    }

    public void emmagatzemar() throws xmlDibuixException{
        try {
            escriu(root);
        } catch (BindingCtrlException ex) {
            throw new xmlDibuixException(ex);
        }
    }

    public void recuperar() throws xmlDibuixException {
        try {
            root.clear();
            root.addAll((Dibuix) this.llegeix());
        } catch (BindingCtrlException ex) {
            throw new xmlDibuixException(ex);
        }
    }

    public Dibuix getDibuix() {
        return root;
    }

}
```