# Fitxers

Joan Arnedo Moreno

**Programació bàsica (ASX)**
**Programació (DAM)**
**Programació (DAW)**

# Índex

## Introducció

La principal funció d'una aplicació informàtica és la manipulació i transformació de dades. Aquestes dades poden representar coses molt diferents segons el context del programa: notes d'estudiants, un recull de temperatures, les dates d'un calendari, etc. Les possibilitats són il·limitades. Totes aquestes tasques de manipulació i transformació es porten a terme normalment mitjançant l'emmagatzemament de les dades en variables, dins la memòria de l'ordinador, de manera que s'hi poden aplicar operacions, ja sigui mitjançant operadors o la invocació de mètodes.

Malauradament, totes aquestes variables només tenen vigència mentre el programa s'està executant. Un cop el programa finalitza, les dades que contenen desapareixen. Això no és problema per a programes que sempre tracten les mateixes dades, que poden prendre la forma de literals dins el programa. O quan el nombre de dades a tractar és petit i es pot preguntar a l'usuari. Ara bé, imagineu-vos haver d'introduir les notes de tots els estudiants cada vegada que s'executa el programa per gestionar-les. No té cap sentit. Per tant, en alguns casos, apareix la necessitat de poder enregistrar les dades en algun suport de memòria externa, de manera que aquestes es mantinguin de manera persistent entre diferents execucions del programa, o fins i tot si s'apaga l'ordinador.

La manera més senzilla d'assolir aquest objectiu és emmagatzemar la informació aprofitant el sistema de fitxers que ofereix el sistema operatiu. Mitjançant aquest mecanisme, és possible tenir les dades en un format fàcil de gestionar i independent del suport real, ja sigui un suport magnètic com un disc dur, una memòria d'estat sòlid, com un llapis de memòria USB, un suport òptic, cinta, etc.

A l'apartat "Gestió de fitxers" es presenten un seguit de conceptes bàsics vinculats a la gestió de fitxers mitjançant el llenguatge Java. Sense voler enumerar cadascuna de les classes o mètodes que conformen la biblioteca d'entrada / sortida de Java (**java.io**), s'explica com dur a terme les primeres passes per controlar i manipular el sistema de fitxers de l'ordinador mitjançant codi font en els vostres programes, independentment del seu sistema operatiu.

Un cop es disposa dels coneixements bàsics per manipular el sistema de fitxers, a l'apartat "Tractament bàsic de dades" s'explica com poder llegir i enregistrar dades dins un fitxer. Hi ha dues maneres de dur a terme aquest procés: mitjançant accés seqüencial, un sistema molt semblant a com es llegirien pel teclat o s'escriurien per pantalla, o relatiu, que s'aproxima més a com es gestionen les dades d'un *array*. Independentment del sistema d'accés usat, un altre aspecte molt important que haureu de tenir en compte i cal estudiar és el format com emmagatzemar les dades, com un fitxer de text o en la seva codificació binària original.

Finalment, l'apartat "Tractament modular de dades" lliga tots els conceptes vinculats a la gestió de fitxers amb els de disseny descendent i modularitat, usant com a fil argumental un exemple de programa de certa complexitat. Es parteix

d'una aplicació ja existent i, aplicant el principi de modularitat, afegireu noves funcionalitats associades a la gestió de fitxers.

## Resultats d'aprenentatge

En finalitzar aquesta unitat, l'alumne/a:

1. Reconeix el conceptes relacionats amb fitxers.
2. Reconeix els diferents tipus de fitxers.
3. Estableix i diferencia les operacions a realitzar sobre els fitxers en el llenguatge de programació emprat
4. Utilitza correctament diferents operacions sobre fitxers.
5. Modularitza adequadament els programes que gestionen fitxers.
6. Dissenya, prova i documenta programes simples que gestionen fitxers.

# 1. Gestió de fitxers

Entre les funcions d'un sistema operatiu hi ha la d'oferir mecanismes genèrics per gestionar sistemes de fitxers. Normalment, dins un sistema operatiu modern (o ja no tant modern), s'espera disposar d'algun tipus d'interfície o explorador per poder gestionar fitxers, ja sigui gràficament o usant una línia d'ordres de text. Si bé la manera com les dades es desen realment en els dispositius físics d'emmagatzematge de dades pot ser molt diferent segons cada tipus (magnètic, òptic, etc.), la manera de gestionar el sistema de fitxers sol ser molt similar en la immensa majoria dels casos: una estructura jeràrquica amb carpetes i fitxers.

Ara bé, en realitat, la capacitat d'operar amb el sistema de fitxers no és exclusiva de la interfície oferta pel sistema operatiu. Molts llenguatges de programació proporcionen biblioteques que permeten accedir directament als mecanismes interns que ofereix el sistema, de manera que és possible crear codi font des del qual, amb les instruccions adients, es poden realitzar operacions típiques d'un explorador de fitxers. De fet, les interfícies ofertes, com un explorador de fitxers, són un programa com qualsevol altre, el qual, usant precisament aquestes llibreries, permet que l'usuari gestioni fitxers fàcilment. Però no és estrany trobar altres aplicacions amb la seva pròpia interfície per gestionar fitxers, encara que només sigui per poder seleccionar quin cal carregar o desar en un moment donat: editors de text, compressors, reproductors de música, etc.

Java no és cap excepció oferint aquest tipus de biblioteca, en forma del conjunt de classes incloses dins del *package* **java.io**. Mitjançant la invocació dels mètodes adients definits d'aquestes classes, és possible dur a terme pràcticament qualsevol tasca sobre el sistema de fitxers.

## 1.1 La classe File

La peça més bàsica per poder operar amb fitxers, independentment del seu tipus, en un programa fet en Java és el tipus compost `File`. Aquesta classe pertany al *package* **java.io**. Per tant, l'haureu d'importar abans de poder usar-la. Aquesta us permet manipular qualsevol aspecte vinculat al sistema de fitxers. Ara bé, cal anar amb compte, ja que el seu nom ("fitxer", en anglès) és una mica enganyós, ja que no es refereix exactament a un fitxer.

La classe **File** indica, més concretament, una ruta dins el sistema de fitxers.

Concretament, serveix per fer operacions tant sobre rutes al sistema de fitxers que ja existeixin com no existents. A més a més, aquesta classe es pot usar tant per manipular fitxers de dades com carpetes.

### 1.1.1 Inicialització

Com qualsevol altra classe, abans de poder treballar amb `File`, a part d'importar-la correctament, cal inicialitzar-la, de manera que sigui possible invocar els seus mètodes. En aquest cas, per fer-ho cal incloure com a paràmetre una cadena de text corresponent a la ruta sobre la qual es volen dur a terme les operacions.

```
1 File f = new File (String ruta)
```

> Una **ruta** és la forma general d'un nom de fitxer o carpeta, de manera que identifica únicament la seva localització en el sistema de fitxers.

Cadascun dels elements de la ruta poden existir realment o no, però això no impedeix de cap manera poder inicialitzar `File`. En realitat, el seu comportament és com una declaració d'intencions sobre quina ruta del sistema de fitxers es vol interactuar. No és fins que es criden els diferents mètodes definits a `File`, o fins que s'hi escriuen o s'hi llegeixen dades, que realment s'accedeix al sistema de fitxers i es processa la informació.

Cal tenir sempre present si el sistema operatiu distingeix entre majúscules i minúscules o no.

Un aspecte important que cal tenir present en inicialitzar `File` és tenir sempre present que el format de la cadena de text que conforma la ruta pot ser diferent segons el sistema operatiu sobre el qual s'executa l'aplicació. Per exemple, el sistema operatiu Windows inicia les rutes per un nom d'unitat (C:, D:, etc.), mentre que els sistemes operatius basats en Unix comencen directament amb una barra ("/"). A més a més, els diferents sistemes operatius usen diferents separadors dins les rutes. Per exemple, els sistemes Unix usen la barra ("/") mentre que el Windows la contrabarra ("\").

- Ruta Unix: `/usr/bin`
- Ruta Windows: `C:\Windows\System32`

Per generar aplicacions totalment portables en diferents sistemes, la classe File ofereix la possibilitat d'accedir a una constant declarada dins d'aquesta classe per especificar separadors de ruta dins una cadena de text de manera genèrica, anomenada `File.separator`. Quan s'usa, el seu valor és el que correspon segons el sistema operatiu on s'executa el programa. L'accés a aquesta constant es fa de manera molt semblant a com s'invoca un mètode estàtic.

```
1 File f = new File(File.separator + "usr" + File.separator + "bin");
```

De totes maneres, Java adopta per defecte el sistema Unix com a separador per defecte (usant la barra, `/`), independentment del sistema operatiu. Per tant, en un sistema Windows és possible separar una ruta amb aquest caràcter i Java ho sabrà interpretar correctament. A partir d'ara, tots exemples es basaran en un sistema Windows (rutes iniciades amb un nom d'unitat,, C:, D:, etc.), però s'usarà la barra per referir-se a qualsevol ruta.

Per escriure una contrabarra a una cadena de text cal escriure '\\'.

De fet, en fer un programa en Java en un sistema Windows cal ser especialment acurat amb aquest fet, ja que la contrabarra no és un caràcter permès dins una cadena de text, i serveix per declarar valors especials d'escapament (`\n` salt de línia, `\t` tabulador, etc.).

Un altre aspecte molt important en inicialitzar `File` és que es tracta d'una classe que defineix un tipus compost (com `String`), i no un simple repositori de mètodes (com `Scanner` o `Random`). En conseqüència, cada cop que s'inicialitza, a l'identificador emprat s'hi assigna un valor associat a aquella ruta. A partir de llavors, l'identificador actua com una variable dins el programa. Per operar amb diferents rutes alhora caldrà inicialitzar i manipular diferents variables, igual que es faria amb diferents cadenes de text.

Per exemple, en el codi següent es declaren tres variables diferents, cadascuna de les quals emmagatzema un valor de tipus `File` diferent. Per treballar amb cada ruta, ja sigui carpeta o fitxer, caldrà usar la variable corresponent.

```
1 ...
2 File carpetaFotos = new File("C:/Personal/LesMevesFotos");
3 File unaFoto = new File("C:/Personal/LesMevesFotos/Foto1.png");
4 File unaAltraFoto = new File("C:/Personal/LesMevesFotos/Foto2.png");
5 ...
```

### 1.1.2 Rutes relatives i absolutes

En els exemples emprats fins al moment per inicialitzar variables del tipus `File` s'han usat **rutes absolutes**, ja que és la manera de deixar més clar a quin element dins del sistema de fitxers, ja sigui fitxer o carpeta, s'està fent referència.

> Una **ruta absoluta** és aquella que es refereix a un element destinació a partir de la carpeta arrel d'un sistema de fitxers. En aquesta s'enumeren, una per una, totes les carpetes que hi ha entre la carpeta arrel fins a la carpeta que conté l'element destinació.

Les rutes absolutes es distingeixen fàcilment, ja que el text que les representa comença d'una manera molt característica depenent del sistema operatiu de l'ordinador. En el cas dels sistemes operatius Windows al seu inici sempre es posa el nom de la unitat ("C:", "D:", etc.), mentre que en el cas dels sistemes operatius Unix, aquestes comencen sempre per una barra ("/").

Per exemple, les cadenes de text següents representen rutes absolutes dins un sistema de fitxers de Windows:

- `C:\Fotos\Viatges` (ruta a una carpeta)
- `M:\Documents\Unitat6\Apartat1` (ruta a una carpeta)
- `N:\Documents\Unitat6\Apartat1\Activitats.txt` (ruta a un fitxer)

En canvi, en el cas d'una jerarquia de fitxers sota un sistema operatiu Unix, un conjunt de rutes podrien estar representades de la manera següent:

- `/Fotos/Viatges` (ruta a una carpeta)
- `/Documents/Unitat6/Apartat1` (ruta a una carpeta)
- `/Documents/Unitat6/Apartat1/Activitats.txt` (ruta a un fitxer)

En inicialitzar variables de tipus `File` usant una ruta absoluta, sempre cal usar la representació correcta segons el sistema en què s'executa el programa.

Si bé l'ús de rutes absolutes resulta útil per indicar amb tota claredat quin element dins el sistema de fitxers s'està manipulant, hi ha casos que el seu ús comporta certes complicacions. Suposeu que heu fet un programa en el qual es duen a terme operacions sobre el sistema de fitxers. Un cop veieu que funciona, deixeu el projecte Java a un amic, que el copia al seu ordinador dins d'una carpeta qualsevol i l'obre amb el seu entorn de treball. Perquè el programa li funcioni perfectament, abans caldrà que al seu ordinador hi hagi exactament les mateixes carpetes que heu usat al vostre ordinador, tal com estan escrites en el codi font del vostre programa. En cas contrari, no funcionarà, ja que les carpetes i fitxers esperats no existiran, i per tant, no es trobaran. Usar rutes absolutes fa que un programa sempre hagi de treballar amb una estructura del sistema de fitxers exactament igual sigui on sigui on s'executi, cosa no gaire còmoda.

Per resoldre aquest problema, a l'hora d'inicialitzar una variable de tipus `File`, també es pot fer referint-se a una ruta relativa.

Una **ruta relativa** es aquella que es considera que parteix des de la carpeta, o directori, de treball de l'aplicació. Aquesta carpeta pot ser diferent cada cop que s'executa el programa i depèn de la manera com s'ha dut a terme aquesta execució.

En els sistemes Windows, dins de les propietats d'una drecera, hi ha una opció per especificar el "Directori de treball" de la seva aplicació associada.

Quan un programa s'executa, ja sigui en Java o en qualsevol altre llenguatge de programació, per defecte se li assigna una carpeta de treball. Aquesta carpeta sol ser, per defecte, la carpeta des d'on es llença el programa, si bé hi ha sistemes operatius que permeten especificar-la explícitament mitjançant alguna opció sobre les seves dreceres o fitxers executables. En el cas d'un programa en Java executat a través d'un IDE, la carpeta de treball sol ser la mateixa carpeta on s'ha triat desar els fitxers del projecte.

El format d'una ruta relativa és semblant a una ruta absoluta, però mai s'indica l'arrel del sistema de fitxers. Directament es comença pel primer element escollit dins la ruta. Per exemple:

- `Viatges`
- `Unitat6\Apartat1`
- `Unitat6\Apartat1\Activitats.txt`

La propietat especial d'una ruta relativa és que, implícitament, es considera que aquesta sempre inclou el directori de treball de l'aplicació com a part inicial del text, tot i no haver-se escrit. Ara bé, el tret distintiu és que, precisament, aquest directori de treball pot variar. Per exemple, donada la inicialització següent, l'element al qual es refereix la ruta varia segons com indica la taula 1.1:

```
File f = new File ("Unitat6/Apartat1/Activitats.txt");
```

**Taula 1.1.** Rangs i paraules clau dels tipus primitius numèrics a Java

| Directori de treball | Ruta real |
|---|---|
| C:\Projectes\Java | C:\Projectes\Java\Unitat6\Apartat1\Activitats.txt |
| D:\Unitats | X:\Unitats\Unitat6\Apartat1\Activitats.txt |
| /Programes | /Programes/Unitat6/Apartat1/Activitats.txt |

Aquest mecanisme permet minimitzar el nombre de carpetes que cal garantir que hi ha quan executem programes en diferents equips, ja que només cal controlar la vinculada a la ruta relativa. La part vinculada al directori de treball varia automàticament per cada versió copiada en cada màquina. En el cas més senzill d'entendre, suposeu la inicialització següent:

```
File f = new File ("Activitats.txt");
```

Donada aquesta ruta, n'hi ha prou a garantir que el fitxer "Activitats.txt" està sempre al mateix directori de treball de l'aplicació, sigui quin sigui aquest (en un ordinador pot ser "C:\Programes", en un altre "/Java", etc.). En qualsevol de tots aquests casos, la ruta sempre serà correcta. De fet, encara més. Noteu com les rutes relatives a Java permeten crear codi independent del sistema operatiu, ja que no cal especificar un format d'arrel lligada a un sistema de fitxers concret ("C:", "D:", "/", etc.).

Java disposa de dues instruccions per consultar quina és la carpeta de treball d'un programa i canviar-la per un altre, si així es vol fer. De totes maneres, normalment es deixa la carpeta que assigna per defecte el sistema operatiu en executar el programa.

```
//Assigna en forma de cadena de text la ruta de la carpeta de treball actual.
String dirTreball = System.getProperty("user.dir");
```

```
//Indica quina és la ruta de la nova carpeta de treball a partir d'una cadena
    de text.
String nouDirTreball = "...";
System.setProperty("user.dir", nouDirTreball);
```

De totes maneres, a mesura que aneu veient com realitzar operacions amb fitxers a partir de rutes, es veurà més clar el funcionament de rutes relatives i absolutes.

### 1.1.3 Operacions típiques de la classe File

Al contrari que les cadenes de text, el tipus compost `File` no permet usar absolutament cap mena d'operador, ni tant sols el '+'. Qualsevol operació que es vulgui dur a terme amb una ruta s'haurà de fer mitjançant la invocació de mètodes sobre la variable corresponent. En aquest sentit, `File` ofereix diversos mètodes per poder manipular el sistema de fitxers o obtenir informació a partir de la seva ruta. Alguns dels més significatius per entendre'n les funcionalitats es mostren tot seguit, ordenats per tipus d'operació.

Aquest cop, per descriure cada mètode s'emprarà la seva definició tal com apareix a la documentació del Java, especificant el tipus del valor de retorn i dels seus paràmetres.

#### Obtenció de la ruta

Un cop s'ha inicialitzat una variable de tipus `File`, pot ser necessari recuperar la informació emprada durant la seva inicialització i conèixer en format text a quina ruta s'està referint, o com a mínim part d'ella.

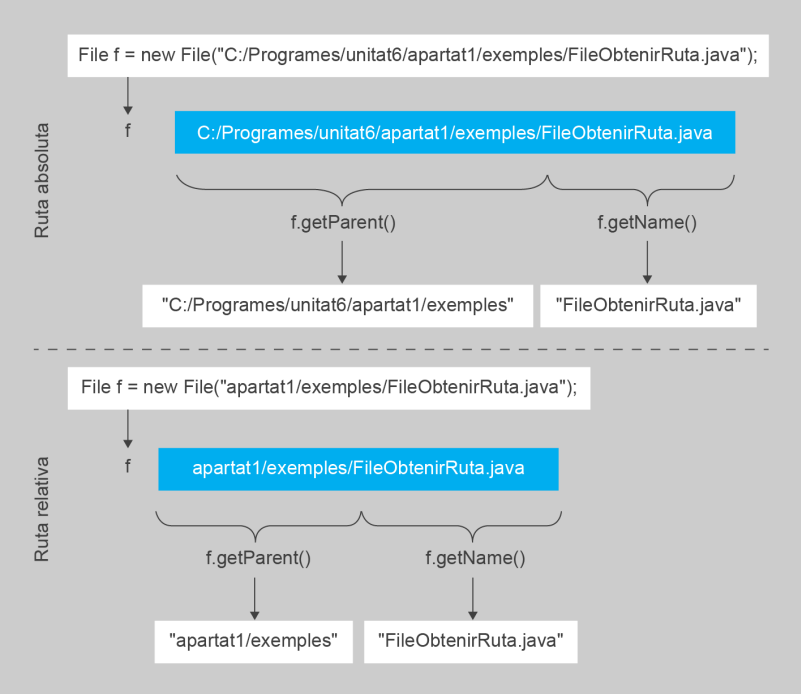
- `String getParent()` avalua la ruta de la carpeta pare de l'element referit per aquesta ruta. Bàsicament, la cadena de text resultant és idèntica a la ruta original usada en inicialitzar la variable de tipus `File`, eliminant el darrer element. Si la ruta tractada es refereix a la carpeta arrel d'un sistema de fitxers ("C:\", "/", etc.), aquest mètode avalua `null`. En el cas de tractar-se d'una ruta relativa, aquest mètode no inclou la part de la carpeta de treball.

- `String getName()` avalua El nom de L'element que representa aquesta ruta, ja sigui una carpeta o un fitxer. Es tracta del cas invers del mètode `getParent()`, ja que el text resultant és només el darrer element, descartant la resta del text de la ruta prèvia.

- `String getAbsolutePath()` avalua el text relatiu a la ruta absoluta. Si la variable `File` es va inicialitzar usant una ruta relativa, el resultat inclou també la part associada a la carpeta de treball.

La figura 1.1 mostra un resum del text que avalua cada mètode, donada una ruta concreta.

**Figura 1.1.** Avaluació dels mètodes d'obtenció de ruta a File




El programa següent mostra un exemple de com funcionen aquests tres mètodes. Noteu com a les rutes relatives se'ls afegeix la carpeta de treball, que serà la carpeta on teniu el vostre projecte amb el codi font del programa.

```
1  package unitat6.apartat1.exemples;
2  //Cal importar la classe File
3  import java.io.File;
4  public class FileObtenirRuta {
5    public static void main(String[] args) {
6      FileObtenirRuta programa = new FileObtenirRuta();
7      programa.inici();
8    }
9    public void inici() {
10     //S'inicialitzen dues rutes absolutes diferents
11     File carpetaAbs= new File ("c:/Temp");
12     File fitxerAbs = new File ("c:/Temp/Document.txt");
13     //I unes rutes relatives
14     File carpetaRel = new File ("Temp");
15     File fitxerRel = new File ("Temp/Document.txt");
16     //Mostrem les dades de cadascuna
17     mostrarRutes(carpetaAbs);
18     mostrarRutes(fitxerAbs );
19     mostrarRutes(carpetaRel );
20     mostrarRutes(fitxerRel );
21   }
22   public void mostrarRutes(File f) {
23     System.out.println("La ruta és " + f.getAbsolutePath());
24     System.out.println("El seu pare és " + f.getParent());
25     System.out.println("El seu nom és " + f.getName() + "\n");
26   }
27 }
```

### Comprovacions d'estat

Donada la ruta emprada per inicialitzar una variable de tipus `File`, aquesta pot ser que realment existeixi dins el sistema de fitxers o no, ja sigui en forma de

fitxer o carpeta. La classe `File` ofereix un conjunt de mètodes que permeten fer comprovacions sobre el seu estat i saber si és així.

`boolean exists()` comprova si la ruta existeix dins del sistema de fitxers o no. Avaluarà `true` o `false`, per a cada cas, respectivament. Es considera que la ruta no existeix si qualsevol dels elements individuals que la conformen no existeix.

Normalment, donat el text d'una ruta, els fitxers es distingeixen de les carpetes pel seu format, en tenir extensió (l'últim element de la ruta sol ser de la forma `nom.extensió`, com `Foto1.jpg` o `Document.txt`). Tot i així, cal tenir en compte que l'extensió no és un element obligatori en el nom d'un fitxer, només s'usa com a mecanisme perquè tant l'usuari com alguns programes puguin discriminar més fàcilment tipus de fitxers. Per tant, només amb el text d'una ruta no es pot estar 100% segur de si aquesta es refereix a un fitxer o a una carpeta. Per poder estar realment segurs, es poden usar els mètodes següents:

- `boolean isFile()` permet comprovar si la ruta es refereix a un fitxer (avaluant `true`) o no (avaluant `false`). Aquesta comprovació no es duu a terme únicament sobre el text de la ruta, sinó que es mira al sistema de fitxers si la ruta existeix, i llavors es mira si l'element amb aquest nom es tracta d'un fitxer o carpeta. Si no existeix, sempre s'avalua a `false`.

- `boolean isDirectory()` funciona igual que l'anterior, però comprovant si es tracta d'una carpeta.

Per exemple, el programa següent fa un seguit de comprovacions sobre un conjunt de rutes. Per poder-lo provar de manera efectiva, però, caldria preparar el sistema de fitxers del vostre ordinador. Per exemple, podeu crear una carpeta anomenada "Temp" a l'arrel de la unitat "C:". Dins d'aquesta, creeu-ne una altra anomenada "Fotos". A la mateixa carpeta "Temp", creeu un nou fitxer (que pot estar buit) anomenat "Document.txt". Després, esborreu la carpeta o el document i torneu a executar el programa.

```
1  package unitat6.apartat1.exemples;
2  import java.io.File;
3  public class FileComprovarFitxers {
4    public static void main(String[] args) {
5      FileComprovarFitxers programa = new FileComprovarFitxers();
6      programa.inici();
7    }
8    public void inici() {
9      File temp = new File ("C:/Temp");
10     File fotos = new File ("C:/Temp/Fotos");
11     File document = new File ("C:/Temp/Document.txt");
12     System.out.println(temp.getAbsolutePath() + " existeix? " + temp.exists());
13     mostrarEstat(fotos);
14     mostrarEstat(document);
15   }
16   public void mostrarEstat(File f) {
17     System.out.println(f.getAbsolutePath() + " és un fitxer? " + f.isFile());
18     System.out.println(f.getAbsolutePath() + " és una carpeta? " + f.
           isDirectory());
19   }
20 }
```

### Lectura i modificació de propietats

Quan la ruta amb la qual es treballa correspon a un fitxer o carpeta que realment existeix dins del sistema, hi ha tot un seguit d'informació subministrada pel sistema operatiu que pot resultar útil conèixer: els seus atributs d'accés, la seva mida, la data de modificació, etc. En general, totes les dades mostrades en accedir a les propietats del fitxer. Aquesta informació també pot ser consultada usant els mètodes adients. Entre els més populars n'hi ha els següents:

- `long length()` avalua la mida d'un fitxer, en bytes. Aquest mètode només pot ser invocat sobre una ruta que representi un fitxer, en cas contrari no es pot garantir la correctesa del resultat (no es pot considerar com a vàlid en cap cas).

- `long lastModified()` avalua la darrera data d'edició de l'element representat per aquesta ruta. El resultat es codifica en un únic número enter on es mesura el nombre de mil·lisegons que han passat des de l'1 de juny de 1970.

**De mil·lisegons a una data llegible**

La veritat és que, per tal de mostrar dades a l'usuari, treballar amb mil·lisegons per indicar una data no és gaire còmode. Afortunadament, dins el *package* **java.util** és possible trobar un tipus compost auxiliar que permet treballar amb dates (igual que `File` permet treballar amb rutes de fitxers). Es tracta de la classe `Date`. Per inicialitzar-la, precisament, cal usar com a paràmetre el nombre de mil·lisegons des de l'1 de juny de 1970, en format `long`.

Date data = new Date(1306751122651L);

No s'entrarà en més detall sobre com funciona aquesta classe. Si voleu, podeu mirar els seus mètodes a la documentació de l'API de Java.

L'exemple següent mostra com funcionen aquests mètodes. Per provar-los, abans creeu un fitxer anomenat "Document.txt" a la carpeta "C:\Temp". Inicialment, deixeu-lo buit i executeu el programa. Després, amb un editor de text, escriviu qualsevol cosa, deseu els canvis i torneu a executar el programa. Observeu com el resultat és diferent. Fixeu-vos també en l'ús de la classe `Date` per poder mostrar correctament la data en un format llegible.

```
1  package unitat6.apartat1.exemples;
2  import java.io.File;
3  import java.util.Date;
4  public class FileLlegirPropietats {
5    public static void main(String[] args) {
6      FileLlegirPropietats programa = new FileLlegirPropietats();
7      programa.inici();
8    }
9    public void inici() {
10     File document = new File("C:/Temp/Document.txt");
11     System.out.println(document.getAbsoluteFile());
12     //S'usa el tipus composta Date per transformar mil·lisegons a data real
13     Date data = new Date(document.lastModified());
14     System.out.println("Darrera modificació: " + data);
15     System.out.println("Mida: " + document.length());
16   }
17 }
```

### Gestió de fitxers

De ben segur, el conjunt d'operacions més habituals quan accediu a un sistema de fitxers en treballar en el vostre ordinador són les vinculades a la seva gestió directa: reanomenar fitxers, esborrar-los, copiar-los o moure'ls. Donat el nom d'una ruta, Java també permet realitzar aquestes accions.

- `boolean mkdir()` permet crear una carpeta d'acord a la ubicació que indica la ruta. La ruta ha d'indicar el nom d'una carpeta que no existeix en el moment d'invocar el mètode. Concretament, es crea una carpeta amb el nom de la ruta (el darrer element del text), ubicada a la carpeta pare indicada (tot el text anterior). Si s'ha creat correctament, el mètode avalua `true`. En cas contrari, avalua `false`. Hi ha diversos motius pels quals pot fallar la creació, però normalment es deurà al fet que la ruta és incorrecta (algun dels elements no existeixen) o la carpeta ja existia.

- `boolean delete()` esborra el fitxer o carpeta que indica la ruta. La ruta ha d'indicar el nom d'una carpeta que sí que existeix en el moment d'invocar el mètode. Igual com en l'anterior mètode, avalua `true` o `false` segons si l'operació realment s'ha pogut dur a terme. Els motius pels quals pot fallar són similars a l'anterior, si bé en el cas de les carpetes també pot fallar si no està buida. Només es pot esborrar una carpeta usant aquest mètode si dins seu no hi té res (ni altres carpetes ni fitxers).

Per provar l'exemple que es mostra tot seguit de manera que es pugui veure com funcionen aquests mètodes, novament, cal una organització prèvia del sistema de fitxers. Primer, assegureu-vos que a l'arrel de la unitat "C:" no hi ha cap carpeta anomenada "Temp" i executeu el programa. Tot fallarà, ja que les rutes són incorrectes (no existeix "Temp"). Un cop fet, creeu la carpeta "Temp", i a dins seu creeu un nou document anomenat "Document.txt" (que pot estar buit si voleu). Ara, en executar el programa, s'haurà creat una nova carpeta anomenada "Fotos". Si sense tocar la carpeta "Temp" executeu el programa per tercer cop, ara tot s'haurà esborrat.

```
1  package unitat6.apartat1.exemples;
2  import java.io.File;
3  public class FileGestioElements {
4    public static void main(String[] args) {
5      FileGestioElements programa = new FileGestioElements();
6      programa.inici();
7    }
8    public void inici() {
9      File fotos = new File ("C:/Temp/Fotos");
10     File document = new File ("C:/Temp/Document.txt");
11     boolean resultat= fotos.mkdir();
12     System.out.println("Creada carpeta " + fotos.getName() + "? " + resultat);
13     if (!resultat) {
14       boolean delCarpeta = fotos.delete();
15       System.out.println("Esborrada carpeta " + fotos.getName() + "? " +
             delCarpeta);
16       boolean delFitxer= document.delete();
17       System.out.println("Esborrat fitxer " + document.getName() + "? " +
             delFitxer);
18     }
19   }
20 }
```

Des del punt de vista d'un sistema operatiu, l'operació de "moure" un fitxer o carpeta no és més que canviar el seu nom, des de la seva ruta original fins a una nova ruta destinació. Per fer això també hi ha un mètode, ja una mica més complex `boolean renameTo(File desti)`.

`boolean renameTo(File desti)` és un mètode amb un nom una mica enganyós ("reanomenar", en anglès), ja que la seva funció real no és simplement canviar el nom d'un fitxer o carpeta, sinó canviar la ubicació completa. El mètode s'invoca sobre la variable amb la ruta que es considera el fitxer origen, i el paràmetre d'entrada és la ruta destinació. En aquest cas, el fitxer o carpeta associat a la ruta origen ha d'existir en el moment d'invocar el mètode, però la destinació no. Com en casos anteriors, avalua `true` o `false` segons si la tasca s'ha pogut dur a terme correctament o no (la ruta origen i destinació són correctes, no existeix ja un fitxer amb aquest nom a la destinació, etc.). Noteu que, en el cas de carpetes, és possible moure-les tot i que continguin fitxers. Tota l'estructura jeràrquica de carpetes es desplaça.

Un cop més, cal una certa preparació prèvia en l'execució de l'exemple que permet veure com funciona amb més detall aquest mètode. Dins d'una carpeta anomenada "Temp" a la unitat "C:", creeu una carpeta anomenada "Media" i una altra anomenada "Fotos". Dins de la carpeta "Fotos", genereu dos documents (que poden estar buits), anomenats "Document.txt" i "Fotos.txt". Després d'executar el programa, observeu com la carpeta "Fotos" s'ha mogut i ha canviat de nom, però manté dins seu el fitxer "Fotos.txt". El fitxer "Document.txt" s'ha mogut fins a la carpeta "Temp".

```java
package unitat6.apartat1.exemples;
import java.io.File;
public class FileMoure {
  public static void main(String[] args) {
    FileMoure programa = new FileMoure();
    programa.inici();
  }
  public void inici() {
    File origenCarpeta = new File("C:/Temp/Fotos");
    File destiCarpeta = new File("C:/Temp/Media/Fotografies");
    File origenDocument = new File("C:/Temp/Media/Fotografies/Document.txt");
    File destiDocument = new File("C:/Temp/Document.txt");
    boolean resultat = origenCarpeta.renameTo(destiCarpeta);
    System.out.println("S'ha mogut i reanomenat la carpeta? " + resultat);
    resultat = origenDocument.renameTo(destiDocument);
    System.out.println("S'ha mogut el document? " + resultat);
  }
}
```

Com es pot veure a l'exemple, aquest mètode també serveix, implícitament, per reanomenar fitxers o carpetes. Si l'element final de les rutes origen i destinació són diferents, el nom de l'element, sigui fitxer o carpeta, canviarà. Per simplement reanomenar un element sense moure'l de lloc, només cal que la seva ruta pare sigui exactament la mateixa. El resultat és que l'element de la ruta origen "es mou" a la mateixa carpeta on està ara, però amb un nom diferent.

Per exemple, el codi següent reanomena un fitxer anomenat "Document.txt", ubicat a la carpeta "C:/Temp", a "Reanomenat.txt":

```java
package unitat6.apartat1.exemples;
import java.io.File;
public class FileReanomenar {
  public static void main(String[] args) {
    FileReanomenar programa = new FileReanomenar();
    programa.inici();
  }
  public void inici() {
    File rutaOrigen = new File("C:/Temp/Document.txt");
    //Reanomenar: mateixa ruta pare però diferent nom final
    String nouNom = rutaOrigen.getParent() + File.separator + "Reanomenat.txt";
    File rutaDesti = new File(nouNom);
    rutaOrigen.renameTo(rutaDesti);
  }
}
```

La creació de fitxers de moment es deixarà pendent, ja que està vinculada al tractament de les dades que ha de contenir, i per tant va més enllà de la mera gestió d'una ruta.

**Repte 1**. Creeu un programa que llegeixi un text pel teclat corresponent a la ruta d'un fitxer existent al vostre ordinador. Canvieu el nom d'aquest fitxer de manera que s'elimini la seva extensió, si en té (per exemple, si el fitxer es diu "Document.txt", passarà a dir-se "Document"). Per dur a terme aquesta tasca haureu de fer memòria de les transformacions sobre cadenes de text.

### Llistat

Finalment, només en el cas de les carpetes, és possible consultar quin és el llistat de fitxers i carpetes que conté.

`File[ ] listFiles()` avalua un *array* de `File` on s'enumeren tots els elements continguts a la carpeta indicada pel text usat en inicialitzar la variable de tipus `File`. Perquè s'executi correctament, el text de la ruta ha d'indicar forçosament una carpeta. En cada posició d'aquest *array* hi ha el nom de la ruta de cadascun dels fitxers. La mida de l'*array* serà igual al nombre d'elements que conté la carpeta. Si la mida és 0, el valor retornat és `null` i tota operació posterior sobre la variable resultant serà errònia. L'ordre dels elements pot ser qualsevol, no es pot garantir res (al contrari que a l'explorador de fitxers del sistema operatiu, no s'ordena automàticament primer carpetes i després fitxers, o alfabèticament).

La particularitat que potser fa una mica complex aquest mètode, a primera vista, és el fet que avaluï un *array* de `File`. Atès que la classe `File` és un tipus compost, amb vista a treballar amb ell només cal que recordeu com es treballa amb un *array* de cadenes de text, ja que és el mateix des del punt de vista d'accés a les posicions i invocació de mètodes sobre els valors emmagatzemats.

El millor, però, es veure'n un exemple. Abans d'executar-lo, creeu una carpeta anomenada "Temp" a l'arrel de la unitat "C:". Dins seu creeu o copieu qualsevol quantitat de carpetes o fitxers.

```java
package unitat6.apartat1.exemples;
import java.io.File;
public class FileLlistarElements {
  public static void main(String[] args) {
    FileLlistarElements programa = new FileLlistarElements();
    programa.inici();
  }
  public void inici() {
    File carpeta = new File("C:/Temp");
    File[] arrayElements = carpeta.listFiles();
    System.out.println("El contingut de " + carpeta.getAbsolutePath() + " és:")
        ;
    //Per recórrer un array, s'usa un bucle
    for (int i = 0; i < arrayElements.length; i++) {
      File f = arrayElements[i];
      if (f.isDirectory()) {
       System.out.print("[DIR] ");
      } else {
       System.out.print("[FIT] ");
      }
      System.out.println(f.getName());
    }
  }
}
```

**Repte 2**. Realitzeu un programa que llegeixi des del teclat el text associat a la ruta a una carpeta existent dins del vostre ordinador. Esborrar tots els elements que hi ha dins, independentment que siguin fitxers o carpetes. Tingueu en compte que per poder eliminar una carpeta cal abans buidar-la de tot el seu contingut (en la qual hi pot haver alhora tant fitxers o carpetes... i així indefinidament fins a arribar a una carpeta que només té fitxers). Pista: Per resoldre aquest problema és molt més senzill plantejar una solució recursiva.

## 1.2 Solució dels reptes proposats

### Repte 1

```java
package unitat6.apartat1.reptes;
import java.io.File;
import java.util.Scanner;
public class CanviaNom {
  public static void main(String[] args) {
    CanviaNom programa = new CanviaNom();
    programa.inici();
  }
  public void inici() {
    File rutaFitxer = llegirNomFitxer();
    //Cal comprovar si realment existeix i si és un fitxer
    if (rutaFitxer.isFile()) {
      File novaRuta = treureExtensio(rutaFitxer);
      //Es canvia el nom
      rutaFitxer.renameTo(novaRuta);
      System.out.println("Nom canviat de " + rutaFitxer + " a " + novaRuta);
    } else {
      System.out.println("Aquest fitxer no existeix!");
    }
  }
  /** Pregunta a l'usuari el nom del fitxer, i d'aquest obté una ruta.
   *
   * @return La ruta associada al text que ha escrit l'usuari.
   */
  public File llegirNomFitxer() {
    Scanner lector = new Scanner(System.in);
    System.out.println("Escriu el nom d'una ruta en un fitxer existent: ");
    String nomFitxer = lector.nextLine();
    //Fixeu-vos com és possible fer return d'una variable de tipus File,
    //igual que es faria per un enter o una cadena de text.
    File f = new File(nomFitxer);
    return f;
  }
  /** Donada una ruta, en crea una de nova igual, però sense extensió (.xxx)
   *
   * @param original Ruta original que cal transformar
   * @return La ruta sense extensió
   */
  public File treureExtensio(File original) {
    String nom = original.getName();
    String pare = original.getParent();
    //Cerquem el darrer punt, per trobar l'extensió
    int posicioPunt = nom.lastIndexOf(".");
    if (posicioPunt >= 0) {
      //eliminem el que hi ha darrera del punt
      String nouNom = nom.substring(0, posicioPunt);
      //Es fa el text per a la nova ruta, sense extensió
      String nouText = pare + File.separator + nouNom;
      File novaRuta = new File(nouText);
      return novaRuta;
    } else {
      //Si no té extensió, es deixa tot igual...
      return original;
    }
  }
}
```

## Repte 2

```java
package unitat6.apartat1.reptes;
import java.io.File;
import java.util.Scanner;
public class NetejarCarpeta {
  public static void main(String[] args) {
    NetejarCarpeta programa = new NetejarCarpeta();
    programa.inici();
  }
  public void inici() {
    File rutaCarpeta = llegirRutaCarpeta();
    netejarCarpeta(rutaCarpeta);
  }
  /** Pregunta a l'usuari el nom de la carpeta, i d'aquest obté una ruta.
   *
   * @return La ruta associada al text que ha escrit l'usuari.
   */
  public File llegirRutaCarpeta() {
    //Un tipus compost es pot deixar sense inicialitzar posant "null"
    File f = null;
    boolean preguntar = true;
    Scanner lector = new Scanner(System.in);
    while (preguntar) {
      System.out.println("Escriu el nom d'una ruta en una carpeta: ");
      String nomCarpeta = lector.nextLine();
      f = new File(nomCarpeta);
      if (f.isDirectory()) {
        preguntar = false;
      } else {
        System.out.println("Aquesta carpeta no existeix...");
      }
    }
    return f;
  }
  /** Donada una ruta associada a una carpeta, esborra tot el seu contingut.
   *
   * @param ruta Ruta de la carpeta que cal netejar
   */
  public void netejarCarpeta(File ruta) {
    File[] elements = ruta.listFiles();
    //Cal mirar tots els elements:
    for (int i = 0; i < elements.length; i++) {
      if (elements[i].isFile()) {
        //Si es un fitxer, s'esborra.
        elements[i].delete();
      } else if (elements[i].isDirectory()) {
        //Si és una carpeta, cal buidar—la primer, o sigui: invocar
            netejarCarpeta!
        netejarCarpeta(elements[i]);
        //Un cop buida, es pot esborrar correctament
        elements[i].delete();
      }
    }
  }
}
```

## 2. Tractament bàsic de dades

Normalment, les aplicacions que fan servir fitxers no estan centrades en la gestió del sistema de fitxers del vostre ordinador. L'objectiu principal d'usar fitxers és poder emmagatzemar-hi dades, de manera que entre diferents execucions del programa, fins i tot en diferents equips, és possible recuperar-les. El cas més típic és un editor de documents, que mentre s'executa s'encarrega de gestionar les dades relatives al text que esteu escrivint, però en qualsevol moment podeu desar-lo en un fitxer per poder recuperar aquest text en qualsevol moment posterior, i afegir-ne de nou, si escau. El fitxer amb les dades del document el podeu obrir tant en l'editor del vostre ordinador com en el d'un altre company.

Per saber com tractar les dades d'un fitxer dins un programa, cal tenir molt clar com s'hi estructuren. Dins un fitxer es poden emmagatzemar tota mena de valors de qualsevol tipus de dades. La part més important és que aquests valors s'emmagatzemen en forma de seqüència, un rere l'altre. Per tant, com aviat veureu, la manera més habitual de tractar fitxers és seqüencialment, de manera semblant a com es fa per llegir-les del teclat, mostrar-les per pantalla o recórrer les posicions d'un *array*.

Un exemple d'accés seqüencial a dades: el missatge d'un telègraf.
Font: Wallace Study-Telegraph

S'anomena **accés seqüencial** al tractament d'un conjunt d'elements de manera que només és possible accedir-hi d'acord al seu l'ordre d'aparició. Per poder tractar un element, cal haver tractat tots els elements anteriors.

Java, juntament amb d'altres llenguatges de programació, diferencia entre dos tipus de fitxers segons com es representen els valors emmagatzemats dins un fitxer.

En els fitxers **orientats a caràcter**, les dades es representen com una seqüència de cadenes de text, on cada valor es diferencia de l'altre usant un delimitador. En canvi, en els fitxers **orientats a byte**, les dades es representen directament d'acord al seu format en binari, sense cap separació.

Si bé, a grans trets, l'accés seqüencial a les dades segueix el mateix esquema independentment del tipus de fitxer, cada cas té certes particularitats que fa interessant estudiar-los de manera diferenciada.

## 2.1 Accés seqüencial a fitxers orientats a caràcter

Un fitxer orientat a caràcter no és més que un document de text, com el que podríeu generar amb qualsevol editor de text simple. Els valors estan emmagatzemats segons la seva representació en cadena de text, exactament en el mateix format que heu usat fins ara per entrar dades des del teclat. De la mateixa manera, els diferents valors es distingeixen en estar separats entre ells amb un delimitador, que per defecte és qualsevol conjunt d'espais en blanc o salt de línia. Tot i que aquests valors es puguin distribuir en línies de text diferents, conceptualment, es pot considerar que estan organitzats un rere l'altre, seqüencialment, com les paraules a la pàgina d'un llibre.

El següent podria ser el contingut d'un fitxer orientat a text on hi ha deu valors de tipus real:

```
1,5 0,75 −2,35 18 9,4 3,1416 −15,785
−200,4 2,56 9,3785
```

I el d'un fitxer amb 4 valors de tipus `String` ("Hi", "havia", "una" i "vegada..."):

```
Hi havia una vegada...
```

El principal avantatge d'un fitxer d'aquest tipus és, doncs, que resulta molt senzill inspeccionar el seu contingut, generar-los d'acord a les vostres necessitats o representar quin ha de ser el seu contingut dins del text d'un altre document, com tot just s'acaba de fer.

### 2.1.1 Inicialització

Per al cas dels fitxers orientats a caràcter, cal usar dues classes diferents segons si el que es vol és llegir o escriure dades a un fitxer. Normalment, això no és gaire problemàtic, ja que en un bloc de codi donat només es duran a terme operacions de lectura de dades o d'escriptura sobre un mateix fitxer, però no els dos tipus d'operació alhora. Normalment, es llegeixen les dades a l'inici del programa, igual que es pot demanar escriure-les usant el teclat, i es desen quan es disposa dels resultats que cal calcular, com quan correspondria mostrar-les per pantalla.

> Per tractar de manera senzilla fitxers orientats a caràcter, Java ofereix les classes `Scanner`, pertanyent al *package* `java.util`, i `PrintStream`, pertanyent al *package* `java.io`.

Com en el cas de qualsevol altra classe que us calgui usar dins el vostre programa, abans de poder realitzar cap operació és imprescindible inicialitzar-la, de manera que es disposi d'una variable sobre la qual es poden invocar els mètodes corresponents.

### Scanner

Recordeu que per usar la classe Scanner us cal importar-la des del *package* java.util.

La classe que permet dur a terme la lectura de dades des d'un fitxer orientat a caràcter és exactament la mateixa que permet llegir dades des del teclat. Al cap i a la fi, els valors emmagatzemats en els fitxers d'aquest tipus es troben exactament en el mateix format que heu usat fins ara per entrar informació als vostres programes: una seqüència de cadenes de text. L'única diferència és que aquests valors no es demanen a l'usuari durant l'execució, sinó que són escrits tots amb anterioritat.

Per tal de processar dades des d'un fitxer, la classe `Scanner` permet usar una ruta com a paràmetre per inicialitzar-la. El codi següent mostra l'esquema bàsic d'inicialització d'una variable d'aquest tipus de manera que es llegeixin dades des d'un fitxer.

```
1 import java.io.File;
2 import java.util.Scanner;
3 ...
4 Scanner lectorFitxer = new Scanner(File f);
```

El paràmetre `f` es correspon a qualsevol variable de tipus `File` prèviament inicialitzada correctament. En fer-ho, cal tenir en compte que la classe `Scanner` també es comporta com un tipus compost. Per tant, en el cas de treballar amb diferents fitxers, caldrà inicialitzar diferents variables.

Per exemple, per inicialitzar una variable de tipus `Scanner` de manera que permeti llegir dades des del fitxer ubicat a la ruta "C:\Programes\Unitat 6\Apartat 2\Exemples\Document.txt", caldria fer:

```
1 import java.io.File;
2 import java.util.Scanner;
3 ...
4 File fitxer = new File("C:\Programes\Unitat 6\Apartat 2\Exemples\Document.txt")
      ;
5 Scanner lectorFitxer = new Scanner(fitxer );
```

### PrintStream

Per usar la classe PrintStream us cal importar-la des del *package* java.io.

Per escriure dades a un fitxer, la classe que cal usar és `PrintStream`. Igual com amb `Scanner`, per inicialitzar-la correctament només cal indicar la ruta del fitxer amb el qual treballar. En aquest cas, en tractar-se d'escriptura, la ruta pot indicar un fitxer que pot existir o no dins el sistema. En el cas d'indicar un fitxer que no existeixi, se'n crearà un de nou. En el cas que ja existeixi, les dades contingudes dins del fitxer es perden totalment i aquest queda en blanc, amb mida igual a 0.

```
1 import java.io.File;
2 import java.io.PrintStream;
3 ...
4 PrintStream escriptorFitxer = new PrintStream(File f);
```

En aquest cas, per inicialitzar una variable de tipus `PrintStream` de manera que permeti escriure dades a un nou fitxer ubicat a la ruta "C:\Programes\Unitat

6\Apartat 2\Exemples\Document.txt", caldria fer el següent. Com podeu veure, l'esquema és pràcticament idèntic al de `Scanner`. Però en aquest cas es prepara el fitxer per a l'escriptura.

```
1  import java.io.File;
2  import java.io.PrintStream;
3  ...
4  File fitxer = new File("C:\Programes\Unitat 6\Apartat 2\Exemples\Document.txt")
       ;
5  PrintStream escriptorFitxer = new PrintStream(fitxer );
```

Alerta, però, ja que el tractament seqüencial de dades mitjançant `PrintStream` només permet treballar amb fitxers nous. Si la ruta especificada es refereix a un fitxer que ja existeix, el seu contingut serà eliminat totalment.

### 2.1.2 Control d'errors

Quan es realitzen operacions de lectura i escriptura sobre fitxers poden passar moltes situacions anòmales. Des que, simplement, el fitxer no existeixi a la ruta especificada, fins a que hi hagi algun error en el sistema d'entrada/sortida de l'ordinador o el fitxer estigui corromput i l'operació es vegi interrompuda inesperadament. El tractament de dades amb fitxers és un procés molt més delicat del que aparenta inicialment.

El llenguatge Java considera imprescindible tenir en consideració totes aquestes circumstàncies especials, a les quals anomena *excepcions*. Per controlar-les, disposa d'un mecanisme una mica especial que és obligatori usar sempre, ja que en cas de no fer-ho s'indicarà que hi ha un error de compilació. Per tant, abans de continuar, és necessari conèixer com funciona el mecanisme de control d'excepcions, ja que en cas contrari no us serà possible dur a terme cap programa que tracti fitxers. Aquesta secció no té cap intenció d'explicar-ho en la seva totalitat, només pretén mostrar-vos el mínim imprescindible per poder seguir endavant, i treballar amb fitxers.

Per començar, i veure que no és possible compilar un programa que no controli excepcions en treballar amb fitxers, proveu el programa següent al vostre entorn de treball. D'acord al seu codi, aquest hauria d'obrir un fitxer anomenat "Document.txt" que hi ha la carpeta de treball de l'aplicació.

```
1  package unitat6.apartat2.exemples;
2  import java.io.File;
3  import java.util.Scanner;
4  public class ObrirFitxer {
5      public static void main(String[] args) {
6        ObrirFitxer programa = new ObrirFitxer();
7        programa.inici();
8    }
9    public void inici() {
10     File f = new File("Document.txt");
11     Scanner lector = new Scanner(f);
```

```
12      System.out.println("Fitxer obert correctament.");
13    }
14  }
```

En fer-ho, podreu veure com el compilador de Java es queixa d'un error d'aquest estil.

```
1  unreported exception java.io.FileNotFoundException; must be caught or declared
       to be thrown
```

El compilador ha detectat que, donada aquesta inicialització, pot succeir una excepció en el procés d'entrada / sortida, però el codi del programa no l'està controlant (*unreported exception ...; must be caught*, "excepció no declarada ni controlada", en anglès). Concretament, avisa que pot donar-se el cas d'intentar obrir un fitxer que no existeix (*FileNotFound*, "fitxer no trobat", en anglès ). Per evitar aquest error, totes les instruccions vinculades a treballar amb fitxers han d'estar dins del que s'anomena una sentència `try/catch`.

> Una sentència **try/catch** es composa de dos blocs de codi. Un per indicar les instruccions que cal anar executant mentre no hi ha excepcions (bloc `try`), i un altre en el moment que se'n produeixi qualsevol (bloc `catch`).

La sintaxi d'aquest bloc de codi té una certa similitud amb una sentència `if/else`, des de la perspectiva que es creen dos blocs de codi. Un d'ells s'executa només si es dóna una condició, i l'altre en cas contrari. La sintaxi és la següent:

```
1  try {
2    //Codi amb les instruccions per fer operacions d'entrada / sortida.
3    //...
4  } catch (Exception e) {
5    //Codi en el cas que hi hagi un error en qualsevol operació anterior
6    //...
7  }
```

La sentència catch captura l'excepció, per tractar-la. Font: Rdikeman

La principal diferència amb un bloc `if/else` és que no hi ha cap condició lògica a comprovar per decidir quin bloc cal executar. Dins el bloc `try` s'escriuen normalment les instruccions del programa. Si en qualsevol moment durant l'execució es produeix una excepció en alguna operació d'entrada / sortida, les instruccions dins d'aquest bloc deixen d'executar-se immediatament, i el flux de control del programa salta a la primera instrucció del bloc `catch`. Llavors, s'executen totes les instruccions dins d'aquest bloc. Si no es produeix cap excepció dins el bloc `try`, mai s'executa el bloc `catch`. Per tant, dins del bloc `catch` es posen les instruccions per tractar que ha succeït un error.

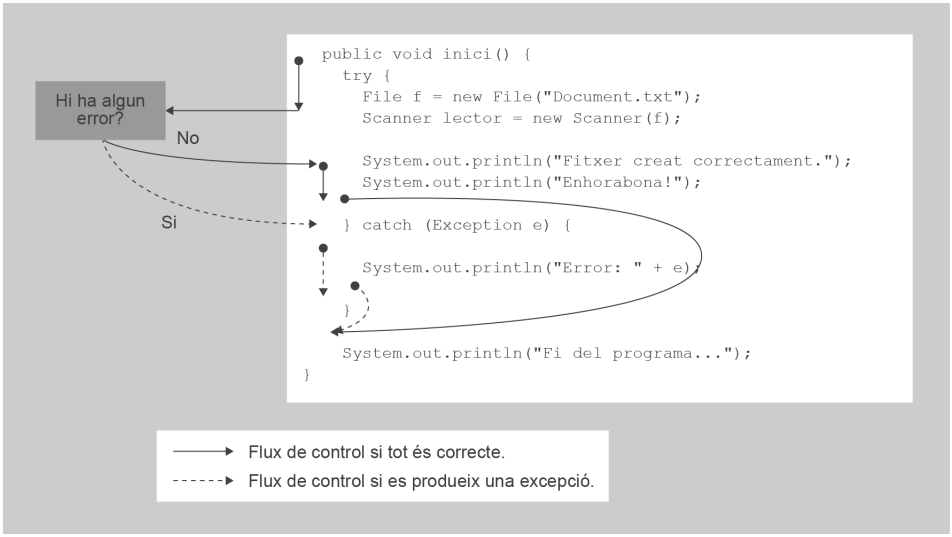
Partint de la traducció del nom de la sentència de l'anglès, bàsicament diu "intenta (*try*) fer això, i si mai hi ha cap excepció durant el procés, la captures (*catch*) i la tractes."

La manera més gràfica de veure aquest fet és directament amb un exemple de codi. Escriviu el programa següent al vostre entorn de treball. Primer de tot, veureu que ja no hi ha cap error de compilació, ja que la instrucció per inicialitzar `Scanner` ara està dins una sentència `try/catch`. Un cop escrit, executeu-lo sense que hi

hagi un fitxer amb aquest nom, i en el cas contrari. Veureu com el resultat del programa serà diferent. La figura 2.1 mostra un esquema del flux de control del programa en cada cas.

```java
package unitat6.apartat2.exemples;
import java.io.File;
import java.util.Scanner;
public class ObrirFitxer {
  public static void main (String[] args) {
    ObrirFitxer programa = new ObrirFitxer();
    programa.inici();
  }
  public void inici() {
    try {
      //Bloc "try": operacions d'entrada / sortida
      //S'intenta obrir un fitxer per a lectura
      File f = new File("Document.txt");
      Scanner lector = new Scanner(f);
      //El fitxer existeix. Tot correcte
      System.out.println("Fitxer obert correctament.");
      System.out.println("Enhorabona!");
    } catch (Exception e) {
      //Bloc "catch": Tractament d'errors
      //S'ha produït una excepció en algun lloc del bloc "try"!
      System.out.println("Error: " + e);
    }
    //Les instruccions fora de la sentència "try/catch" sempre s'executen
    //Igual que les que hi ha després d'una sentència "if/else"
    System.out.println("Finalització del programa...");
  }
}
```

**Figura 2.1.** Esquema de flux de control davant d'un bloc try-catch.




No s'haurien de dur a terme operacions d'entrada / sortida dins un bloc `catch`

Si us fixeu en l'exemple, aquest també mostra una particularitat de la sintaxi d'aquesta sentència. A l'inici del bloc `catch` hi ha una variable declarada, amb l'identificador **e**, de manera molt semblant a com es faria amb el paràmetre d'un mètode: **(Exception e)**. Aquesta variable pertany a la classe `Exception`, que està al *package* `java.lang` (i per tant, no cal importar).

Les variables de la classe **Exception** contenen informació sobre l'excepció que ha desencadenat el bloc `catch` on pertanyen. Java s'encarrega d'inicialitzar-les correctament sempre que es produeix alguna excepció.

Per tant, només us heu de preocupar que ja disposeu d'aquesta variable per poder descriure quin error s'ha produït exactament. Aquesta descripció és sempre una cadena de text, que pot ser impresa de la mateixa manera que es faria amb una variable de tipus `String`. Quan es produeix una excepció, si el codi afectat és dins el mètode `inici()`, normalment n'hi ha prou a mostrar el missatge d'error per pantalla.

Ara bé, si el codi és dins un altre mètode auxiliar, i aquest a la seva declaració indica que té un paràmetre de sortida, cal tenir en compte que dins el codi és sempre obligatori invocar la sentència `return`, tant si ha succeït una excepció com si no. En cas contrari, Java dóna un error de compilació. Donat aquest fet, caldrà que dins del bloc `catch` s'invoqui `return` amb algun valor que es consideri no vàlid, i que sigui *a posteriori* quan es comprovi si el valor retornat en invocar aquest mètode és vàlid o no.

A mesura que aneu veient nous aspectes del tractament de fitxers al llarg de l'apartat, es veurà més clar el funcionament de la sentència `try/catch` i el tractament d'excepcions, especialment pel que fa a la detecció d'errors a mètodes auxiliars.

### 2.1.3 Lectura de dades

Des del punt de vista d'instruccions que cal executar, la lectura seqüencial de dades des d'un fitxer orientat a caràcter és pràcticament igual a llegir dades des del teclat. Els valors es van llegint de manera ordenada, des de l'inici de la seqüència que representen fins al final. De fet, ara que ja coneixeu què és una classe, i que aquestes es poden comportar com tipus compostos, com inicialitzar-les sobre una variable i com invocar mètodes correctament a partir d'aquesta variable, és un bon moment per recapitular sobre el funcionament de la classe `Scanner`, amb vista a mostrar com dur a terme la lectura de dades des d'un fitxer.

Les operacions de lectura usant `Scanner` es duen a terme mitjançant la invocació dels mètodes que ofereix, resumits a la taula 2.1, els quals avaluen el valor tot just llegit.

**Taula 2.1.** Mètodes de lectura de dades de la classe "Scanner"

| Mètode | Tipus de dada llegida |
|---|---|
| `nextByte()` | `byte` |
| `nextShort()` | `short` |
| `nextInt()` | `int` |
| `nextLong()` | `long` |
| `nextFloat()` | `float` |

**TAULA 2.1** (continuació)

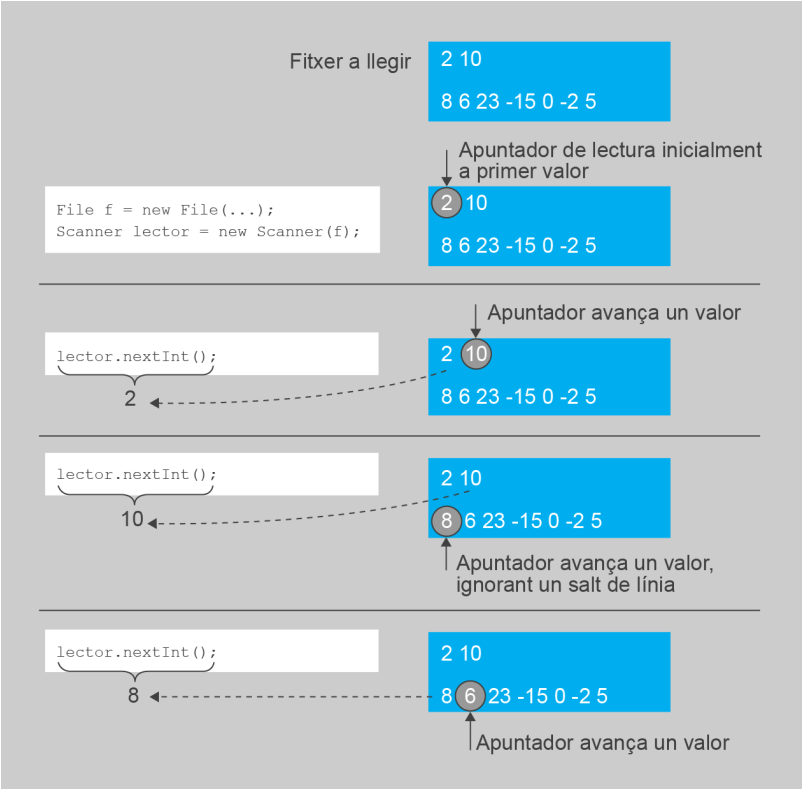
| Mètode | Tipus de dada llegida |
|---|---|
| `nextDouble()` | `double` |
| `nextBoolean()` | `boolean` |
| `next()` | `String` (una paraula) |
| `nextLine()` | `String` (una frase) |

Un cop s'ha llegit un valor, aquest no pot ser "rellegit".

En el cas d'un fitxer, la variable de tipus `Scanner` gestiona internament un apuntador que sempre indica sobre quin valor actuaran les operacions de lectura. Cada cop que es fa una lectura, l'apuntador avança automàticament fins al valor següent dins el fitxer, i no hi ha cap manera de fer-lo recular. Quan una variable de tipus `Scanner` s'inicialitza, aquest apuntador es troba en el primer valor dins el fitxer. Aquest procés es va repetint fins que s'han llegit tants valors com es desitja. La figura 2.2 mostra un petit esquema d'aquest procés, recalcant com avança l'apuntador a l'hora de fer operacions de lectura sobre un fitxer que conté valors de tipus enter.

Els diferents valors es distingeixen en separar-se per espais o salts de línia.

**FIGURA 2.2.** Lectura seqüencial de valors en un fitxer orientat a caràcter




Un cop s'ha mostrat com funciona l'apuntador intern en llegir valors des d'un fitxer, val la pena tornar a repassar la diferència entre el mètode `next()` i `nextLine()`, ja que ambdós avaluen una cadena de text. El primer només llegeix una paraula individual, considerant "paraula" un conjunt de caràcters que no estan separats per espais o salts de línia. Aquest conjunt de caràcters poden ser tant paraules, tal com es trobarien en un diccionari ("casa", "hola", etc.), com valors numèrics expressats en format text ("2", "3,14", "1024", etc.). En canvi, `nextLine()` llegeix una frase completa. Concretament, el que llegeix és tot el text

que hi ha entre la posició actual de l'apuntador i el proper salt de línia. L'apuntador es posa llavors a l'inici de la línia següent.

El sistema operatiu diu que un fitxer on s'estan fent operacions d'entrada / sortida està en ús o obert. Si no, es diu que està tancat.

Un cop s'ha finalitzat la lectura de les dades del fitxer, ja siguin totes o només una part, i ja no cal llegir-ne més, és imprescindible executar un mètode especial anomenat `close()`. Aquest indica al sistema operatiu que el fitxer ja no està essent utilitzat pel programa. Això és molt important, ja que mentre un fitxer es considera en ús, el seu accés es pot veure limitat. Si no invoqueu `close()`, el sistema operatiu pot trigar un temps a adonar-se que el fitxer ja no es troba en ús tot i que el vostre programa ja hagi finalitzat. Sempre cal tancar el fitxer un cop heu finalitzat totes les operacions d'entrada / sortida.

El programa següent mostra un exemple de com llegir deu valors enters des d'un fitxer anomenat "Enters.txt", ubicat a la seva carpeta de treball. Aquest fitxer de text l'haureu de crear vosaltres, garantint que conté exactament 10 valors enters, separats entre ells per espais en blanc o salts de línia. Fixeu-vos com totes les instruccions relatives a la lectura del fitxer, les operacions d'entrada / sortida, són dins una sentència `try/catch`. També, fixeu-vos que en acabar la lectura, s'invoca el mètode **close)**.

```
1  package unitat6.apartat2.exemples;
2  import java.io.File;
3  import java.util.Scanner;
4  public class LlegirEnters {
5    public static final int NUM_VALORS = 10;
6    public static void main (String[] args) {
7      LlegirEnters programa = new LlegirEnters();
8      programa.inici();
9    }
10   public void inici() {
11       try {
12         //S'intenta obrir el fitxer
13         File f = new File("Enters.txt");
14         Scanner lector = new Scanner(f);
15         //Si s'executa aquesta instrucció, s'ha obert el fitxer
16         for (int i = 0; i < NUM_VALORS; i ++) {
17           int valor = lector.nextInt();
18           System.out.println("El valor llegit és: " + valor);
19         }
20         //Cal tancar el fitxer
21         lector.close();
22       } catch (Exception e) {
23         //Excepció!
24         System.out.println("Error: " + e);
25       }
26   }
27 }
```

Una diferència important a l'hora de tractar amb fitxers respecte a llegir dades del teclat és que les operacions de lectura no són producte d'una interacció directa amb l'usuari, que és qui escriu les dades. Només es pot treballar amb les dades que hi ha al fitxer i res més. Això té dos efectes sobre el procés de lectura.

Els mètodes next() o nextLine() mai poden donar error, ja que els valors en un fitxer orientat a caràcter sempre es poden considerar cadenes de text.

D'una banda, recordeu que quan es duu a terme el procés de lectura d'una seqüència de valors, sempre cal anar amb compte d'usar el mètode adient al tipus de valor que s'espera que vingui a continuació. Saber quin tipus de valor s'espera és quelcom que heu decidit vosaltres a l'hora de fer el programa, però res garanteix que, en escriure el fitxer de text, no us hagueu equivocat sense voler, com pot passar

quan escriviu dades mitjançant el teclat. Ara bé, en aquest cas, com que opereu amb fitxers, i no pel teclat, no hi ha l'opció de demanar simplement a l'usuari que el torni a escriure. Per tant, el programa hauria de dir que s'ha produït un error ja que el fitxer no té el format correcte i finalitzar el procés de lectura.

D'altra banda, també cal que controleu que mai llegiu més valors dels que hi ha disponibles per llegir. En un cas com aquest, a l'hora de fer entrada de dades pel teclat, el programa simplement es bloquejava, esperant que l'usuari escrivís nous valors. Quan opereu amb fitxers això no succeeix. Intentar llegir un nou valor quan l'apuntador ja ha superat el darrer disponible es considera erroni. Un cop s'arriba al final del fitxer, ja no queda més remei que invocar `close()` i finalitzar la lectura.

> Si el tipus llegit no es correspon al mètode invocat (per exemple, s'ha invocat el mètode `nextInt()` quan l'apuntador és sobre un real) o s'intenta fer una lectura quan l'apuntador ja ha superat tots els valors del fitxer, es produirà una excepció.

Per veure-ho, proveu d'executar l'exemple anterior amb fitxer on, o bé algun dels valors continguts no és de tipus enter, o el nombre de valors emmagatzemats és inferior a deu.

**Repte 1**. Feu un programa que llegeixi successivament 15 valors de tipus real des d'un fitxer anomenat "Reals.txt". Aquest fitxer el podeu fer vosaltres mateixos. El programa ha de mostrar quin dels valors dins el fitxer és el més gran.

### 2.1.4 Escriptura de dades

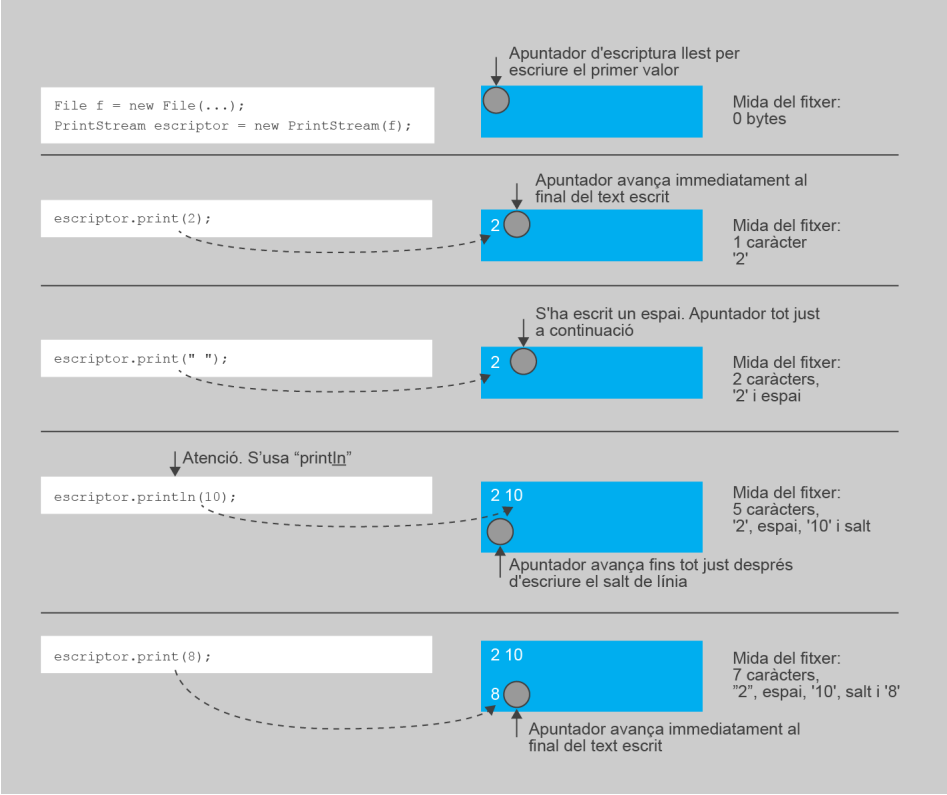
L'escriptura seqüencial de valors també és com usar una màquina d'escriure. Font: LjL

En aquest cas, l'escriptura seqüencial de dades en un fitxer orientat a caràcter és molt semblant a mostrar text per pantalla. Les dades es van escrivint una rere l'altra, amb l'opció de fer-ho consecutivament o separades per línies. Un cop s'ha escrit una dada, ja no hi ha marxa enrere. No és possible escriure informació abans o enmig de valors que ja estan escrits.

Recordeu que `PrintStream` només opera amb fitxer nous. Si s'escriu en un fitxer existent, se sobreescriu totalment.

En aquest cas, les operacions d'escriptura es porten a terme mitjançant els mètodes proporcionats per la classe `PrintStream`. Aquests són bàsicament dos, ara ja vells coneguts vostres, que accepten com a únic paràmetre qualsevol tipus primitiu de dades, o una cadena de text: `print(...)` i `println(...)`. La diferència és que el primer només escriu el valor usat com a paràmetre en format cadena de text, mentre que el segon també inclou automàticament un salt de línia, ja que "println" significa, "*print line*" ("imprimir línia", en anglès).

**Figura 2.3.** Escriptura seqüencial de valors en un fitxer orientat a caràcter




Com en el cas de la lectura, la classe `PrintStream` també gestiona un apuntador que li permet saber a partir de quina posició del text ha d'anar escrivint. Cada cop que s'invoca un dels seus mètodes d'escriptura, l'apuntador avança automàticament, i no és possible fer-lo recular. A efectes pràctics, aquest apuntador sempre està al final del fitxer, de manera que, a mesura que es van escrivint dades, el fitxer va incrementant la seva mida. La figura 2.3 mostra un esquema del funcionament de les escriptures.

Un aspecte que val la pena remarcar, i que preveu la figura, és que el procés d'escriptura no genera automàticament el delimitador entre valors. Per tant, és responsabilitat del vostre programa escriure els espais en blanc o salt de línia corresponents. En cas contrari, els valors quedaran enganxats, i en una posterior lectura s'interpretaran com un únic valor. Per exemple, si s'escriu el valor enter 2 i després el 4, sense preveure cap delimitador, en el fitxer s'haurà escrit al final el text del valor 24.

El codi següent serveix com a exemple d'un programa que escriu un fitxer dins la seva carpeta de treball anomenat "Enters.txt". Aquest conté 20 valors enters, de manera que, començant per l'1, cadascun sigui el doble de l'anterior. Compileu-lo i proveu-lo. Fixeu-vos que si ja existia un fitxer amb aquest nom, quedarà totalment sobreescrit. Després, podeu intentar llegir-lo amb el programa d'exemple anterior per llegir 10 valors enters i mostrar-los per pantalla.

```
1 package unitat6.apartat2.exemples;
2 import java.io.File;
3 import java.io.PrintStream;
4 public class EscriureEntersDobles {
5   public static final int NUM_VALORS = 20;
```

```
6   public static void main (String[] args) {
7     EscriureEntersDobles programa = new EscriureEntersDobles();
8     programa.inici();
9   }
10  public void inici() {
11      try {
12        //S'intenta obrir el fitxer
13        File f = new File("Enters.txt");
14        PrintStream escriptor = new PrintStream(f);
15        //S'escriu el primer valor, que no necessita un espai abans
16        int valor = 1;
17        escriptor.print(valor);
18        //Es van generant la resta de valors i escrivint
19        for (int i = 1; i < 20; i ++) {
20          //S'escriu abans com delimitador un espai en blanc
21          escriptor.print(" ");
22          //Es calcula i escriu el nou valor
23          valor = valor*2;
24          escriptor.print(valor);
25        }
26        //Cal tancar el fitxer
27        escriptor.close();
28        System.out.println("Fitxer escrit satisfactòriament.");
29      } catch (Exception e) {
30        //Excepció!
31        System.out.println("Error: " + e);
32      }
33  }
34  }
```

El procés d'actualització de les dades es coneix com *flush* ("tirar de la cadena", en anglès).

En acabar l'escriptura d'un fitxer també és imprescindible invocar el mètode **close()** sobre la variable de tipus `PrintStream`.

En el cas de les operacions d'escriptura, el tancament del fitxer encara és molt més important. Això es deu al fet que els sistemes operatius sovint porten a terme de manera diferida l'actualització de les dades. O sigui, el fet d'executar una instrucció d'escriptura no vol dir que immediatament aquestes dades ja estan escrites al fitxer. Pot passar un interval de temps variable entre les dues accions. Només a l'hora d'executar el mètode `close()` es força a què totes les dades pendents d'escriure s'actualitzin definitivament al fitxer.

**Repte 2**. Modifiqueu l'exemple de manera que els valors enters, en lloc d'estar tots escrits en una sola línia en el fitxer de text, estiguin dividits en 4 línies. A cada línia hi ha d'haver escrits 5 valors.

## 2.2 Aspectes importants de l'accés seqüencial

Un cop s'ha presentat com dur a terme l'accés seqüencial a un fitxer orientat a caràcter, i ja sabeu exactament com es materialitza en sentències del Java, val la pena exposar algunes situacions o problemàtiques que us podeu trobar i que poden complicar el procés. L'objectiu d'aquest apartat és indicar alguns esquemes o estratègies que podeu preveure per dur a terme correctament la lectura o l'escriptura d'un fitxer d'aquest tipus.

### 2.2.1 Detecció de final de seqüència

Tot i que el procés de lectura seqüencial de fitxers és força semblant a la lectura de dades des del teclat, una de les diferències substancials que cal tenir molt en compte és que el nombre de valors que es poden llegir està fixat pel propi contingut de fitxer. Mai es donarà el cas que el programa es quedi esperant que l'usuari escrigui text. Si s'intenta llegir valors d'un fitxer quan ja s'ha arribat al final de la seqüència i s'han esgotat els valors continguts, es produirà una excepció. Tot i que mitjançant el control d'excepcions es pot detectar i tractar aquest cas, en general, és millor generar codi dins els vostres programes que eviti arribar a aquesta situació. Per això, disposeu bàsicament de tres estratègies:

**1. Nombre de valors coneguts a priori**

El nombre de valors dins el fitxer ja és conegut dins del programa. Per exemple, si s'espera que el fitxer tingui 9 valors, el programa només hauria de fer 9 lectures i cap més. Si se'n fan menys, no passarà res, a part que deixarem valors sense llegir, però si se'n fan més, es produirà una excepció. Aquest és el cas de l'exemple de lectura que s'ha mostrat.

**2. Usar una marca de finalització**

El darrer valor no representa realment una dada que es vol tractar, sinó que és un valor especial que identifica el final de la seqüència. Aquest valor no es considera com part dels valors a tractar dins el fitxer, només és una marca de finalització. Ara bé, l'estratègia només val si hi ha valors que no es donaran mai dins la seqüència, ja que en cas contrari és impossible distingir entre un valor que indica la marca de finalització, o un que realment és vàlid i cal tractar. Per exemple, si se sap que tots els valors seran sempre entre -50 i 50, es podria usar un valor -100 com a marca de finalització, tal com mostra l'exemple de fitxer següent. Si el fitxer pogués preveure qualsevol valor, llavors aquesta estratègia no us serveix.

```
2 10 8 6 23 -15 0 -2 5 -100
```

El programa següent seria l'adaptació de l'exemple de lectura de dades des d'un fitxer anomenat "EntersMarca.txt" partint d'aquesta estratègia. Proveu-lo usant fitxers amb diferents valors escrits. En aquest cas, la marca de següent és el valor enter -100.

```
package unitat6.apartat2.exemples;
import java.io.File;
import java.util.Scanner;
public class LLegirMarcaFi {
  public static final int MARCA_FI = -100;
  public static void main (String[] args) {
    LLegirMarcaFi programa = new LLegirMarcaFi();
    programa.inici();
  }
  public void inici() {
      try {
        //S'intenta obrir el fitxer
        File f = new File("EntersMarca.txt");
        Scanner lector = new Scanner(f);
```

```
15          //Aquesta estratègia es basa en un semàfor
16          boolean llegir = true;
17          //Si s'executa aquesta instrucció, s'ha obert el fitxer
18          while (llegir) {
19            int valor = lector.nextInt();
20            if (valor == MARCA_FI) {
21              //Marca de finalització. Ja s'ha acabat la lectura
22              llegir = false;
23            } else {
24              //Encara no s'ha acabat. Tractar dada
25              System.out.println("El valor llegit és: " + valor);
26            }
27          }
28          //Cal tancar el fitxer
29          lector.close();
30        } catch (Exception e) {
31          //Excepció!
32          System.out.println("Error: " + e);
33        }
34    }
35  }
```

### 3. Nombre de valors indicat al propi fitxer

El primer valor de tota la seqüència pot ser un enter que identifica quants valors hi ha escrits a continuació, de manera que el programa mai farà més lectures de les establertes. Per exemple, en el fitxer següent, el primer valor indica que tot seguit hi ha 9 enters. Per tant, un programa no hauria d'intentar llegir-ne més dels indicats, o està garantit que es produirà una excepció.

```
1  9 2 10 8 6 23 −15 0 −2 5
```

El programa següent seria l'adaptació de l'exemple de lectura de dades des d'un fitxer anomenat "EntersMida.txt" partint d'aquesta estratègia. Proveu-lo usant fitxers amb diferents valors escrits.

```
1  package unitat6.apartat2.exemples;
2  import java.io.File;
3  import java.util.Scanner;
4  public class LlegirMida {
5    public static void main (String[] args) {
6      LlegirMida programa = new LlegirMida();
7      programa.inici();
8    }
9    public void inici() {
10       try {
11         //S'intenta obrir el fitxer
12         File f = new File("EntersMida.txt");
13         Scanner lector = new Scanner(f);
14         //Es llegeix la mida
15         int mida = lector.nextInt();
16         System.out.println("Hi ha " +  mida + " valors.");
17         //Si s'executa aquesta instrucció, s'ha obert el fitxer
18         for (int i = 0; i < mida; i ++) {
19           int valor = lector.nextInt();
20           System.out.println("El valor llegit és: " + valor);
21         }
22         //Cal tancar el fitxer
23         lector.close();
24       } catch (Exception e) {
25         //Excepció!
26         System.out.println("Error: " + e);
27       }
28   }
29 }
```

Evidentment, aquests mecanismes no són immunes als errors de programació, ja que pot succeir que us despisteu i que les coses no encaixin. Potser us heu oblidat d'escriure tots els valors que toca al fitxer, que per algun motiu s'hagi truncat o corromput i s'hagi perdut la marca de finalització, o que el nombre de valors escrit al principi no sigui correcte. Per això és important tractar les excepcions i mostrar l'error per pantalla, ja que en operar amb fitxers, mai es pot estar del tot segur que les dades que es volen llegir tindran un format correcte.

### 2.2.2 Processament de les dades llegides

Una de les particularitats del tractament seqüencial de dades des d'un fitxer és que, un cop s'inicialitza la variable a usar per accedir-hi, les dades només es poden llegir una rere l'altra, però no és possible recular. L'apuntador del fitxer només avança cada cop que es fa una operació, però no pot retrocedir. Això vol dir que s'hauria de plantejar el processament de les dades de manera que amb un únic recorregut el programa ja sigui possible de fer la tasca en qüestió. Això normalment porta a dos plantejaments possibles.

**1. Tractament a mesura que es llegeix**

En alguns casos, les tasques que ha de dur a terme el programa es basen en un únic recorregut. Es pot establir un cert paral·lelisme entre una situació en la qual es parteix d'un *array* on estan disposades ja les dades, i vosaltres per tractar-les en teniu prou a fer un recorregut des del primer al darrer element. En aquest cas, en lloc d'avançar posició a posició dins un *array*, s'avança valor a valor dins d'un fitxer. Exemples d'aquest cas pot ser trobar el màxim, el mínim, calcular una mitjana, mostrar els valors que compleixen certes condicions per pantalla, etc.

Per exemple, el programa següent, el qual, donat un fitxer anomenat "Document.txt" amb diferents paraules, compta la mitjana, sense decimals, de vocals per paraula que hi ha en total (nombre de vocals / nombre de paraules). Per saber quan acaba el fitxer, s'usa la paraula "fi" com a marca (que no ha de ser tractada). Com es pot veure, fins i tot en casos on cal acumular resultats i fer codi una mica complex usant disseny descendent es poden dur a terme les tasques d'una sola passada, si es plantegen correctament.

```
1  package unitat6.apartat2.exemples;
2  import java.io.File;
3  import java.util.Scanner;
4  public class MitjanaVocals {
5    //la paraula "fi" serveix com a marca de final de fitxer
6    public static final String MARCA_FI = "fi";
7    public static void main (String[] args) {
8      MitjanaVocals programa = new MitjanaVocals();
9      programa.inici();
10   }
11   public void inici() {
12      try {
13        //S'intenta obrir el fitxer
14        File f = new File("Document.txt");
15        Scanner lector = new Scanner(f);
16        //Dades a recopilar
```

```
17        int numParaules = 0;
18        int numVocals = 0;
19        //Marca de finalització de lectura
20        boolean llegir = true;
21        while (llegir) {
22          String paraula = lector.next();
23          if (MARCA_FI.equals(paraula)) {
24            //Marca de final
25            llegir = false;
26          } else {
27            //Tractar dada
28            numParaules++;
29            numVocals = numVocals + comptarVocals(paraula);
30          }
31        }
32        System.out.println("Hi ha " + numParaules + " paraules.");
33        System.out.println("Hi ha " + numVocals + " vocals.");
34        double mitjana = numVocals/numParaules;
35        System.out.println("La mitjana és " + mitjana);
36        //Cal tancar el fitxer
37        lector.close();
38      } catch (Exception e) {
39        //Excepció!
40        System.out.println("Error: " + e);
41      }
42    }
43    /** Compta les vocals en una paraula.
44     *
45     * @param paraula Cadena de text on cal comptar les vocals
46     * @return Nombre de vocals
47     */
48    public int comptarVocals(String paraula) {
49      int res = 0;
50      //Es passa a minúscula tot per fer-ho mes fàcil
51      paraula = paraula.toLowerCase();
52      for(int i = 0;i < paraula.length(); i++) {
53        if (esVocal(paraula.charAt(i))) {
54          res++;
55        }
56      }
57      return res;
58    }
59    /** Diu si, donat un caràcter en minúscula, aquest és o no una vocal.
60     *
61     * @param c Caràcter a comprovar
62     * @return Si és (true) o no (false) una vocal
63     */
64    public boolean esVocal(char c) {
65      switch(c) {
66          case 'a':
67          case 'e':
68          case 'i':
69          case 'o':
70          case 'u': return true;
71          default: return false;
72      }
73    }
74  }
```

## 2. Primer lectura i tractament posterior

Una altra opció disponible és primer anar emmagatzemant les dades llegides dins un *array* a mesura que es llegeixen, i posteriorment operar sobre aquest *array*. Aquest estratègia té el desavantatge que els vostres programes són una mica menys eficients, ja que primer fan la lectura de dades i després realment es dediquen a fer la seva feina. A més a més, ocupen més memòria, ja que cal declarar l'*array* on desar totes les dades del fitxer.

Malauradament, hi ha situacions on és inevitable fer-ho d'aquesta manera. Qualsevol situació on abans de tractar les dades cal fer alguna modificació que depèn de totes elles, caldrà fer-ho d'aquesta manera. Molts programes d'ordinador precisament per això usen aquesta estratègia. Per exemple, els editors de text o fulls de càlcul fan precisament això. Primer carreguen el fitxer a memòria i posteriorment el mostren per pantalla de manera que us permeten modificar-lo. Fins que no feu "desar", no s'actualitzen les dades de memòria a disc.

L'exemple següent és un altre cas, molt més senzill que un processador de text, en què no es pot fer la feina requerida sense carregar abans les dades a un *array*. Llegeix les dades d'un fitxer anomenat "Enters.txt", les ordena, i llavors les escriu en un nou fitxer anomenat "EntersOrdenat.txt". Per ordenar un conjunt de valors, cal disposar de tots ells alhora. No hi ha cap altra manera. El programa es basa en indicar en el seu primer valor el nombre d'elements del fitxer per saber quants cal llegir. Fixeu-vos també com, mitjançant disseny descendent, es distribueix el codi per tractar cada fitxer en mètodes diferenciats.

També fixeu-vos que aquest és un cas en què es fa tractament d'excepcions en un mètode auxiliar, a `llegirDades`. En un cas com aquest, normalment no es mostra directament l'error per pantalla, sinó que s'invoca la sentència `return` amb un valor invàlid, i és **inici** qui comprova si aquest mètode ha acabat correctament o incorrectament. Això es deu al fet que tot mètode declarat amb un paràmetre de sortida sempre ha de retornar algun valor, tant si succeeix una excepció com si no.

```
1  package unitat6.apartat2.exemples;
2  import java.io.File;
3  import java.io.PrintStream;
4  import java.util.Arrays;
5  import java.util.Scanner;
6  public class OrdenarFitxer {
7    public static void main (String[] args) {
8      OrdenarFitxer programa = new OrdenarFitxer();
9      programa.inici();
10   }
11   public void inici() {
12     //Es llegeixen els valors
13     int[] valors = llegirDades("Enters.txt");
14     if (valors != null) {
15       //S'han pogut llegir les dades correctament
16       //S'ordenen
17       Arrays.sort(valors);
18       //S'escriuen en un nou fitxer
19       escriureArray("EntersOrdenats.txt", valors);
20     } else {
21       //Ha succeït un error llegint les dades
22       System.out.println("Hi ha hagut un error llegint les dades.");
23     }
24   }
25   /** Donat el nom d'un fitxer, llegeix els seus valors i el carrega en un
         array
26    * d'enters.
27    * @param nom Nom del fitxer
28    * @return Array amb les dades carregades des del fitxer
29    */
30   public int[] llegirDades(String nom) {
31     try {
32       File f = new File(nom);
33       Scanner lector = new Scanner(f);
34       int mida = lector.nextInt();
35       int[] dades = new int[mida];
36       for (int i = 0; i < mida; i++) {
```

```
37          dades[i] = lector.nextInt();
38        }
39        return dades;
40      } catch (Exception e) {
41        //No s'han pogut llegir les dades...
42        return null;
43      }
44    }
45    /** Donat un nom de fitxer i un array d'enters, l'escriu en aquest fitxer.
46     *
47     * @param nom Nom de la ruta del fitxer destinació
48     * @param dades Array amb les dades que cal escriure
49     */
50    public void escriureArray(String nom, int[] dades) {
51      try {
52        //S'intenta crear el fitxer
53        File f = new File(nom);
54        PrintStream escriptor = new PrintStream(f);
55        //Primer s'escriu el nombre de valors
56        escriptor.print(dades.length);
57        //S'escriuen els valors de l'array, separats per espais
58        for (int i = 0; i < dades.length; i++) {
59          escriptor.print(" " + dades[i]);
60        }
61        System.out.println("Fitxer generat satisfactòriament.");
62        //Cal tancar el fitxer
63        escriptor.close();
64      } catch (Exception e) {
65        //Excepció!
66        System.out.println("Error escrivint dades: " + e);
67      }
68    }
69  }
```

Un altre cas on cal usar aquesta estratègia per tractar dades és quan cal modificar alguna de les dades que hi ha en un fitxer orientat a caràcter. Per exemple, suposeu que voleu canviar només un dels seus valors, deixant la resta igual. Atès que `PrintStream` elimina les dades d'un fitxer tan bon punt s'inicialitza, en realitat no és possible modificar un valor concret. Cal reescriure el seu contingut totalment, preveient els valors que es volen modificar en fer-ho.

Per exemple, el programa següent modifica un fitxer orientat a caràcter que conté 10 valors de tipus enter i suma 5 als valors que hi ha a les posicions parell. Per fer-ho, caldrà llegir tots els valors, emmagatzemar-los en un *array*, fer les modificacions escaients, i després sobreescriure el fitxer original al complet. Per saber quants valors hi ha, s'usa l'estratègia d'indicar-ho a l'inici del fitxer.

```
1   package unitat6.apartat2.exemples;
2   import java.io.File;
3   import java.io.PrintStream;
4   import java.util.Scanner;
5   public class ModificarValorsParells {
6     public static final String NOM_FITXER = "Enters.txt";
7     public static void main (String[] args) {
8       ModificarValorsParells programa = new ModificarValorsParells();
9       programa.inici();
10    }
11    public void inici() {
12      //Es llegeixen els valors
13      int[] valors = llegirDades(NOM_FITXER);
14      if (valors != null) {
15        //S'han pogut llegir les dades correctament
16        //Es modifiquen
17        for (int i = 0; i < valors.length; i = i + 2) {
```

```
18          valors[i] = valors[i] + 5;
19        }
20        //Es tornen a escriure tots al fitxer original
21        escriureArray(NOM_FITXER, valors);
22      } else {
23        //Ha succeït un error llegint les dades
24        System.out.println("Hi ha hagut un error llegint les dades.");
25      }
26    }
27    /** Donat el nom d'un fitxer, llegeix els seus valors i el carrega en un
          array
28     * d'enters.
29     * @param nom Nom del fitxer
30     * @return Array amb les dades carregades des del fitxer
31     */
32    public int[] llegirDades(String nom) {
33      try {
34        File f = new File(nom);
35        Scanner lector = new Scanner(f);
36        int mida = lector.nextInt();
37        int[] dades = new int[mida];
38        for (int i = 0; i < mida; i++) {
39          dades[i] = lector.nextInt();
40        }
41        return dades;
42      } catch (Exception e) {
43        //No s'han pogut llegir les dades...
44        return null;
45      }
46    }
47    /** Donat un nom de fitxer i un array d'enters, l'escriu a aquest fitxer.
48     *
49     * @param nom Nom de la ruta del fitxer destinació.
50     * @param dades Array amb les dades que cal escriure
51     */
52    public void escriureArray(String nom, int[] dades) {
53      try {
54        //S'intenta crear el fitxer
55        File f = new File(nom);
56        PrintStream escriptor = new PrintStream(f);
57        //Primer s'escriu el nombre de valors
58        escriptor.print(dades.length);
59        //S'escriuen els valors de l'array, separats per espais
60        for (int i = 0; i < dades.length; i++) {
61          escriptor.print(" " + dades[i]);
62        }
63        System.out.println("Fitxer generat satisfactòriament.");
64        //Cal tancar el fitxer
65        escriptor.close();
66      } catch (Exception e) {
67        //Excepció!
68        System.out.println("Error escrivint dades: " + e);
69      }
70    }
71  }
```

### Múltiples accessos al fitxer

Us podeu trobar casos en què un programa ha de fer moltes operacions senzilles sobre les dades d'un fitxer, de manera que cadascuna d'elles, individualment, es podria fer a mesura que el llegeix el fitxer. Per exemple, suposeu un programa que ha de trobar el màxim, el mínim, i la mitjana aritmètica dels valors a un fitxer. En un cas com aquest, intentar fer moltes coses alhora en una única passada sobre les dades d'un fitxer pot arribar a complicar el codi, o fins i tot trencar el vostre

disseny descendent, ja que tindreu blocs de codi que fan moltes coses en paral·lel en lloc de fer una tasca molt concreta i fàcil d'identificar. Es va calculant tot alhora a mesura que es llegeixen les dades, de manera que en lloc de tres mètodes `calculMaxim`, `calculMinim` i `calculMitjana`, podeu acabar amb un de sol anomenat `calculMinimMaximMitjana`!.

Una possible opció en aquest cas seria fer múltiples accessos al mateix fitxer, de manera que es fa un recorregut seqüencial nou per cada tasca individual. El fitxer s'obre, es tracten els valors i es tanca diverses vegades. Ara bé, això no és massa eficient donades les capacitats dels ordinadors moderns, en els quals la memòria és abundant, ja que precisament l'accés a fitxers és un dels punts més lents del sistema. En casos com aquest, el millor és usar l'estratègia de primer llegir els valors i desar-los en un *array*, i després tractar-lo per dur a terme cada tasca individual.

Només seria aconsellable fer múltiples recorreguts en el cas de voler limitar la despesa de memòria del vostre programa. Per exemple, si realment els fitxers són enormes, o el programa ha de funcionar en un ordinador amb una quantitat modesta de memòria. A escala d'aquests materials, no es donarà mai el cas. Al cap i a la fi, recordeu que, actualment, un simple processador de text no té problemes per carregar fitxers amb diversos megabytes d'informació.

### 2.2.3 Fitxers amb valors de diferents tipus

Un aspecte que també val la pena tenir en compte és que, tot i que en els exemples mostrats sempre s'ha treballat amb fitxers que contenen valors del mateix tipus, res impedeix barrejar-los. Simplement haureu de tenir sempre present quin és el format del fitxer que llegiu, i seguir-lo fidelment quan n'escriviu un de nou. En casos com aquest, és molt recomanable fer-se un esquema d'aquest format, per tenir-lo sempre present. Llavors, només cal cridar el mètode adient segons el tipus de dada esperat.

Per exemple, suposeu un programa que mostra la nota mitjana d'un conjunt d'estudiants. Per cada estudiant, els valors es desen en línies diferents, per facilitar la seva comprensió, d'acord al format següent. A la darrera línia hi ha la paraula "fi" com a marca de final.

```
1 Nom(String) Cognom(String) Número de notes(enter) Nota1(real) ... NotaN(real)
```

Per exemple:

```
1 Maria Almerich 3 4 3,5 6
2 Joan Sebastià 5 4,5 8,5 5 6,5 7
3 Carme Gonzàlez 4 6,5 8,75 10 9,5
4 fi
```

El codi que el resol podria ser el següent. Estudieu-lo amb atenció, ja que pot resultar una mica més complicat, a l'hora de llegir dades de diferents tipus

barrejats. El punt més interessant és el fet que una variable de tipus `Scanner` pot ser usada com a paràmetre d'entrada d'un mètode. En aquest aspecte, en tractar-se d'un tipus compost, no és diferent d'un `String` o un *array*.

```
package unitat6.apartat2.exemples;
import java.io.File;
import java.util.Scanner;
public class LlegirNotes {
  //la paraula fi serveix com a marca de final de fitxer
  public static final String MARCA_FI = "fi";
  public static void main (String[] args) {
    LlegirNotes programa = new LlegirNotes();
    programa.inici();
  }
  public void inici() {
    try {
      File f = new File("Notes.txt");
      Scanner lector = new Scanner(f);
      boolean llegir = true;
      while (llegir) {
        String nom = lector.next();
        if (MARCA_FI.equals(nom)) {
          llegir = false;
        } else {
          String cognom = lector.next();
          System.out.print("Estudiant: " + nom + " " + cognom);
          //Noteu com un scanner es pot passar com paràmetre
          double mitjana = llegirNotes(lector);
          System.out.println("— Mitjana: " + mitjana);
        }
      }
      //Cal tancar el fitxer
      lector.close();
    } catch(Exception e) {
      //Excepció!
      System.out.println("Error llegint estudiants: " + e);
    }
  }
  /** Donat un Scanner en un fitxer quan l'apuntador es troba sobre l'inici
   * de les notes, s'extreuen i es calcula la mitjana.
   *
   * @param lector Scanner a processar
   * @return Mitjana de notes
   */
  public double llegirNotes(Scanner lector) {
    double res = 0;
    try {
      //Es fan lectures. Cal controlar excepcions també!
      int numNotes = lector.nextInt();
      for (int i = 0; i < numNotes; i++) {
        //S'acumula el valor de les notes
        res = res + lector.nextDouble();
      }
      //Es calcula nota mitjana
      res = res/numNotes;
    } catch (Exception e) {
      //Excepció!
      System.out.println("Error llegint notes: " + e);
    }
    return res;
  }
}
```

**Repte 3**. Modifiqueu l'exemple de manera que ara faci el següent. D'una banda, en lloc de mostrar les dades per pantalla, les ha d'escriure en un fitxer anomenat "NotaMitja.txt". D'altra banda, es canvia el format del fitxer "Notes.txt". Per controlar el nombre de notes de cada estudiant, enlloc d'indicar-ho directament al

tercer valor de cada línia, ara es posa al final de cada línia el valor -1. Per exemple, ara ha de poder llegir el fitxer següent:

```
Maria Almerich 4 3,5 6 -1
Joan Sebastià 4,5 8,5 5 6,5 7 -1
Carme Gonzàlez 6,5 8,75 10 9,5 -1
fi
```

## 2.3 Accés seqüencial a fitxers orientats a byte

En el cas dels fitxers orientats a byte, les dades també s'emmagatzemen com una seqüència de valors, i per tant, l'esquema general que aplica en fitxers orientats a caràcter també es pot aplicar a aquest altre tipus. Totes les estratègies per saber quantes lectures és possible dur a terme també apliquen. La diferència fonamental és que no s'utilitza la seva representació en forma de cadena de text, sinó que s'usa el seu format binari, directament tal com s'usa dins de la memòria de l'ordinador. Segons el tipus de dades del valor emmagatzemat, aquest s'emmagatzemarà dins d'un nombre determinat de bytes. A més a més, els fitxers orientats a byte no usen cap delimitador per separar els valors.

La taula 2.2 mostra un recull de la mida de les representacions en binari dels diferents tipus primitius de Java.

**Taula 2.2.** Espai necessari per emmagatzemar els diferents tipus primitius

| Tipus | Paraula clau Java | Mida (bytes) |
|---|---|---|
| caràcter | `char` | normalment 2 |
| byte | `byte` | 1 |
| enter curt | `short` | 2 |
| enter simple | `int` | 4 |
| enter llarg | `long` | 8 |
| real de simple precisió | `float` | 4 |
| real de doble precisió | `double` | 8 |

La representació binària dels enters en Java és el complement a dos.

Donat aquest fet, suposeu que voleu emmagatzemar dins un fitxer d'aquestes característiques els valors enters 2, -14 i 25. Això vol dir que dins el fitxer es desen d'acord a la seva representació binària en 4 bytes (32 bits), que seria la següent per a Java:

- **2**: `00000000000000000000000000000010`
- **-14:** `11111111111111111111111111110010`
- **25:** `00000000000000000000000000011001`

Per inspeccionar un fitxer orientat a byte cal un editor hexadecimal, com l'HexEdit (Windows) o el GHex (Unix).

La figura 2.4 mostra un esquema de quina seria l'estructura d'un fitxer orientat a byte que conté aquests tres valors. Primer es mostra el seu equivalent en fitxer

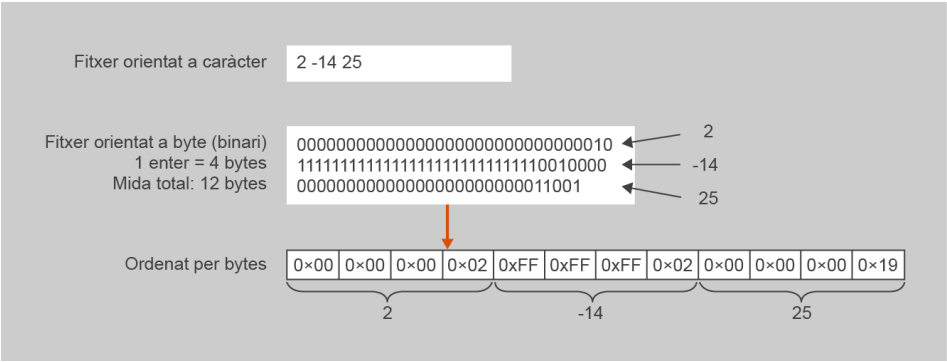
orientat a caràcter i quin seria el contingut directament en format binari (els tres conjunts de 0 i 1 tot just presentats, un rere l'altre). Tot i que, per motius d'espai a la figura, cada valor està en una fila diferent, dins un fitxer estarien escrits com una seqüència completa. No hi ha salts de línia. Finalment, a la figura es mostra en un format en què es diferencia el valor de cada byte, de manera que és més fàcil de seguir-ne el contingut, comptant que cada enter es representa amb 4 bytes.

**La notació hexadecimal**

Treballar amb notació binària és una mica farragós, ja que els valors es codifiquen en seqüències de 0 i 1 que poden ser llargues (32-64 bits), de manera que ocupen força espai i resulten difícils de llegir. Normalment, per facilitar la comprensió, els valors es representen en sistema hexadecimal, de manera que queden millor resumits i són fàcils de dividir en bytes. En el sistema hexadecimal, els dígits van del 0 al 15, comptant que els valors del 10 al 15 es representen amb les lletres A a la F. Cada dígit representa 4 bits, de manera que 0 = 0000, 1 = 0001 ... E = 1110, F = 1111. D'aquesta manera, un byte es representa usant dos dígits.

Quan un valor es vol representar en hexadecimal, s'usa el prefix "0x". Aquest no és part del valor, només és un símbol per indicar això.

**Figura 2.4.** Estructura d'un fitxer orientat a byte




Tingueu en compte que quan es diu que les dades es representen en binari, es vol dir exactament això. No es tracta que al fitxer s'escriu text amb els caràcters '1' i '0'. Si intenteu obrir un fitxer orientat a byte amb un editor de text, no visualitzareu res. A mode d'exemple, la majoria de fitxers dels processadors de text o de fulls de càlcul són orientats a byte. Per tant, si els obriu amb un editor de text simple, el que es visualitzarà serà totalment incomprensible (i a més a més, patiu el risc de fer malbé les dades contingudes).

Si us fixeu en la figura, quan els valors es resumeixen separant els seus bytes en hexadecimal, un fitxer orientat a byte té una estructura semblant a la d'un *array*, on cada posició seria representada per cadascun dels seus bytes, i cada valor concret en realitat ocupa un cert nombre de posicions. Aquesta característica cal que la tingueu ben present, ja que serà molt rellevant més endavant.

Tot això vol dir que no és possible generar fitxers orientats a byte només amb un editor de text, ni tampoc són gens fàcils d'inspeccionar a simple vista. El principal avantatge d'un fitxer d'aquest tipus és que, a costa d'una certa complicació a l'hora de controlar el seu contingut, tots els valors d'un mateix tipus sempre ocuparan exactament el mateix espai dins del fitxer. En molts casos, això vol dir que el fitxer ocupa menys espai. Per exemple, si es volen emmagatzemar els valors de

tipus enter 15 i 10371, en un fitxer orientat a caràcter el primer ocupa 2 caràcters i el segon cinc. Tenint en compte que s'han de separar d'un espai, això equival a 8 o 16 bytes, ja que, depenent de la configuració del vostre entorn de treball, cada caràcter a Java es desa en 1 o 2 bytes. En el cas d'un fitxer orientat byte, escriure tots dos sempre ocuparà 8 bytes.

### 2.3.1 Inicialització

En aquest cas, per als fitxers orientats a byte es pot usar la mateixa classe tant si es volen llegir dades com si es volen escriure.

> Per tractar de manera senzilla fitxers orientats a byte, Java ofereix la classe `RandomAccessFile`, pertanyent al *package* `java.io`. Aquesta s'usa tant per llegir com per escriure dades.

Novament, aquesta actua com un tipus compost. Per tant, igual que passa amb les classes `File`, `Scanner` o `PrintStream`, cal inicialitzar-la amb una variable diferent per cada fitxer sobre el qual es vol treballar (en el cas de treballar amb més d'un fitxer alhora). La manera de fer-ho és amb la instrucció següent:

```
1 import java.io.File;
2 import java.io.RandomAccessFile;
3 ...
4 RandomAccessFile raf = new RandomAccessFile(File ruta, String mode);
```

En aquest cas, per inicialitzar una variable d'aquest tipus correctament calen dos paràmetres. D'una banda, i igual que passava amb `Scanner`, cal especificar la ruta del fitxer amb el qual es vol treballar. Addicionalment, cal indicar mitjançat una cadena de text el mode de treball a l'hora de processar les dades del fitxer. Hi ha diversos modes de treball, però només veureu els dos més senzills, usats en la immensa majoria de casos. La utilitat dels altres modes és més marginal.

Aquests dos modes són:

- **r**: mode lectura. Indica que només es volen llegir dades des del fitxer representat per la ruta especificada. Això implica que la ruta ha de referir-se a un fitxer que hi ha al sistema de fitxers. En cas contrari, es produirà un error. També produirà un error qualsevol intent d'invocar mètodes associats a l'escriptura de dades sobre la variable `RandomAccessFile` tot just inicialitzada.
- **rw**: mode escriptura-lectura. Aquest mode permet tant llegir dades des d'un fitxer, igual que l'anterior, com escriure'n. Si la ruta usada es refereix a un fitxer que no existeix, se'n crearà un de nou, que estigui buit.

Per tant, si només es volen llegir dades d'un fitxer ubicat a la ruta "C:\Programes\Unitat 6\Apartat 2\Exemples\Document.txt", caldria fer:

```
import java.io.File;
import java.io.RandomAccessFile;
...
File f = new File("C:\Programes\Unitat 6\Apartat 2\Exemples\Document.txt");
RandomAccessFile raf = new RandomAccessFile(f, "r");
```

Si, per contra, es volen escriure dades a un fitxer en aquesta mateixa ubicació, caldrà fer:

```
import java.io.File;
import java.io.RandomAccessFile;
...
File f = new File("C:\Programes\Unitat 6\Apartat 2\Exemples\Document.txt");
RandomAccessFile raf = new RandomAccessFile(f, "rw");
```

Al contrari que amb la classe `PrintStream`, si la ruta usada en inicialitzar una variable d'aquesta classe en mode "rw" es correspon a un fitxer que ja existeix, no s'esborren les seves dades, ni queden afectades en cap mesura.

Potser us pot semblar una mica estrany o redundant el mode "r", existint el mode "rw", que permet fer exactament el mateix i més coses. En realitat, usar el mode "r" sempre que només es volen llegir dades és força útil, ja que garanteix que si, per un error en programar, el vostre programa intenta escriure al fitxer amb el qual treballeu, l'operació d'escriptura s'avortarà immediatament. Aquesta possibilitat és especialment interessant quan s'està treballant amb dades de fitxers, ja que no fa gens de gràcia sobreescriure (i, per tant, perdre) la informació que hi havia en un fitxer a causa d'un error involuntari al codi.

Per tant, a partir d'ara, si només es volen llegir dades d'un fitxer, sempre s'usarà el mode "r" exclusivament.

### 2.3.2 Escriptura de dades

En aquest cas, es començarà per explicar l'escriptura de dades, en lloc de la lectura. El motiu principal és que, atès que `RandomAccesFile` treballa amb fitxers orientats a byte, no hi ha cap manera senzilla de generar fitxers de dades pel vostre compte, al contrari que per al cas dels fitxers orientats a caràcter, on n'hi ha prou a usar un editor de text. L'única manera de generar fitxers orientats a byte que puguin ser llegits correctament és mitjançant codi d'un programa.

> Cada cop que es fa una operació de lectura o escriptura, l'apuntador es desplaça automàticament el mateix nombre de bytes amb què s'ha operat.

El nombre de bytes desplaçats dependrà de la mida associada al tipus primitiu del Java que s'ha escrit. Cal tenir sempre en compte que, com que `RandomAccesFile` treballa amb fitxers orientats a byte, aquesta classe transforma el valor del tipus primitiu a una seqüència de bytes, segons el mètode invocat. Per saber exactament en quant ha variat l'apuntador del fitxer, cal saber quants bytes ocupa cada tipus primitiu del Java.

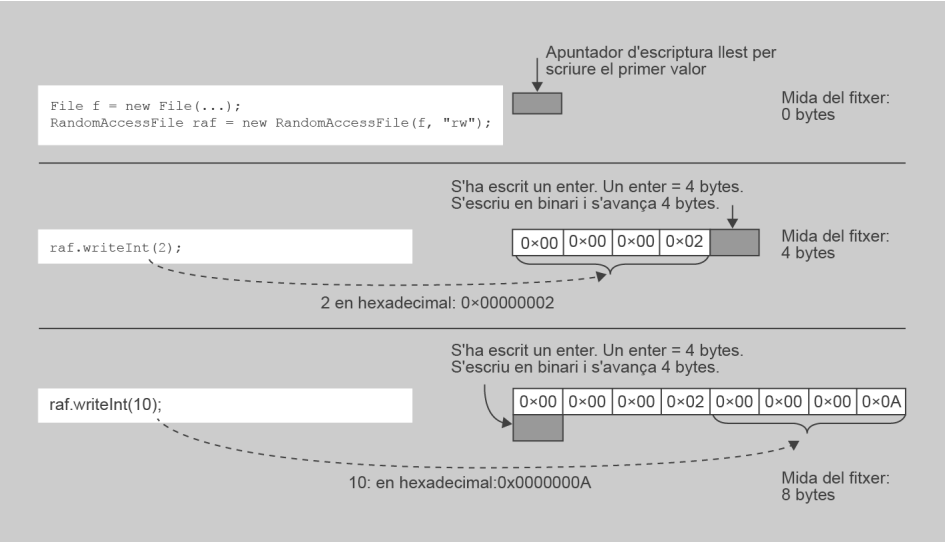
A la taula 2.3 es mostren els mètodes disponibles per escriure dades.

**TAULA 2.3.** Mètodes d'escriptura de dades de la classe "RandomAccessFile"

| Mètode | Bytes escrits |
|---|---|
| `writeBoolean(boolean b)` | 1 |
| `writeByte(byte v)` | 1 |
| `writeChar(char c)` | 2 |
| `writeDouble(double d)` | 8 |
| `writeFloat(float f)` | 4 |
| `writeInt(int i)` | 4 |
| `writeLong(long l)` | 8 |
| `writeShort(short s)` | 2 |

A més a més de la diferència en el format de representació de les dades del fitxer i que no cal encarregar-se d'escriure cap delimitador entre els valors, el procés general d'escriptura de dades és similar a la dels fitxers orientats a caràcter: un valor rere l'altre. La figura 2.5 mostra un esquema del funcionament del procés d'escriptura.

**FIGURA 2.5.** Escriptura seqüencial de valors en un fitxer orientat a byte




Com a exemple, estudieu i proveu el programa següent. Aquesta és la versió de l'exemple anterior que escriu una seqüència de 20 valors enters, de manera que, començant per l'1, cada valor és el doble de l'anterior. Fixeu-vos com ara no cal escriure cap delimitador. Els valors es van escrivint consecutivament, un rere l'altre.

```
1  package unitat6.apartat2.exemples;
2  import java.io.File;
3  import java.io.RandomAccessFile;
4  public class EscriureEntersDoblesBinari {
5    public static void main (String[] args) {
6      EscriureEntersDoblesBinari programa = new EscriureEntersDoblesBinari();
7      programa.inici();
8    }
9    public void inici() {
```

```
10      try {
11        File f = new File("Enters.bin");
12        RandomAccessFile raf = new RandomAccessFile(f,"rw");
13        //Ara no hi ha delimitadors. S'escriuen els valors consecutius.
14        //Es van generant els valors i escrivint
15        int valor = 1;
16        for (int i = 0; i < 20; i ++) {
17          raf.writeInt(valor);
18          valor = valor*2;
19        }
20        System.out.println("Fitxer escrit satisfactòriament.");
21        //La mida d'un enter són 4 bytes.
22        //La mida del fitxer hauria de ser 20*4 = 80 bytes
23        //No oblidar-se de tancar el fitxer
24        raf.close();
25      } catch (Exception e) {
26        //Excepció!
27        System.out.println("Error escrivint dades: " + e);
28      }
29    }
30  }
```

### Sobreescriptura de fitxers

La classe `RandomAccessFile` té un comportament una mica especial quan escriu dades sobre un fitxer que ja existeix, en lloc de fer-ho sobre un de completament nou. Al contrari que `PrintStream`, aquest fitxer existent no s'esborra i es comença a escriure des de zero, sinó que les noves dades es van escrivint directament sobre les antigues, reemplaçant-les byte a byte. Aquest és un fet amb el qual cal tenir molta cura, ja que si el nombre de bytes escrits és inferior a la mida del fitxer existent, quedarà "brossa" al final (les dades del fitxer existent que no s'han arribat a sobreescriure).

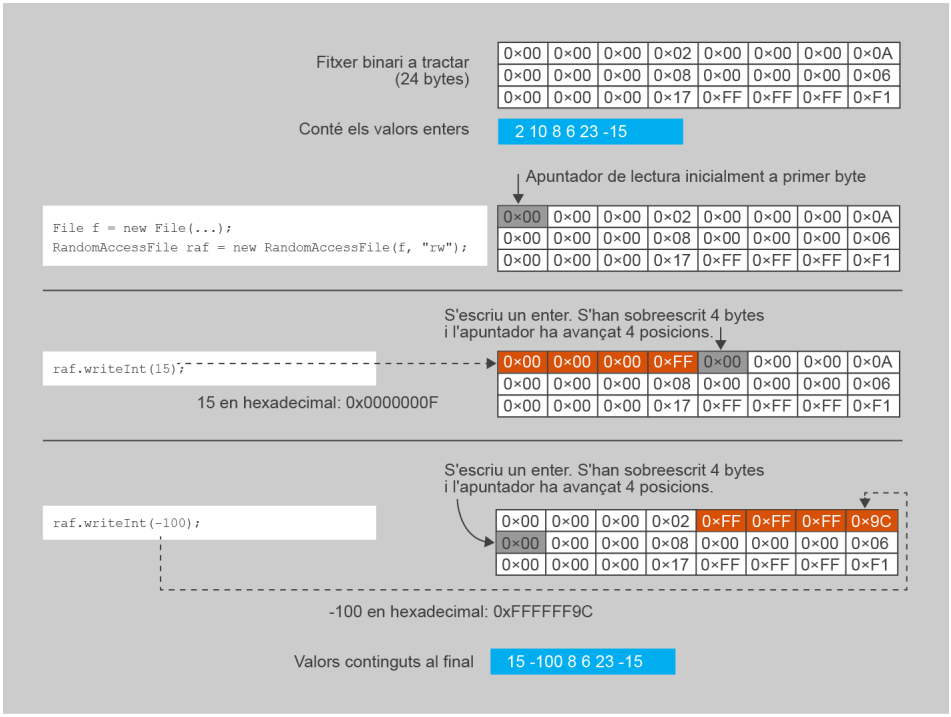
La figura 2.6 us mostra un esquema d'aquest fet, en un cas en què hi ha un fitxer amb 6 valors enters escrits i volem reemplaçar el seu contingut a només dos valors, 15 i -100.

Segons aquest comportament, cal anar molt amb compte amb el fet que les dades que no són sobreescrites es mantenen al fitxer, sobretot quan es vol reemplaçar tot el contingut d'un fitxer per dades noves. Normalment, el que se sol fer en aquests casos, és eliminar tots els bytes sobrants un cop s'ha acabat l'escriptura de dades. Per fer-ho, es disposa dels mètodes:

- `setLength(long mida)`. Modifica la mida del fitxer, de manera que si el valor especificat com a paràmetre és més petit que la mida actual, s'eliminen totes les dades per sobre de la mida especificada. Si és més gran, s'omple la diferència amb bytes tot a 0.

- `long getFilePointer()`. Avalua la posició on és en aquests moments l'apuntador, mesurat en el nombre de bytes des de l'inici del fitxer. Per al cas que esteu tractant, si s'invoca quan heu acabat totes les operacions d'escriptura, us dirà la mida de les dades escrites, i per tant, la nova mida que ha de tenir el fitxer.

**FIGURA 2.6.** Sobreescriptura de fitxers en usar RandomAccessFile




El programa següent usa aquest mecanisme per reemplaçar completament el fitxer resultant de l'exemple anterior (20 valors enters, cadascun el doble de l'anterior), anomenat "enters.bin", per cinc valors enters -1. El resultat final és un fitxer de només 20 bytes, en lloc de 80. Proveu també què succeeix si abans d'executar-lo sobre aquest fitxer de 80 bytes s'elimina la sentència `raf.setLength(apuntador)`.

```
1  import java.io.File;
2  import java.io.RandomAccessFile;
3  public class SobreescriureEntersBinari {
4    public static void main (String[] args) {
5      SobreescriureEntersBinari programa = new SobreescriureEntersBinari();
6      programa.inici();
7    }
8    public void inici() {
9      try {
10       File f = new File("Enters.bin");
11       RandomAccessFile raf = new RandomAccessFile(f, "rw");
12       //L'apuntador està al primer byte
13       long apuntador = raf.getFilePointer();
14       System.out.println("Inici: Apuntador a posició " + apuntador);
15       //Sobreescrivim els primers cinc valors
16       for (int i = 0; i < 5; i++) {
17         raf.writeInt(-1);
18       }
19       //Si el fitxer ja tenia més de cinc enters, al final hi ha brossa
20       apuntador = raf.getFilePointer();
21       System.out.println("Fi: Apuntador a posició " + apuntador);
22       //Es fixa la mida del fitxer als valors escrits
23       raf.setLength(apuntador);
24       raf.close();
25       System.out.println("Fitxer modificat correctament.");
26     } catch (Exception e) {
27       //Excepció!
28       System.out.println("Error escrivint dades: " + e);
29     }
30   }
31 }
```

### 2.3.3 Lectura de dades

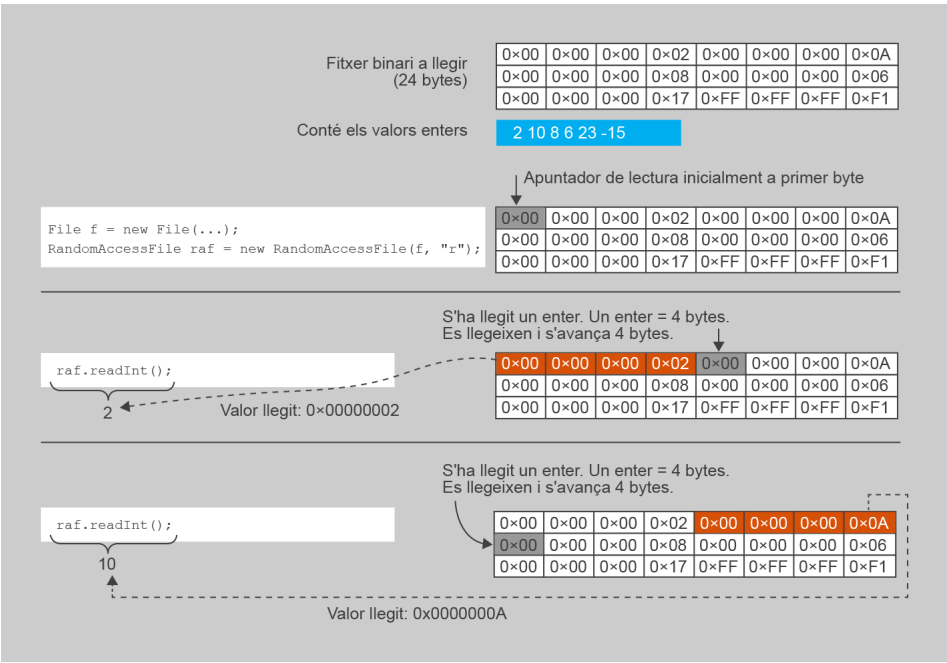
Un cop conegut el procés de lectura de dades de manera seqüencial i el format com s'escriuen les dades en un fitxer orientat a byte, no hi ha gaires sorpreses en el procés de lectura. Novament, es parteix d'un apuntador a la posició inicial que va avançant a mesura que s'invoquen mètodes per fer operacions de lectura. Exactament igual com passava amb la classe `Scanner`, hi ha un mètode específic per a cada tipus de dades esperat dins la seqüència de valors, enumerats a la taula 2.4.

**Taula 2.4.** Mètodes de lectura de dades de la classe "RandomAccessFile"

| Mètode | Tipus de dada llegida |
|---|---|
| `readByte()` | `byte` |
| `readShort()` | `short` |
| `readInt()` | `int` |
| `readLong()` | `long` |
| `readFloat()` | `float` |
| `readDouble()` | `double` |
| `readBoolean()` | `boolean` |
| `readChar()` | `char` |

La figura 2.7 mostra un exemple del funcionament del procés de lectura en un fitxer orientat a byte.

**Figura 2.7.** Lectura seqüencial de valors en un fitxer orientat a byte




En el cas dels fitxers binaris també cal controlar no llegir més valors dels que realment hi ha escrits al fitxer, o en cas contrari es produirà una excepció. Ara bé,

en aquest tipus de fitxer teniu un avantatge, i és que no sempre cal una marca de finalització o indicar el nombre d'elements prèviament. Atès que cada valor del mateix tipus sempre ocupa el mateix espai (els `int` 4 bytes, els `double` 8 bytes, etc.), si coneixeu la mida del fitxer en bytes, es pot calcular quants valors d'un tipus conté fent una simple divisió: mida fitxer/mida tipus.

El codi d'exemple següent llegeix un fitxer orientat a byte anomenat "Enters.bin", el qual conté cert nombre de valors enters, i els va mostrant per pantalla. Podeu usar el resultat dels exemples d'escriptura de dades. Fixeu-vos especialment en la manera com es calcula el nombre de valors que cal llegir, donada la mida del fitxer i coneguda la mida de la representació en binari d'un enter a Java (4 bytes).

```
1  package unitat6.apartat2.exemples;
2  import java.io.File;
3  import java.io.RandomAccessFile;
4  public class LlegirEntersBinari {
5    public static void main (String[] args) {
6      LlegirEntersBinari programa = new LlegirEntersBinari();
7      programa.inici();
8    }
9    public void inici() {
10     try {
11       File f = new File("Enters.bin");
12       RandomAccessFile raf = new RandomAccessFile(f, "r");
13       //Càlcul del nombre d'enters
14       long numEnters = f.length() / 4;
15       System.out.println("Hi ha " + numEnters + " enters.");
16       for (int i = 0; i < numEnters; i++) {
17         int valor = raf.readInt();
18         System.out.println("S'ha llegit el valor " + valor);
19       }
20       raf.close();
21     } catch (Exception e) {
22       //Excepció!
23       System.out.println("Error en la lectura: " + e);
24     }
25   }
26 }
```

**Repte 4**. Feu un programa que llegeixi un fitxer orientat a byte que contingui una seqüència de valors reals qualsevol. Cal mostrar-los per pantalla ordenats de més gran a més petit. Tingueu en compte que, per poder provar-lo, abans haureu de generar aquest fitxer vosaltres mateixos d'alguna manera.

### Lectura incorrecta de dades

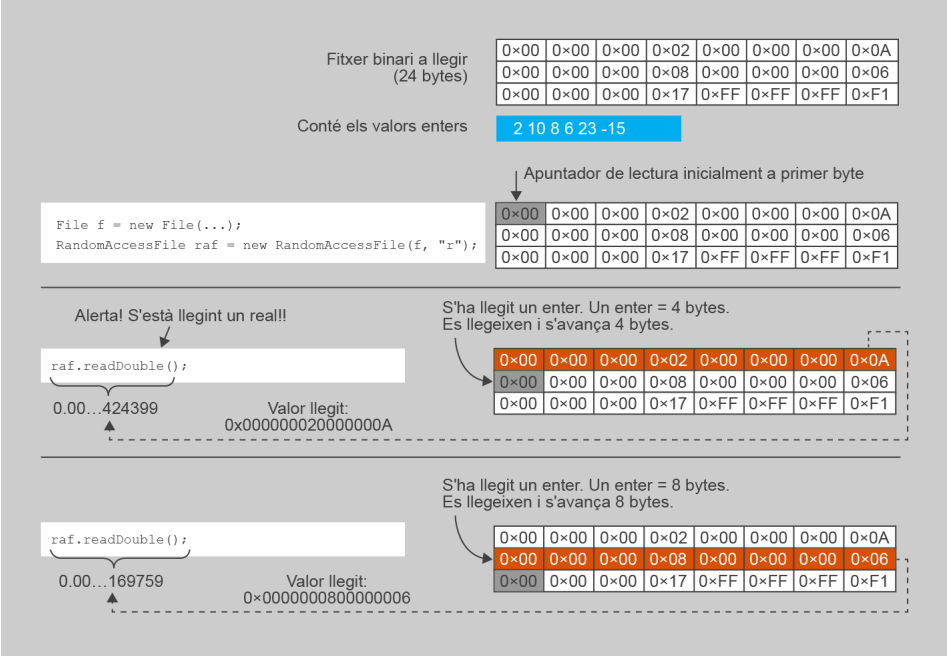
**Lectura incorrecta de dades**

És semblant al que passa si, amb un processador de text, per exemple, intenteu obrir un fitxer que no li pertoca, fet amb un altre programa.

A l'hora de llegir dades en un fitxer orientat a byte, cal procurar sempre usar el mètode encertat a cada moment, d'acord al tipus que s'espera llegir. Ara bé, la classe `RandomAccessFile` actua d'una manera força diferent a `Scanner`, ja que quan realitza una lectura incorrecta no es produeix cap excepció. Aquesta classe, quan executa qualsevol dels seus mètode de lectura, sempre llegeix el nombre de bytes associats al tipus de dada que es vol llegir, i els interpreta al valor que correspon dins aquest tipus. Això ho fa independentment del fet que, en realitat, això sigui correcte o no.

La figura 2.8 mostra un exemple d'aquest comportament, llegint un valor de tipus real sobre un fitxer on hi ha emmagatzemats valors enters.

**Figura 2.8.** Lectura incorrecta de dades en un fitxer orientat a byte




Per veure-ho més clar, tot seguit hi ha una versió de l'exemple anterior que llegeix i presenta valors reals. En aquest cas, executar-lo sobre un fitxer on només hi ha escrits valors enters seria incorrecte, ja que no s'està usant el mètode adient per llegir els valors continguts. Ara bé, el mètode `readDouble()` simplement llegeix 8 bytes consecutius (o sigui, en aquest cas, la representació dins el fitxer de dos enters seguits, 4 + 4 bytes), i intenta interpretar aquests 8 bytes com si fossin un valor de tipus real. Evidentment, el valor resultant de fer aquest procés serà totalment incorrecte, ja que dins del fitxer no hi ha cap valor de tipus real. Executeu-lo per veure'n el resultat.

```
1  package unitat6.apartat2.exemples;
2  import java.io.File;
3  import java.io.RandomAccessFile;
4  public class LlegirDoubleBinari {
5    public static void main (String[] args) {
6      LlegirDoubleBinari programa = new LlegirDoubleBinari();
7      programa.inici();
8    }
9    public void inici() {
10     try {
11       File f = new File("Enters.bin");
12       RandomAccessFile raf = new RandomAccessFile(f, "rw");
13       //Càlcul del nombre d'enters
14       long numEnters = f.length() / 8;
15       System.out.println("Crec que hi ha " + numEnters + " reals.");
16       for (int i = 0; i < numEnters; i++) {
17         double valor = raf.readDouble();
18         System.out.println("S'ha llegit el valor real " + valor);
19       }
20       raf.close();
21     } catch (Exception e) {
22       //Excepció!
23       System.out.println("Error en la lectura: " + e);
24     }
25   }
26 }
```

Per tant, cal ser molt acurat a l'hora de fer lectures sobre fitxers orientats a byte, i tenir ben clar el seu format i quin tipus de dades cal llegir a cada moment, ja que el programa no ens avisarà amb cap excepció. Simplement, aquest comportarà de manera ben estranya en executar-lo.

## 2.4 Accés relatiu a les dades

L'accés seqüencial a les dades d'un fitxer permet processar el seu contingut de manera relativament senzilla, en basar-se en un mecanisme pràcticament idèntic a la lectura de dades des del teclat, o la seva escriptura a la pantalla. Els blocs d'instruccions per dur a terme operacions de lectura i escriptura són bàsicament els mateixos, només varia l'origen i la destinació de les dades en les operacions d'entrada / sortida, així com els aspectes vinculats a si el fitxer és orientat a caràcter o a byte.

Ara bé, en realitat, un fitxer no és ben bé com el teclat o la pantalla, ja que les dades ni deixen d'estar disponibles un cop llegides, ni té sentit que resulti impossible sobreescriure-les parcialment. L'accés seqüencial és una manera còmoda de tractar fitxers, però a canvi de perdre un cert grau de versatilitat.

Si considereu que un fitxer orientat a byte, i només aquest tipus, és com un *array* de certa envergadura on cada byte representa una posició, ha de ser possible accedir a les dades que conté de la mateixa manera que es fa amb els *arrays*. D'una banda, de manera seqüencial, fent recorreguts sobre cadascuna de les seves posicions, com heu vist fins ara. Però també ha de ser possible accedir directament a una posició concreta només indicant una ubicació per al seu apuntador de lectura o escriptura. O sigui, donat un fitxer orientat a byte, es pot dir que disposeu d'*accés relatiu*.

L'accés relatiu a un fitxer sovint també s'anomena *accés aleatori* o *accés directe*. Dins d'aquest context, són termes sinònims.

S'anomena `accés relatiu` a la capacitat d'accedir a qualsevol element dins una seqüència de dades sense haver de tractar prèviament els elements anteriors.

Una bona manera d'entendre aquesta definició és contrastant l'accés relatiu al seqüencial, en ser precisament termes contraposats. Donat un fitxer amb una seqüència de 10 elements, per poder tractar només el desè, si s'hi accedeix seqüencialment, primer cal llegir (i anar descartant successivament) els nou elements anteriors. L'accés relatiu, en canvi, proporciona operacions que fan possible llegir aquest desè element directament, sense haver de processar abans tots els anteriors usant mètodes de lectura.

La classe **RandomAccessFile** permet l'accés relatiu als fitxers orientats a byte.

Al cap i a la fi, el seu nom, en anglès, vol dir "fitxer d'accés aleatori".

### 2.4.1 Posicionament

El primer pas per poder dur a terme accés relatiu a un fitxer és especificar l'índex de la posició que es vol accedir, ja sigui tant per llegir la dada que hi ha emmagatzemada com per escriure-hi un valor nou. Un cop conegut aquest índex, mitjançant la sintaxi adient, és possible aquella posició. En el cas dels *arrays*, aquesta sintaxi és del tipus:

```
1 array[índex]
```

Atès que `RandomAccessFile` és una classe, la sintaxi per accedir a les posicions del fitxer és mitjançant la invocació de mètodes. Cada byte individual dins del fitxer és com una posició d'un *array*, i el primer byte del fitxer es consideraria la seva posició 0.

Per dur a terme la tasca d'indicar l'índex per fer una operació de lectura o escriptura, les variables de tipus `RandomAccessFile` tenen una diferència important en l'accés a les dades respecte a un *array*. Si ho recordeu, els seus mètodes de lectura i escriptura no disposen de cap paràmetre on s'especifiqui un índex. Això es deu al fet que les classes que tracten les dades d'un fitxer disposen d'un apuntador intern que indica en tot moment quina és la dada següent a processar. Cada cop que es fa una operació, aquest apuntador es va desplaçant automàticament fins a arribar al final de la seqüència de valors. El que `RandomAccessFile` proporciona són un seguit de mètodes que permeten modificar el valor d'aquest apuntador, indicant de manera explícita la seva posició. Per tant, per accedir a una posició qualsevol del fitxer, primer cal un pas previ on s'indica la posició en què es vol operar, invocant algun d'aquests mètodes de posicionament de l'apuntador i, posteriorment, es pot dur a terme la lectura o l'escriptura.

Per gestionar el posicionament de l'apuntador, la classe defineix els mètodes següents:

**1.** `void seek(long pos)`. Ubica l'apuntador exactament a la posició especificada pel paràmetre `pos`, mesurat en bytes, de manera que qualsevol accés a les dades serà tot just sobre el byte següent. No hi ha cap restricció en el valor d'aquest paràmetre, essent possible ubicar l'apuntador molt mes enllà del final real del fitxer. Això pot semblar una mica estrany, però aviat veureu el motiu.

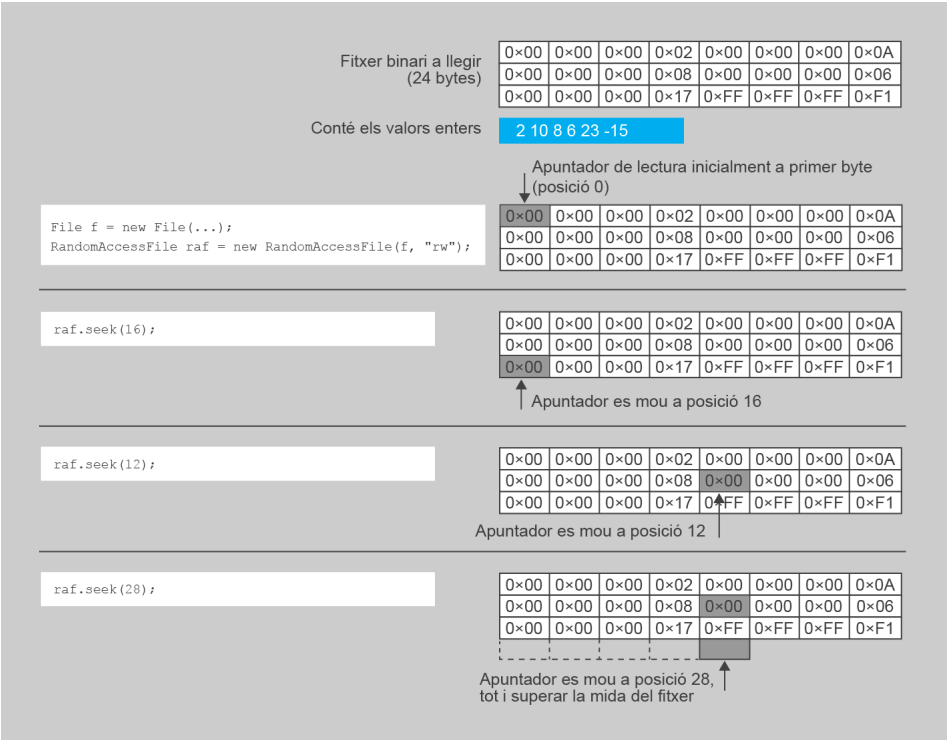
La figura 2.9 mostra un esquema del funcionament d'aquest mètode donat un fitxer orientat a bytes que conté un conjunt d'enters. Recordeu que cada enter ocupa 4 bytes.

El codi d'exemple següent usa el mètode **seek** per moure l'apuntador del fitxer a diferents posicions de manera arbitrària, tal com es podria fer amb l'índex d'un *array*.

```
1 package unitat6.apartat2.exemples;
2 import java.io.File;
3 import java.io.RandomAccessFile;
4 public class MoureApuntadorSeek {
```

```
5   public static void main (String[] args) {
6     MoureApuntadorSeek programa = new MoureApuntadorSeek();
7     programa.inici();
8   }
9   public void inici() {
10    try {
11      File f = new File("Enters.bin");
12      RandomAccessFile raf = new RandomAccessFile(f, "r");
13      raf.seek(20);
14      long pos = raf.getFilePointer();
15      System.out.println("L'apuntador està a la posició " + pos);
16      raf.seek(0);
17      pos = raf.getFilePointer();
18      System.out.println("L'apuntador està a la posició " + pos);
19      raf.seek(100);
20      pos = raf.getFilePointer();
21      System.out.println("L'apuntador està a la posició " + pos);
22      raf.close();
23    } catch (Exception e) {
24      //Excepció!
25      System.out.println("Error: " + e);
26    }
27  }
28 }
```

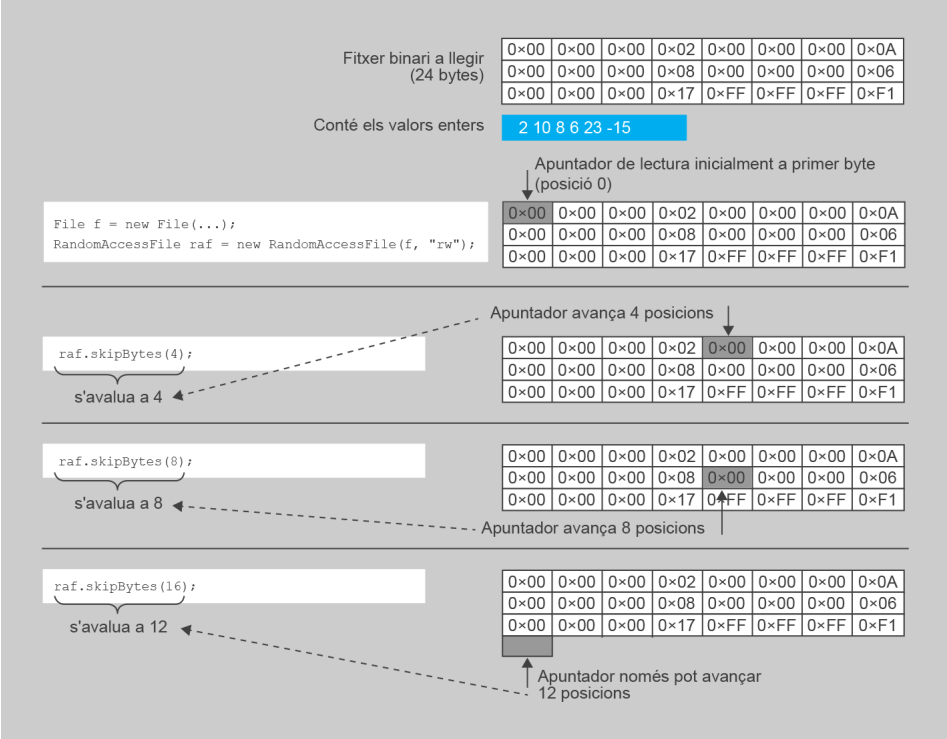
**FIGURA 2.9.** Accés aleatori mitjançant posicionament d'apuntador usant el mètode **seek**




**2.** `int skipBytes(int n)`. Avança la posició de l'apuntador en `n` bytes, de manera que aquest passa a valer `valor actual + n`. La seva invocació avalua el nombre de bytes que realment s'ha avançat. Cal tenir en compte que es pot donar el cas quan el valor resultant no sigui igual a `n`. Per exemple, si l'apuntador arriba al final del fitxer, el desplaçament de l'apuntador s'atura. Per tant, Independentment del valor del paràmetre `n`, sempre heu de controlar la posició de l'apuntador del fitxer a partir del valor retornat.

La figura 2.10 mostra un exemple de com es modifica l'apuntador a partir d'un seguit de crides al mètode **skipBytes**. Fixeu-vos que si a l'hora d'executar-la l'apuntador sobrepassa la mida del fitxer, aquest incrementa la seva mida amb nous bytes, tots a 0.

**FIGURA 2.10.** Accés aleatori mitjançant posicionament d'apuntador usant el mètode **skipBytes**




El codi d'exemple següent usa el mètode **skipBytes** per moure l'apuntador a diferents posicions dins del fitxer. Fixeu-vos com, en el darrer posicionament, s'intenta anar molt més enllà de la mida del fitxer. Si se sobrepassa aquest límit, l'apuntador mai avança més enllà de la mida del fitxer.

```
1  package unitat6.apartat2.exemples;
2  import java.io.File;
3  import java.io.RandomAccessFile;
4  public class MoureApuntadorSkipBytes {
5    public static void main (String[] args) {
6      MoureApuntadorSkipBytes programa = new MoureApuntadorSkipBytes();
7      programa.inici();
8    }
9    public void inici() {
10     try {
11       File f = new File("Enters.bin");
12       RandomAccessFile raf = new RandomAccessFile(f, "r");
13       long skip = raf.skipBytes(20);
14       long pos = raf.getFilePointer();
15       System.out.print("L'apuntador ha avançat " + skip + " posicions. ");
16       System.out.println("Està a la posició " + pos);
17       skip = raf.skipBytes(8);
18       pos = raf.getFilePointer();
19       System.out.print("L'apuntador ha avançat " + skip + " posicions. ");
20       System.out.println("Està a la posició " + pos);
21       //S'intenta avançar molt més enllà de la mida del fitxer
22       skip = raf.skipBytes(400);
23       pos = raf.getFilePointer();
24       System.out.print("L'apuntador ha avançat " + skip + " posicions. ");
25       System.out.println("Està a la posició " + pos);
26       raf.close();
```

```
    } catch (Exception e) {
      //Excepció!
      System.out.println("Error: " + e);
    }
  }
}
```

### 2.4.2 Lectura de dades relativa

Mitjançant els mètodes `seek` i `skipBytes` és possible indicar des de quina posició del fitxer, mesurada en bytes, es vol dur a terme la lectura, la qual es duu a terme amb les mateixes instruccions que el tractament seqüencial (els mètodes `readInt()`, `readDouble`,etc.), i de fet, mentre no es torni a invocar cap dels dos mètodes de posicionament de l'apuntador, el tractament del fitxer passa a ser totalment seqüencial. Cada cop que es llegeix un valor, l'apuntador avança tants bytes com la mida del tipus del valor (4 bytes si és un enter, 8 si és un `double`, etc.).

El programa d'exemple següent fa ús dels mètodes `seek` i `skipBytes` per llegir les dades d'un fitxer que només conté valors enters. S'ignora la primera meitat dels valors i es van mostrant els valors de la segona meitat de manera intermitent. O sigui, si hi ha 20 valors, se salten els 10 primers valors i es mostren per pantalla els valors 11, 13, 15, etc. Al llarg de la seva execució, mitjançant el mètode `getFilePointer`, es va controlant l'evolució de l'apuntador de lectura abans d'accedir a cada valor.

```
package unitat6.apartat2.exemples;
import java.io.File;
import java.io.RandomAccessFile;
public class LlegirEntersRelatiu {
  public static void main (String[] args) {
    LlegirEntersRelatiu programa = new LlegirEntersRelatiu();
    programa.inici();
  }
  public void inici() {
    try {
      File f = new File("Enters.bin");
      RandomAccessFile raf = new RandomAccessFile(f, "r");
      long numEnters = f.length()/4;
      long meitat = numEnters/2;
      //Apuntador a l'inici de l'enter a la meitat del fitxer
      raf.seek(meitat*4);
      long pos = raf.getFilePointer();
      //Es llegeix fins a arribar al final del fitxer
      do {
        System.out.print("(apuntador a la posició " + pos + ") ->");
        //Es llegeix un enter
        int valor = raf.readInt();
        System.out.println(" S'ha llegit el valor " + valor);
        //Se salta l'enter següent (4 bytes)
        raf.skipBytes(4);
        pos =  raf.getFilePointer();
      } while (pos < f.length());
    } catch (Exception e) {
      //Excepció!
      System.out.println("Error llegint dades: " + e);
    }
  }
}
```

Com succeïa en el tractament seqüencial, a l'hora de fer lectures mitjançant accés relatiu cal anar amb compte i usar sempre el mètode que correspon al tipus del valor al qual es vol accedir, així com procurar no accedir a posicions més enllà de la mida real del fitxer.

Per al cas de l'accés relatiu, a més a més, també cal ser molt acurat per indicar quina ha de ser la posició de l'apuntador. Aquest sempre ha d'estar a la primera posició del grup de bytes que conformen cada valor. En cas contrari, el programa es comportarà de manera erràtica, ja que es llegiran bytes de diferents valors barrejats. Per exemple, si un fitxer només conté valors enters, les úniques posicions vàlides per fer lectures són la 0, 4, 8, 12, etc.

### 2.4.3 Escriptura de dades relativa

Cal calcular molt bé quina és la posició en número de bytes on ha d'anar l'apuntador.

La classe `RandomAccessFile` té la particularitat que, sempre que fa una escriptura sobre un fitxer que ja existeix, només modifica el valor on s'ubica l'apuntador i cap altre. Per tant, amb l'ajut del mètode de posicionament és possible modificar els valors dins d'un fitxer orientat a byte directament, tal com si fossin els valors dins un *array*. De fet, normalment, l'escriptura de dades relativa només té sentit quan es treballa sobre un fitxer ja existent.

> L'accés relatiu a un fitxer evita haver de carregar totes les dades a memòria, per exemple a un *array*, i després tornar-les a escriure si es volen fer modificacions a valors individuals. Aquestes esmenes es poden fer directament sobre les posicions del fitxer.

Un cop l'apuntador es troba a la posició que es vol modificar, el procés d'escriptura mitjançant accés relatiu és similar al de lectura, simplement canviant la invocació de mètodes de lectura pels d'escriptura (`writeInt(...)`, `writeDouble(...)`, etc.).

El programa següent mostra un exemple d'escriptura mitjançant accés relatiu, usant els mètodes de posicionament de l'apuntador. En aquest cas, el programa accedeix a un fitxer orientat a byte que conté un conjunt de valors enters i modifica els valors que ocupen una posició múltiple de cinc amb la seva pròpia posició. O sigui, reemplaça el cinquè valor per un 5, el desè element per un 10, el quinzè element per un 15, etc. fins a arribar al final del fitxer. En acabar l'escriptura, es mostren els nous valors del fitxer per pantalla.

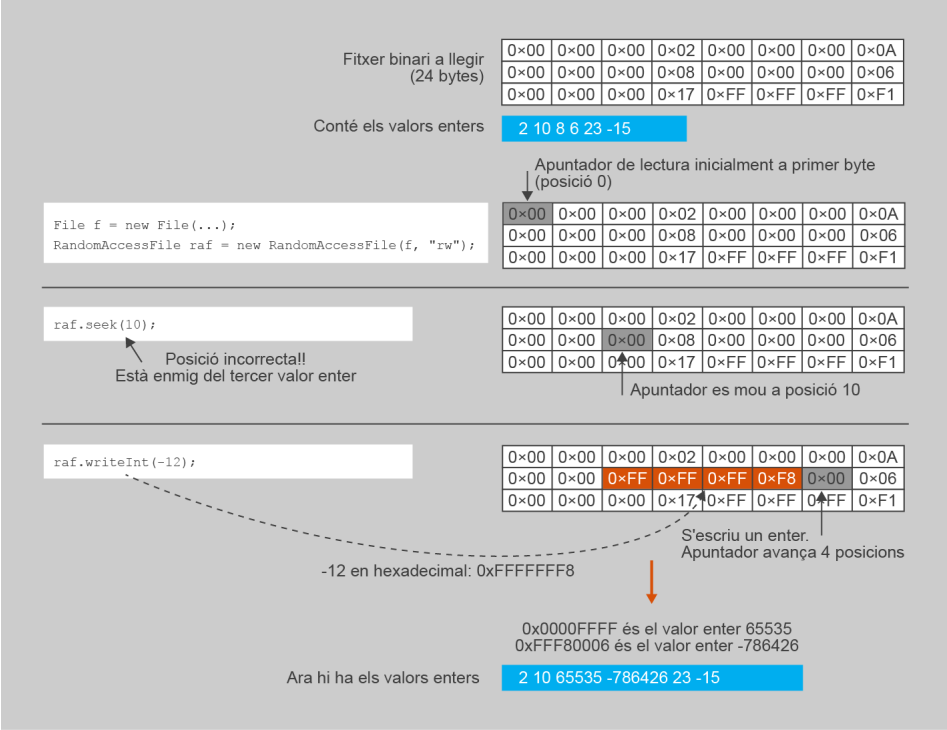
```
1  package unitat6.apartat2.exemples;
2  import java.io.File;
3  import java.io.RandomAccessFile;
4  public class EscriureEntersRelatiu {
5    public static void main (String[] args) {
6      EscriureEntersRelatiu programa = new EscriureEntersRelatiu();
7      programa.inici();
8    }
9    public void inici() {
10     try {
```

```
      File f = new File("Enters.bin");
      //Es mostra el contingut original (si existeix)
      System.out.println("Valors inicials del fitxer.");
      mostrarFitxerBinari(f);
      //Es fan les modificacions
      modificaFitxerBinari(f);
      //I ara es mostra el nou contingut
      System.out.println("Nous valors del fitxer.");
      mostrarFitxerBinari(f);
    } catch (Exception e) {
      //Excepció!
      System.out.println("Error escrivint dades: " + e);
    }
  }
  /** Mostra per pantalla tots els valors d'un fitxer orientat a byte que
   * conté enters.
   *
   * @param f Ruta del fitxer a mostrar
   */
  public void mostrarFitxerBinari(File f) {
    try {
      RandomAccessFile raf = new RandomAccessFile(f, "r");
      long pos = raf.getFilePointer();
      while (pos < f.length()) {
        int valor = raf.readInt();
        System.out.print(" " + valor);
        pos = raf.getFilePointer();
      }
      raf.close();
      System.out.println();
    } catch (Exception e) {
      //Excepció!
      System.out.println("Error llegint dades: " + e);
    }
  }
  /** Modifica el contingut d'un fitxer orientat a byte que conté enters, de
   * manera que cada 5 posicions se sobreescriu el valor pel número de
   * la pròpia posició.
   *
   * @param f Ruta del fitxer a modificar
   */
  public void modificaFitxerBinari(File f) {
    try {
      RandomAccessFile raf = new RandomAccessFile(f, "rw");
      //S'avança fins a l'inici del cinquè enter
      //valor 1 = posició 0, valor 2 = posició 4, valor 3 = posició 8
      //valor 4 = posició 12, valor 5 = posició 16
      raf.seek(16);
      long pos = raf.getFilePointer();
      int i = 1;
      while (pos < f.length()) {
        //S'escriu un valor a l'apuntador actual
        raf.writeInt(i*5);
        i++;
        //Se salten 4 valors enters. Aquests no es toquen
        raf.skipBytes(4*4);
        //En total, l'apuntador ha avançat 5 valors (escrit + saltats)
        pos = raf.getFilePointer();
      }
      //Escriptura finalitzada
      raf.close();
    } catch (Exception e) {
      //Excepció!
      System.out.println("Error escrivint dades: " + e);
    }
  }
}
```

Com passava amb la lectura, cal tenir molta cura a l'hora de posicionar l'apuntador, de manera que aquest estigui sempre en el primer byte del valor que es vol sobreescriure. En aquest cas, però, fer bé els càlculs és especialment crític, ja que en cas contrari fareu malbé dades dins del fitxer.

La figura 2.11 mostra un exemple d'escriptura sobre una posició invàlida en fer un accés relatiu incorrecte sobre un fitxer orientat a byte amb 6 valors enters. Fixeu-vos que l'apuntador se situa just a la meitat del tercer valor, de manera que, al escriure els 4 bytes d'un nou valor enter, se sobreescriu la segona meitat del tercer valor (2 bytes) i la primera meitat del quart (2 bytes més). El resultat és que es canvia part dels bytes corresponents a aquests valors, de manera que ara passen a representar valors completament diferents. Ja a simple vista, aquests nous valors només deixen entreveure que alguna cosa s'ha fet malament.

**FIGURA 2.11.** Escriptura incorrecta en un accés relatiu.




### Seek més enllà de la mida del fitxer

Mitjançant el sistema d'accés relatiu, també és possible escriure dades directament des del final d'un fitxer existent, de manera que s'annexin.

En el procés d'escriptura mitjançant accés relatiu també hi ha un cas especial. Es tracta de quan s'ha cridat el mètode `seek` de manera que l'apuntador s'ubica més enllà del final del fitxer (per exemple, el fitxer té 80 bytes i s'ubica a la posició 100), i llavors es duu a terme una escriptura. En aquest cas, la mida del fitxer s'incrementa automàticament fins a la posició de l'apuntador, omplint tots els nous bytes amb 0. Un cop fet, llavors es materialitza l'escriptura.

Per exemple, si donat un fitxer de 80 bytes s'usa el mètode **seek** per ubicar l'apuntador fins a la posició 100 i llavors es fa l'escriptura d'un enter, el fitxer creixerà fins als 100 bytes, i els seus darrers 20 bytes estaran tots a 0. Llavors es durà a terme l'escriptura, de manera que la mida del fitxer final queda en 104 bytes. El programa següent mostra aquest cas.

```java
package unitat6.apartat2.exemples;
import java.io.File;
import java.io.RandomAccessFile;
public class SeekCreixFitxerBinari {
  public static final String NOM_FITXER = "Seek.bin";
  public static void main (String[] args) {
    SeekCreixFitxerBinari programa = new SeekCreixFitxerBinari();
    programa.inici();
  }
  public void inici() {
    File f = new File(NOM_FITXER);
    crearFitxer(f);
    System.out.println("La mida inicial és " + f.length());
    executaSeek(f);
    System.out.println("La mida a l'hora de fer seek i escriure és " + f.length
        ());
  }
  /** Crea un fitxer qualsevol de 80 bytes.
   *
   * @param f Ruta del fitxer a crear
   */
  public void crearFitxer(File f) {
    try {
      RandomAccessFile raf = new RandomAccessFile(f, "rw");
      for (int i = 0; i < 20; i++) {
        raf.writeInt(i);
      }
      raf.setLength(20*4);
      raf.close();
    } catch (Exception e) {
      //Excepció!
      System.out.println("Error escrivint dades: " + e);
    }
  }
  /** Crea un fitxer qualsevol de 80 bytes.
   *
   * @param f Ruta del fitxer a crear
   */
  public void executaSeek(File f) {
    try {
      RandomAccessFile raf = new RandomAccessFile(f, "rw");
      //Es va 20 bytes més enllà de la mida del fitxer
      raf.seek(f.length() + 20);
      //S'escriu un valor qualsevol
      raf.writeInt(100);
      raf.close();
    } catch (Exception e) {
      //Excepció!
      System.out.println("Error escrivint dades: " + e);
    }
  }
}
```

Mitjançant aquest sistema, també és possible escriure dades directament des del final d'un fitxer existent, de manera que s'annexin.

**Repte 5**. Feu un programa que, donat un fitxer orientat a byte que conté qualsevol nombre de valors reals, els ordeni de menor a major. Aquesta tasca l'ha de dur directament sobre el fitxer, i no pas carregant les dades a un *array*, ordenant i després escrivint-les de nou al fitxer. Per veure que funciona, fer que mostri per pantalla els valors continguts abans i després de l'ordenació.

## 2.5 Solucions als reptes proposats

### Repte 1

```
package unitat6.apartat2.exemples;
import java.io.File;
import java.util.Scanner;
public class LlegirEnters {
  public static final int NUM_VALORS = 15;
  public static void main (String[] args) {
    LlegirEnters programa = new LlegirEnters();
    programa.inici();
  }
  public void inici() {
      Scanner lector = null;
      try {
        //S'intenta obrir el fitxer
        File f = new File("Reals.txt");
        lector = new Scanner(f);
        //De moment, el màxim és el primer valor
        double maxim = lector.nextDouble();
        //Recorrem la resta de valors
        for (int i = 1; i < NUM_VALORS; i ++) {
          double valor = lector.nextDouble();
          //El nou valor és més gran?
          if (maxim < valor) {
            maxim = valor;
          }
        }
        System.out.println("El valor més gran és: " + maxim);
      } catch (Exception e) {
        //Excepció!
        System.out.println("Error: " + e);
      }
      //Amb error o sense, cal tancar el fitxer
      lector.close();
  }
}
```

## Repte 2

```java
package unitat6.apartat2.exemples;
import java.io.File;
import java.io.PrintStream;
public class EscriureEntersDobles {
  public static final int NUM_VALORS = 20;
  public static void main (String[] args) {
    EscriureEntersDobles programa = new EscriureEntersDobles();
    programa.inici();
  }
  public void inici() {
      PrintStream escriptor = null;
      try {
        //S'intenta obrir el fitxer
        File f = new File("Enters.txt");
        escriptor = new PrintStream(f);
        //S'escriu el primer valor, que no necessita un espai abans
        int valor = 1;
        escriptor.print(valor);
        //Es van generant la resta de valors i escrivint
        for (int i = 1; i < 20; i ++) {
          if (i%5 == 0) {
            //Cada 5 elements, s'escriu com a delimitador un salt de línia
            escriptor.print("\n");
          }  else {
            //Si no, s'escriu com a delimitador un espai en blanc
            escriptor.print(" ");
          }
          //Es calcula i escriu el nou valor
          valor = valor*2;
          escriptor.print(valor);
        }
        System.out.println("Fitxer escrit satisfactòriament.");
      } catch (Exception e) {
        //Excepció!
        System.out.println("Error: " + e);
      }
      //Amb error o sense, cal tancar el fitxer
      escriptor.close();
  }
}
```

## Repte 3

```java
package unitat6.apartat2.reptes;
import java.io.File;
import java.util.Scanner;
import java.io.PrintStream;
public class GenerarFitxerNotes {
  //la paraula fi serveix com a marca de final de fitxer
  public static final String MARCA_FI = "fi";
  public static final double MARCA_FI_NOTES = -1;
  public static void main (String[] args) {
    GenerarFitxerNotes programa = new GenerarFitxerNotes();
    programa.inici();
  }
  public void inici() {
    try {
      File in = new File("NotesMarca.txt");
      Scanner lector = new Scanner(in);
      File out = new File ("NotaMitja.txt");    //Es genera el fitxer de
          sortida
      PrintStream escriptor = new PrintStream(out);
      boolean llegir = true;
      //Només cal canviar les escriptures a pantalla per al fitxer de sortida
      //O sigui, on posa "System.out" posar "escriptor"
      while (llegir) {
        String nom = lector.next();
        if (MARCA_FI.equals(nom)) {
          llegir = false;
        } else {
          String cognom = lector.next();
          escriptor.print("Estudiant: " + nom + " " + cognom);
          //Noteu com un Scanner es pot passar com a paràmetre
          double mitjana = llegirNotes(lector);
          escriptor.println(" - Mitjana: " + mitjana);
        }
      }
      lector.close();              //Cal tancar els fitxers
      escriptor.close();
      System.out.println("Fitxer escrit satisfactòriament.");
    } catch(Exception e) {         //Excepció!
      System.out.println("Error llegint estudiants: " + e);
    }
  }
  /** Donat un Scanner en un fitxer quan l'apuntador és sobre l'inici
   * de les notes, s'extreuen i es calcula la mitjana.
   * @param lector - Scanner a processar
   * @return Mitjana de notes
   */
  public double llegirNotes(Scanner lector) {
    double res = 0;
    try {
      //Ara es llegeix fins a una marca de fi (-1)
      boolean llegir = true;
      int numNotes = 0;
      while (llegir) {
        double valor = lector.nextDouble();
        if (valor == MARCA_FI_NOTES) {    //Final
          llegir = false;
        } else {                  //S'acumula el valor de les notes
          res = res + valor;
          numNotes++;
        }
      }
      res = res/numNotes;          //Es calcula nota mitjana
    } catch (Exception e) {        //Excepció!
      System.out.println("Error llegint notes: " + e);
    }
    return res;
  }
}
```

## Repte 4

```java
package unitat6.apartat2.reptes;
import java.io.File;
import java.io.RandomAccessFile;
import java.util.Arrays;
import java.util.Random;
public class CalculsRealsBinari {
  public static void main (String[] args) {
    CalculsRealsBinari programa = new CalculsRealsBinari();
    programa.inici();
  }
  public void inici() {
    File f = new File("Reals.bin");
    crearReals(f);
    double[] valors = llegirReals(f);
    if (valors != null) {
      //Tot correcte
      Arrays.sort(valors);
      for (int i = valors.length - 1; i >= 0; i--) {
        System.out.println(valors[i]);
      }
    } else {
      //Hi ha hagut un error en la lectura
      System.out.println("Error llegint dades.");
    }
  }
  /** Donat un fitxer orientat a byte amb reals, els llegeix tots
   * i els posa a un array.
   * @param f Ruta del fitxer a llegir
   * @return Array de reals
   */
  public double[] llegirReals(File f) {
    try {
      RandomAccessFile raf = new RandomAccessFile(f, "r");
      //Càlcul del nombre de reals
      long numReals = f.length() / 8;
      //per inicialitzar un array cal un "int", no un "long"
      int n = (int)numReals;
      double[] array = new double[n];
      for (int i = 0; i < numReals; i++) {
        array[i] = raf.readDouble();
      }
      raf.close();
      return array;
    } catch (Exception e) {
      //Excepció!
      return null;
    }
  }
  /** Genera un fitxer orientat a byte amb 20 valors reals, entre 0 i 100,
   * a l'atzar.
   *
   * @param f Ruta del fitxer a generar
   */
  public void crearReals(File f) {
    try {
      RandomAccessFile raf = new RandomAccessFile(f, "rw");
      Random r = new Random();
      for (int i = 0; i < 20; i++) {
        raf.writeDouble(100*r.nextDouble());
      }
      raf.close();
    } catch (Exception e) {
      System.out.println("Error generant fitxer: " + e);
    }
  }
}
```

## Repte 5

```java
package unitat6.apartat2.reptes;
import java.io.File;
import java.io.RandomAccessFile;
public class OrdenarRealsBinari {
  public static void main (String[] args) {
    OrdenarRealsBinari programa = new OrdenarRealsBinari();
    programa.inici();
  }
  public void inici() {
    File f = new File ("Reals.bin");
    System.out.println("Contingut original del fitxer:");
    mostrarFitxer(f);
    ordenarFitxer(f);
    System.out.println("————————————————————————————");
    System.out.println("Contingut ordenat del fitxer:");
    mostrarFitxer(f);
  }
  /** Donat un fitxer orientat a byte que conté reals, ordena directament sobre
   * el fitxer tots els seus valors, de menor a major.
   *
   * @param f Ruta al fitxer amb les dades a ordenar
   */
  public void ordenarFitxer(File f) {
    try {
     RandomAccessFile raf = new RandomAccessFile(f, "rw");
     //Cada real ocupa 8 byes, per tant, cal avançar de 8 en 8
     for(long i = 0; i < f.length(); i = i + 8) {
       for(long j = i + 8; j < f.length(); j = j + 8) {
         //Es llegeix el valor a la posició "i"
         raf.seek(i);
         double valorI = raf.readDouble();
         //Es llegeix el valor a la posició "j"
         raf.seek(j);
         double valorJ = raf.readDouble();
         //Es comparen
         if (valorI > valorJ) {
           //Si "i" major que "j", s'intercanvien el lloc
           raf.seek(i);
           raf.writeDouble(valorJ);
           raf.seek(j);
           raf.writeDouble(valorI);
         }
       }
     }
     raf.close();
    } catch (Exception e) {
      System.out.println("Error ordenant fitxer: " + e);
    }
  }
  /** Donat un fitxer orientat a byte que conté reals, mostra els valors
   * per pantalla.
   *
   * @param f Ruta del fitxer a mostrar
   */
  public void mostrarFitxer(File f) {
    try {
     RandomAccessFile raf = new RandomAccessFile(f, "r");
     long numReals = f.length()/8;
     for (int i = 0; i < numReals; i++) {
       System.out.println(raf.readDouble());
     }
     raf.close();
    } catch (Exception e) {
      System.out.println("Error mostrant fitxer: " + e);
    }
  }
}
```

## 3. Tractament modular de dades. El joc de combats a l'arena

Els principis de la modularitat també són aplicables per resoldre problemes en els quals es duen a terme operacions amb fitxers. De fet, normalment dins de les aplicacions se solen generar mòduls clarament diferenciats vinculats a la gestió de totes les dades persistents.

En aquest apartat es planteja un exemple d'aplicació modular on es duu a terme el tractament de dades emmagatzemades en fitxers. Concretament, s'estudia com ampliar una aplicació ja existent, generada usant els principis de modularitat, de manera que aquests principis se segueixin mantenint en la nova versió amb noves funcionalitats.

### 3.1 Descripció del problema

El programa que serveix com a punt de partida és un joc on l'usuari, el jugador, es va enfrontant amb successius adversaris en una arena de combat. Tant el jugador com els seus adversaris són definits per un seguit de valors, que anomenaran els seus atributs, mitjançant els quals s'indica la seva perícia lluitant: resistència, nivell de poder, capacitat d'atacar o defensar, etc.

Cada combat es divideix en rondes, a l'inici de les quals el jugador i el seu adversari trien secretament una estratègia a seguir. En cada ronda es pot seguir una estratègia diferent. Segons les estratègies triades per cadascú, el combat s'anirà resolent més favorablement cap a un o cap a l'altre, fins que finalment es consideri que un dels dos ha estat derrotat. Si es derrota l'adversari, s'atorga una puntuació al jugador. Si el jugador és derrotat, acaba la partida. L'objectiu final del jugador és sobreviure deu combats, assolint la màxima puntuació possible en el procés.

A la secció "Annexos" del web disposeu del codi font complet de l'aplicació sobre la qual es treballa en aquest apartat. Estudieu-lo atentament.

#### 3.1.1 Divisió del problema

Si bé el procés de resoldre les rondes de combat segons els atributs dels lluitadors té les seves particularitats, per plantejar quin és el funcionament de l'aplicació, es deixarà en aquesta descripció general, sense entrar en més detall. De totes maneres, si voleu fer la idea més aclaridora, tot seguit s'exposa la descomposició del problema mitjançant disseny descendent:

1. Generar els atributs del nou jugador.
2. Anunciar inici del combat.
   (a) Mostrar estat del jugador.

3. Triar l'adversari.
4. Combatre.
   (a) Mostrar estat dels lluitadors.
      i. Mostrar estat del jugador.
      ii. Mostrar estat de l'adversari.
   (b) Triar estratègia del jugador.
   (c) Triar estratègia de l'adversari.
   (d) Resoldre resultats d'estratègies.
      i. Llançar monedes.
      ii. Penalitzar lluitador.
      iii. Danyar lluitador.
      iv. Guarir lluitador.
   (e) Restaurar lluitador.
5. Resoldre resultat del combat.
   (a) Atorgar puntuació.
   (b) Pujar de nivell.
   (c) Finalització del joc.

### 3.1.2 Mòduls del programa

Partint de la divisió en subproblemes del joc de combats de l'arena, s'ha fet la divisió en mòduls següent. D'una banda, en el *package* `joc.arena.regles` hi haurà les classes:

- **Monedes:** per a les tasques vinculades al llançament de monedes per resoldre una ronda.
- **Lluitador:** per a les tasques vinculades a la manipulació de les dades d'un lluitador (danyar, guarir, etc.).
- **Bestiari:** per a les tasques vinculades a la generació d'adversaris i el jugador.
- **Combat:** per a les tasques vinculades a la resolució d'estratègies enfrontades.

D'altra banda, en el *package* `joc.arena.interficie` es decideix dividir les classes que tracten la pantalla i el teclat, de manera que hi haurà:

- **EntradaTeclat:** s'encarrega de les tasques importants que són donades pel que escriu l'usuari usant el teclat.
- **SortidaPantalla:** com l'anterior, però per mostrar informació en pantalla.

La classe principal, **JocArena** és al *package* que engloba els anteriors, `joc.arena`, donada la jerarquia de noms.

Partint d'aquesta descripció, es volen afegir dues funcionalitats noves:

- Mantenir un rànquing amb les deu millor puntuacions obtingudes. Aquestes es desen en un fitxer, de manera que es mantenen entre partides diferents. En finalitzar el joc, si un jugador ha obtingut una puntuació que mereix estar entre les deu primeres, pot indicar quines són les seves inicials (tres lletres), perquè hi constin associades a la puntuació.

- Ara mateix, els atributs dels adversaris estan escrits en el codi font del programa (a la classe `Bestiari`). Això impedeix afegir nous adversaris fàcilment. Caldria modificar el seu codi font i compilar de nou el programa. Estaria bé que les dades dels adversaris s'obtinguin d'un fitxer, de manera que, només modificant aquest fitxer, el programa ja incorpora sempre automàticament els adversaris continguts.

## 3.2 Criteris d'elecció de tipus de fitxer

Una de les primeres decisions que cal prendre quan cal tractar fitxers és establir com s'hi emmagatzemaran les dades (en format caràcter o byte) i l'accés que es durà a terme (seqüencial o relatiu). En molts casos, sigui quin sigui el sistema triat, el problema es podrà resoldre igualment. L'única diferència serà la complexitat de l'algorisme que cal implementar i el nombre de vegades que s'ha de llegir el fitxer. En qualsevol cas, usar accés relatiu sempre implicarà que el fitxer serà orientat a byte.

A la taula 3.1 es resumeixen alguns pros i contres de cada tipus, perquè els tingueu en compte.

**Taula 3.1.** Avantatges i desavantatges segons el tipus de fitxers

| Tipus de fitxer | Avantatges | Desavantatges |
|---|---|---|
| Orientat a caràcter | Són fàcils de crear i editar amb eines externes (un editor de text simple).<br>Són fàcils de depurar (veure si el seu contingut és correcte.<br>En llegir-los, és fàcil comprovar mitjançant codi de tipus d'una dada dades abans de llegir-la (mètodes `hasNex...`). | La mida que ocupa cada valor pot ser molt variable.<br>Cal controlar el nombre de valor en el fitxer, o alguna marca de finalització<br>Només permeten tractament seqüencial.<br>No es poden sobreescriure fragments concrets. Cal reescriure tot el fitxer al complet. |
| Orientat a byte | La mida que ocupa cada valor es sempre la mateixa.<br>Lligat al punt anterior, es fàcil calcular quants valors contenen.<br>Permeten l'accés relatiu.<br>Permeten sobreescriptures parcials de les dades. | El codi és més complicat.<br>No és fàcil veure què contenen.<br>No permeten comprovar els tipus dels valors abans de llegir-los. En cas d'error, és difícil de detectar. |

## 3.3 La biblioteca "joc.arena.fitxers"

Normalment, el tractament de dades dins de fitxers se sol fer en un *package* diferenciat. Aquesta decisió és coherent amb la de disposar, per exemple, d'un altre vinculat a l'entrada / sortida de dades amb el teclat i la pantalla. A l'hora de triar el nom del *package*, és recomanable mantenir també en el seu nom la relació jeràrquica entre mòduls dins l'aplicació. Per tant, aquest *package* es pot dir `joc.arena.fitxers`.

Una aproximació assenyada per enfocar l'estructura d'aquesta biblioteca és fer-ho de manera que cada fitxer sigui tractat per una classe diferent. En aquest cas, doncs, caldrà dues classes noves, una per tractar el fitxer amb la llista de puntuacions i un altre per al tractament dels adversaris.

### 3.3.1 La classe Ranquing

Una llista de màximes puntuacions.

Aquesta classe serà l'encarregada de gestionar el fitxer amb les màximes puntuacions. Ha de poder tant llegir-les per mostrar-les per pantalla com modificar-ho per escriure'n de noves.

Amb vista a tenir una idea clara de com ha de funcionar, cal decidir quin serà el funcionament exacte de la llista de puntuacions màximes. Aquesta llista tindrà sempre 10 entrades, i inicialment, quan encara no s'ha jugat cap partida, hi haurà un seguit de puntuacions per defecte.

#### Elecció de tipus de fitxer i accés

Abans de començar cal escollir el nom del fitxer on mantenir les dades persistents, la seva ubicació al sistema de fitxers, i el seu tipus i com tractar-lo.

Pel que fa al nom, això ja és a gust del programador. En aquest cas, es pot anomenar "Ranquing". Ara bé, per triar-ne la ubicació, el més assenyat normalment és que la ruta a un fitxer sigui relativa, per no imposar cap restricció sobre quines carpetes hi han d'haver en cada ordinador on es copiï el programa. En aquest cas es pot usar la pròpia carpeta de treball de l'aplicació.

Només queda decidir el tipus de fitxer. D'entrada, sempre val la pena preveure l'opció d'un fitxer de text seqüencial, ja que és el més senzill de processar i depurar. En aquest sentit, un llistat de 10 puntuacions és una informació que es pot tractar seqüencialment de manera simple, tant per mostrar-la per pantalla com per cercar on cal afegir nous elements. L'única part on no tot encaixa perfectament és per inserir puntuacions. Caldrà reescriure el fitxer des de zero. Però com és un fitxer molt petit, 10 línies, no és un gran problema i es considera assumible.

### Propagació d'errors

Quan es realitzen operacions sobre fitxer, poden succeir un seguit de situacions errònies. Per exemple, intentar llegir dades d'un fitxer inexistent, llegir dades quan ja s'ha arribat al final del fitxer o d'un tipus inesperat, en tenir el fitxer un format incorrecte. En Java, aquests errors s'anomenen excepcions, i cal controlar-los usant una sentència `try/catch`.

En el cas d'un programa modular amb fitxers, és molt important que, cada cop que s'invoca un mètode on dins el seu codi es tracten fitxers, el codi que ha dut a terme la invocació pugui establir si tot ha anat bé o no. Per tant, els mètodes que tracten dades en fitxers han de poder avisar si han fet totes les tasques correctament o no.

> La **propagació d'errors** és el mecanisme mitjançant el qual un mètode avisa al codi que l'ha invocat si ha pogut dur a terme la seva tasca correctament o no.

Hi ha diferents mecanismes per dur a terme la propagació d'errors. Per a aquesta classe s'ha usat el més simple, que és reservar un dels valors del paràmetre de sortida per indicar que ha succeït un error (sovint, s'usa el valor -1). Un cop s'ha avaluat el mètode invocat, cal comprovar si el resultat és el valor reservat o no. Si és així, és que hi ha hagut una excepció dins el mètode, i per tant aquest no ha pogut dur a terme la seva tasca correctament. Llavors, cal actuar en conseqüència. Per exemple, mostrant un missatge d'error per pantalla.

En el cas de mètodes que no disposen de cap paràmetre de sortida (tipus `void`), es força a què tinguin un paràmetre de sortida de tipus enter, que servirà exclusivament per establir si tot ha anat bé o no. Normalment, es fa que s'avaluïn a 0 si tot ha anat bé i -1 si hi ha hagut alguna excepció.

De moment, en el codi de la classe `Ranquing` només es pot veure la propagació d'errors pel que fa a la part de tractament d'excepcions i retorn del valor -1. Més endavant, quan es vegi com s'invoquen aquests mètodes des d'altres classes, us quedarà més clar com es controla la propagació per establir si el mètode ha funcionat correctament o no i actuar en conseqüència.

### Codi font de la classe

Donades les necessitats exposades, el codi font de la classe pot ser el que es mostra tot seguit. Entre les seves particularitats, a part dels aspectes anteriors, hi ha les crides internes al mètode `generarFitxerInicial`, que comprova si el el fitxer de puntuacions existeix, i si no és el cas, el crea des de zero. Això és més eficient que no pas dir que hi ha un error i no poder fer-hi res. Sobretot en el cas que s'acaba de jugar una partida on s'ha obtingut una màxima puntuació, ja que aquesta es perdria! Sempre és millor corregir un error si és possible, en lloc de simplement anunciar-lo i no fer-hi res més.

També, donades les particularitats dels fitxers seqüencials, observeu en el mètode

`entrarPuntuacio` com cal generar un fitxer temporal, ja que no es pot sobreescriure directament l'original.

```java
package joc.arena.fitxers;
import java.io.File;
import java.io.PrintStream;
import java.util.Scanner;
public class Ranquing {
   //Nom fitxer com a constant
   public static final String RANQUING = "Ranquing";
   /** Crea el fitxer de puntuacions inicial
    * @return 0 si tot correcte, -1 si error
    */
   public int generarFitxerInicial() {
    try {
      File ranquing = new File(RANQUING);
      if (ranquing.isFile() == false) {
        PrintStream ps = new PrintStream(ranquing);
        for (int i = 0; i < 10; i++) {
          ps.println("IOC " + (10 - i)*10);
        }
        ps.close();
      }
      return 0;
    } catch (Exception e) {
      //Propagació d'error
      return -1;
    }
   }
  /** Donada una puntuació, estableix la seva posició al fitxer
   * @param punts Punts que cal comprovar
   * @return Posició per ala puntuació. -1 si error.
   */
  public int cercarRanking(int punts) {
    try {
      int err = generarFitxerInicial();
      if (err == -1) {
        return -1;
      }
      File ranquing = new File(RANQUING);
      Scanner lector = new Scanner(ranquing);
      int pos = 0;
      while (pos < 10) {
        lector.next();
        if (lector.hasNextInt()) {
          int ranqPts = lector.nextInt();
          if (punts > ranqPts) {
            return pos;
          }
        } else {
          //Error en el format del fitxer
          return -1;
        }
        pos++;
      }
      lector.close();
      return pos;
    } catch (Exception e) {
      return -1;
    }
  }
  /** Insereix una puntuació al ranquing
   * @param inicials Inicials del jugador
   * @param punts Puntuació assolida
   * @param pos Posició dins el ranquing
   * @return 0 si tot correcte, -1 si error.
   */
  public int entrarPuntuacio(String inicials, int punts, int pos) {
    try {
      int err = generarFitxerInicial();
```

```
68        if (err == -1) {
69          return -1;
70        }
71        File ranquing = new File(RANQUING);
72        Scanner lector = new Scanner(ranquing);
73        File tmp = new File (RANQUING + ".tmp");
74        PrintStream ps = new PrintStream(tmp);
75        //Reescriure anteriors a posicio
76        for(int i = 0; i < pos; i++) {
77          String txt = lector.nextLine();
78          ps.println(txt);
79        }
80        //Escriure nova puntuació
81        ps.println(inicials + " " + punts);
82        //Reescriure posteriors a posició
83        for(int i = pos + 1; i < 10; i++) {
84          String txt = lector.nextLine();
85          ps.println(txt);
86        }
87        ps.close();
88        lector.close();
89        //S'esborra fitxer antic i es posa el nou
90        ranquing.delete();
91        tmp.renameTo(ranquing);
92        return 0;
93      } catch (Exception e) {
94        return -1;
95      }
96    }
97    /** Llegeix les puntuacions i les formata com una cadena de text
98     * @return Cadena de text resultant. null si hi ha error
99     */
100   public String llegirRanquing() {
101     try {
102       int err = generarFitxerInicial();
103       if (err == -1) {
104         return null;
105       }
106       String txtRanquing = "Ranquing de puntuacions\n----------\n";
107       File ranquing = new File(RANQUING);
108       Scanner lector = new Scanner(ranquing);
109       for (int i = 0; i < 10; i++) {
110         //Llegir inicials
111         txtRanquing = txtRanquing + lector.next();
112         //Llegir punts
113         if (lector.hasNextInt()) {
114           txtRanquing = txtRanquing + "\t" + lector.nextInt() + "\n";
115         } else {
116           //Error en el format del fitxer
117           return null;
118         }
119       }
120       lector.close();
121       return txtRanquing;
122     }  catch (Exception e) {
123       return null;
124     }
125   }
126 }
```

### 3.3.2 La classe Bestiari

La classe `Bestiari` és l'encarregada de generar els lluitadors, tant el jugador a l'inici del joc com els successius adversaris. A la versió original tot està

emmagatzemat en el codi font del programa. Ara es vol fer una nova versió, pertanyent a un *package* diferent, que obtingui tota la informació des d'un fitxer. D'aquesta manera, és possible usar diferents llistes d'adversaris, o afegir-ne de nous, sense haver de modificar el codi font del programa. Només cal canviar aquest fitxer.

### Elecció de tipus de fitxer i accés

En aquest cas, el nom del fitxer on tenir desades les dades dels adversaris serà "Adversaris", i també s'usarà una ruta relativa.

Abans de poder escriure el codi font cal tenir ben clar quin serà el format del fitxer i la seva estructura. En aquest cas, us trobeu que heu d'adaptar una classe en la qual tot s'estructura dins un *array* originalment, i el que es vol és treballar sobre un fitxer. El tipus de fitxer que s'assembla més a un *array* en la seva manera de treballar, per poder accedir a les seves posicions de manera directa, és un fitxer d'accés relatiu. Per tant, s'usarà aquest sistema de tractament de dades, cosa que implica que cal usar un fitxer orientat a byte.

Amb un fitxer relatiu orientat a byte, fins i tot és possible plantejar cerques dicotòmiques directament sobre aquest.

Cal definir detalladament l'estructura del fitxer i com s'agrupen els conjunts de valors emmagatzemats (els atributs dels adversaris). El més sensat és proposar una adaptació més o menys directa de l'*array* usat en la versió original. Ara bé, per poder treballar amb un fitxer orientat a byte de manera relativa, com si fos un *array*, és important que cada conjunt de dades tingui una mida fixa (en aquest cas, cada adversari). Això vol dir que cal fitar la mida del nom. No pot ser de qualsevol llargària. Per a aquest programa, la mida es fitarà a 15 caràcters. En el cas de noms amb menys de 15 caràcters, la resta de bytes s'omplen tots a 0 fins a arribar als 15 caràcters (30 bytes).

D'aquesta manera, cada adversari ocupa sempre 50 bytes: 15*2 (nom, compost de 15 caràcters de 2 bytes) + 5*4 (els 5 atributs, que són enters de 4 bytes). La taula 3.2 mostra un resum de l'estructura.

**Taula 3.2.** Estructura de les dades associades a un adversari

| Bytes 0-29 | Bytes 30-33 | Bytes 34-37 | Bytes 38-41 | Bytes 42-45 | Bytes 46-49 |
|---|---|---|---|---|---|
| Nom (caràcters) | Nivell (enter) | Punts (enter) | Vida Màx (enter) | Atac (enter) | Defensa (enter) |

Donada aquesta estructura, la figura 3.1 mostraria un exemple de visualització del fitxer mitjançant un editor hexadecimal. Les dades relatives al primer adversari (de nom "Nan") estan ombrejades. Fixeu-vos com els primers 34 bytes corresponen al nom. Els 6 primers són els caràcters del nom pròpiament i la resta són a 0. A partir del byte número 30, hi ha cinc enters, cadascun de 4 bytes, associats als seus atributs, de manera que:

- Nivell = 0x00000001
- Punts = 0x00000019 (25 en decimal)
- Vida = 0x00000008

- Atac = 0x00000003
- Defensa = 0x00000003

**Figura 3.1.** Contingut del fitxer d'adversaris visualitzat amb un editor hexadecimal

```
00000000 00 4E 00 61 00 6E 00 00 00 00 00 00 00 00 00 00 00 00 .N.a.n..............
00000014 00 00 00 00 00 00 00 00 00 00 00 00 01 00 00 00 19 00 00 ...................
00000028 00 08 00 00 00 03 00 00 00 03 00 4B 00 6F 00 62 00 6F 00 6C .........K.o.b.o.l
0000003C 00 64 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .d..................
00000050 00 00 00 01 00 00 00 1E 00 00 00 0A 00 00 00 04 00 00 00 02 ....................
00000064 00 4F 00 72 00 63 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .O.r.c..............
00000078 00 00 00 00 00 00 00 00 00 00 00 00 00 02 00 00 00 23 00 00 ..................#..
0000008C 00 0C 00 00 00 04 00 00 00 03 00 50 00 72 00 6F 00 66 00 75 ...........P.r.o.f.u
000000A0 00 6E 00 64 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .n.d................
000000B4 00 00 00 02 00 00 00 28 00 00 00 0E 00 00 00 03 00 00 00 04 .......(............
000000C8 00 42 00 72 00 75 00 69 00 78 00 6F 00 74 00 20 00 4D 00 61 .B.r.u.i.x.o.t. .M.a
000000DC 00 6C 00 76 00 61 00 74 00 00 00 00 00 03 00 00 00 2D 00 00 .l.v.a.t.........-..
```

### Codi font de la classe

Canviar un mòdul d'un programa hauria de ser com canviar un moble modular. Font: Bercik

Aquest cas és una mica diferent del rànquing de puntuacions, ja que no es tracta d'afegir un mòdul nou. Es tracta de reemplaçar un mòdul amb unes funcions concretes (gestionar adversaris a partir d'unes variables globals) per un amb unes altres (que aquesta gestió es faci usant un fitxer). Això és una situació molt interessant, ja que si s'apliquen els principis de modularitat correctament, les modificacions a la resta del programa haurien de ser mínimes.

Aquest és, precisament, el sentit de la modularitat. Poder canviar un mòdul (en aquest cas, una classe), per un altre sense haver de tocar res més. Tal com es faria quan es canvia una calaixera modular o un mòdul de memòria de l'ordinador. Treieu l'antic, poseu el nou, i tot continua funcionant sense haver de tocar res més. Si s'assoleix això, tot s'ha fet perfectament.

L'estratègia per assolir aquesta fita sempre és la mateixa. Intentar mantenir els mateixos mètodes, amb els mateixos paràmetres, que hi havia a la classe original, i només modificar el bloc de codi que contenen. D'aquesta manera, no cal esmenar el codi on hi ha invocacions en qualsevol dels mètodes originals, ja que la seva definició serà la mateixa a la classe nova que a l'antiga. Si és necessari, es poden afegir nous mètodes, però és imprescindible mantenir el format dels que ja hi havia abans del canvi.

Per tant, fixeu-vos com, a la nova classe, els mètodes que hi ha són exactament els mateixos que hi havia inicialment, mantenint el seu format (nom i paràmetres). Tot i així, s'han creat alguns mètodes nous per tal reutilitzar codi (per exemple, `crearAdversari` o `llegirNom`).

```
1  package joc.arena.fitxers;
2  import java.io.File;
3  import java.io.RandomAccessFile;
4  import java.util.Random;
5  import joc.arena.regles.Lluitador;
6  public class Bestiari {
7    //Constant amb el nom del fitxer d'adversaris
8    private static final File ADVERSARIS = new File("Adversaris");
9    //Jugador: ID = 0
10   private int[] jugador = {0,  1,  0, 10, 10, 3, 3, 3, 3};
11   private Lluitador lluitador = new Lluitador();
12   //Tots els mètodes mantenen la seva declaració (nom i paràmetres)
```

```
13    /** Genera un nou jugador
14     *
15     * @return Un array amb les dades d'un jugador inicial
16     */
17    public int[] generarJugador() {
18      lluitador.renovar(jugador);
19      return jugador;
20    }
21    /** Donat un nom, genera l'adversari corresponent. Si aquest nom no existeix,
22     * es genera a l'atzar.
23     *
24     * @param nomAdv Nom de l'adversari a obtenir
25     * @return El Lluitador amb aquest nom, o null si no existeix
26     */
27    public int[] cercarAdversari(String nomAdv) {
28      try {
29        int[] adversari = null;
30        RandomAccessFile raf = new RandomAccessFile(ADVERSARIS, "r");
31        long numAdv = ADVERSARIS.length()/50;
32        for (int i = 0; i < numAdv; i++) {
33          raf.seek(50*i);
34          String nom = llegirNom(raf);
35          if (nom.equalsIgnoreCase(nomAdv)) {
36            adversari = crearAdversari(raf,i);
37          }
38        }
39        raf.close();
40        return adversari;
41      } catch (Exception e) {
42        return null;
43      }
44    }
45    /** Obté l'adversari que hi ha en un ordre concret del fitxer,
46     * en fomat array.
47     *
48     * @param raf Fitxer relatiu d'on llegir-lo
49     * @param pos Posició de l'adversari dins del fitxer
50     * @return LLuitador llegit
51     */
52    public int[] crearAdversari(RandomAccessFile raf, int pos) {
53      try {
54        int[] adversari = new int[9];
55        raf.seek(pos*50 + 30);
56        adversari[0] = pos + 1;
57        adversari[1] = raf.readInt();
58        adversari[2] = raf.readInt();
59        adversari[3] = raf.readInt();
60        adversari[4] = adversari[3];
61        adversari[5] = raf.readInt();
62        adversari[6] = adversari[5];
63        adversari[7] = raf.readInt();
64        adversari[8] = adversari[7];
65        return adversari;
66      } catch (Exception e) {
67        return null;
68      }
69    }
70    /** Donat un fitxer relatiu orientat a byte, correctament posicionat,
71     * llegeix el nom de l'adversari.
72     *
73     * @param raf Fitxer a tractar
74     * @return Nom llegit, o null si hi ha error
75     */
76    public String llegirNom(RandomAccessFile raf) {
77      try {
78        String nom = "";
79        char c = raf.readChar();
80        while (c != 0x0000) {
81          nom = nom + c;
82          c = raf.readChar();
```

```
      }
      return nom;
    } catch (Exception e) {
      return null;
    }
  }
 /**Donat un nivell, genera l'adversari corresponent a l'atzar. Es tracta
   * d'un adversari que sigui almenys d'aquest nivell
   *
   * @param nivell Nivell proper al de l'adversari a obtenir
   * @return Un adversari
   */
  public int[] triarAdversariAtzar(int nivell) {
    try {
      RandomAccessFile raf = new RandomAccessFile(ADVERSARIS,"r");
      Random rnd = new Random();
      int numAdv = (int)raf.length()/50;
      int[] adversari = null;
      boolean cercar = true;
      while (cercar) {
        int i = rnd.nextInt(numAdv);
        adversari = crearAdversari(raf, i);;
        int nivellAdv = lluitador.llegirNivell(adversari);
        int dif = nivell - nivellAdv;
        if ((dif >= -1)&&(dif <= 1)) {
          cercar = false;
        }
      }
      raf.close();
      return adversari;
    } catch (Exception e)  {
      return null;
    }
  }
  /** Transforma un identificador de Lluitador al seu nom.
   *
   * @param id Identificador
   * @return La cadena de text amb el nom.
   */
  public String traduirIDANom(int id) {
    if (id == 0)  {
      return "Aventurer";
    }
    id--;
    try {
      RandomAccessFile raf = new RandomAccessFile(ADVERSARIS, "r");
      int pos = 50*id;
      String nom = "DESCONEGUT";
      if (pos < raf.length()) {
        raf.seek(pos);
        nom = llegirNom(raf);
      }
      raf.close();
      return nom;
    } catch (Exception e) {
      return "DESCONEGUT";
    }
  }
  /** Diu si hi ha el fitxer d'adversaris
   *
   * @return Si existeix (true) o no (false)
   */
  public boolean existeixFitxer() {
    return ADVERSARIS.isFile();
  }
}
```

### 3.3.3 La classe auxiliar EditorBestiari

**Eines de desenvolupament de jocs**

És molt habitual que els programadors de videojocs tinguin eines auxiliars per generar els fitxers de dades dels jocs que estan creant. Aquestes s'anomenen eines de desenvolupament de jocs.

El problema dels fitxers orientats a byte és que no resulten fàcils d'editar. Per això, per a aquest cas, pot resultar útil disposar d'un programa auxiliar que serveixi d'editor de la llista d'adversaris, de manera que sigui fàcil manipular-la. Aquest pot visualitzar, afegir i esborrar entrades al fitxer binari.

#### Codi font de la classe

Tot seguit es mostra el codi font d'aquest programa auxiliar, creat monolíticament en una sola classe, però aplicant disseny descendent. El fitxer d'adversaris es pot editar partint des de zero, o bé des d'un ja existent. Per simplificar el seu codi, totes les operacions d'afegir o eliminar treballen sobre el final del fitxer del fitxer.

```
1  package joc.arena.fitxers;
2  import java.io.File;
3  import java.io.RandomAccessFile;
4  import java.util.Scanner;
5  public class EditorBestiari {
6    public static final int MAX_CHAR_NOM = 15;
7    public static final int MIDA_BYTES_ADV = 50;
8    public static void main (String[] args) {
9      EditorBestiari programa = new EditorBestiari();
10     programa.inici();
11   }
12   public void inici() {
13     File fitxer = preguntarFitxer();
14     boolean executar = true;
15     while (executar) {
16       executar = tractarMenu(fitxer);
17     }
18   }
19   /** Pregunta a l'usuari el nom del fitxer a editar.
20    * @return Ruta al fitxer a editar.
21    */
22   public File preguntarFitxer() {
23     Scanner lector = new Scanner(System.in);
24     System.out.print("Nom del fitxer a editar: ");
25     String nom = lector.nextLine();
26     File fitxer = new File(nom);
27     return fitxer;
28   }
29   /** Fa el tractament del menú de l'usuari.
30    * @param fitxer Fitxer que cal tractar
31    * @return Si cal seguir executant del programa (true) o encara no (false)
32    */
33   public boolean tractarMenu(File fitxer) {
34     mostrarFitxer(fitxer);
35     System.out.println("————————————————————————————————");
36     System.out.println("[A]fegir\t[E]liminar darrer\t[S]ortir");
37     boolean preguntar = true;
38     Scanner lector = new Scanner(System.in);
39     while (preguntar) {
40       System.out.print("Accio: ");
41       String resposta = lector.nextLine();
42       if ("A".equalsIgnoreCase(resposta)) {
43         afegirAdversari(fitxer);
44         preguntar = false;
45       } else if ("E".equalsIgnoreCase(resposta)) {
46         eliminarAdversari(fitxer);
47         preguntar = false;
```

```
48        } else if ("S".equalsIgnoreCase(resposta)) {
49          return false;
50        } else {
51          System.out.print("Accio incorrecta...");
52        }
53      }
54      return true;
55    }
56    /** Mostra el contingut del fitxer per pantalla
57     * @param fitxer Ruta al fitxer a mostrar
58     */
59    public void mostrarFitxer(File fitxer) {
60      try {
61        if (fitxer.isFile() == false) {
62          System.out.println("Encara no s'ha creat el fitxer.");
63        } else {
64          RandomAccessFile raf = new RandomAccessFile(fitxer, "r");
65          long numAdversaris = fitxer.length()/MIDA_BYTES_ADV;
66          if (numAdversaris == 0) {
67            System.out.println("El fitxer és buit.");
68          } else {
69            for (int i = 0; i <numAdversaris; i++) {
70              String nom = llegirNom(raf);
71              System.out.print(nom);
72              for (int z = 0; z < (MAX_CHAR_NOM – nom.length()); z++) {
73                System.out.print(" ");
74              }
75              //Cada adversari coupa 50 bytes. El tros del nom n'ocupa 15*2
76              raf.seek(i*MIDA_BYTES_ADV + MAX_CHAR_NOM*2);
77              System.out.print(": \tNivell: " + raf.readInt());
78              System.out.print(" (punts: " + raf.readInt() + ")");
79              System.out.print("\tVIDA: " + raf.readInt());
80              System.out.print("\tATAC: " + raf.readInt());
81              System.out.println("\tDEFENSA: " + raf.readInt());
82            }
83          }
84          raf.close();
85        }
86      } catch (Exception e) {
87        System.out.println("Error accedint al fitxer!");
88      }
89    }
90    /** Donat un fitxer relatiu orientat a byte, correctament posicionat,
91     * llegeix el nom de l'adversari.
92     * @param raf Fitxer a tractar
93     * @return Nom llegit, o null si hi ha error
94     */
95    public String llegirNom(RandomAccessFile raf) {
96      try {
97        String nom = "";
98        char c = raf.readChar();
99        while (c != 0x0000) {
100         nom = nom + c;
101         c = raf.readChar();
102       }
103       return nom;
104     } catch (Exception e) {
105       return null;
106     }
107   }
108   /** Escriu un nou adversari al final del fitxer, preguntant tot el que calgui
109    * a l'usuari.
110    * @param fitxer Ruta al fitxer on cal afegir l'adversari.
111    */
112   public void afegirAdversari(File fitxer) {
113     try {
114       RandomAccessFile raf = new RandomAccessFile(fitxer, "rw");
115       //Es posa al final de tot
116       raf.seek(fitxer.length());
117       System.out.println("Escriu el nom de l'adversari (màx. 12 lletres): ");
```

```
118       Scanner lector = new Scanner(System.in);
119       String nom = lector.nextLine();
120       //Escriure el nom (màx. 15 caràcters)
121       int err = escriureNom(nom, raf);
122       if (err == -1) {
123         System.out.println("Error escrivint dades al fitxer " + fitxer);
124       } else {
125         //Escriure valors - Nivell:XP:PV:Max PV:Atac:Max Atac:max Defensa
126         System.out.print("Escriu els seus atributs, separats per espais. ");
127         System.out.println ("(5 enters = Nivell Punts PV Atac Defensa):");
128         int[] valors = llegirValors(lector);
129         for (int i = 0; i < valors.length; i++) {
130           raf.writeInt(valors[i]);
131         }
132       }
133       raf.close();
134     } catch (Exception e) {
135       System.out.println("Error escrivint dades al fitxer " + fitxer);
136     }
137   }
138   /** Llegeix cinc valors de tipus enter des del teclat.
139    * @param lector Scanner que llegeix del teclat
140    * @return Els cinc valors llegits, ordenadament
141    */
142   public int[] llegirValors(Scanner lector) {
143     int[] valors = new int[5];
144     boolean preguntar = true;
145     while(preguntar) {
146       int numLlegits = 0;
147       for (int i = 0; i < valors.length; i++) {
148         if (lector.hasNextInt()) {
149           valors[i] = lector.nextInt();
150           numLlegits++;
151         } else {
152           lector.next();
153         }
154       }
155       if (numLlegits == 5) {
156         preguntar = false;
157       } else {
158         System.out.println("Els 5 valors no han estat correctes.");
159       }
160     }
161     lector.nextLine();
162     return valors;
163   }
164   /** Donat un nom en forma de cadena de text, l'escriu a un fitxer orientat a
165    * byte. Com a màxim el nom pot tenir 15 caràcters. La resta, s'omple a 0.
166    * @param nom Nom a escriure
167    * @param raf Fitxer relatiu correctament posicionat per a l'escriptura
168    * @return Si ha funcionat (0) o no (-1)
169    */
170   public int escriureNom(String nom, RandomAccessFile raf) {
171     try {
172       int numChars = nom.length();
173       if (numChars > MAX_CHAR_NOM) {
174         numChars = MAX_CHAR_NOM;
175       }
176       for (int i = 0; i < numChars; i++) {
177         raf.writeChar(nom.charAt(i));
178       }
179       char blank = 0x0000;
180       for (int i = 0; i < (MAX_CHAR_NOM - numChars); i++) {
181         raf.writeChar(blank);
182       }
183       return 0;
184     } catch (Exception e) {
185       return -1;
186     }
187   }
```

```
188   /** Elimina el darrer adversari en un fitxer.
189    * @param fitxer Ruta del fitxer a modificar
190    */
191   public void eliminarAdversari(File fitxer) {
192     try {
193       RandomAccessFile raf = new RandomAccessFile(fitxer,"rw");
194       long novaMida = fitxer.length() - MIDA_BYTES_ADV;
195       //Eliminar el darrer adversari és esborrar els darrers 50 bytes...
196       if (novaMida >= 0) {
197         raf.setLength(fitxer.length() - MIDA_BYTES_ADV);
198       }
199       raf.close();
200     } catch (Exception e) {
201       System.out.println("Error esborrant dades al fitxer " + fitxer);
202     }
203   }
204 }
```

## 3.4 Esmenes als mòduls originals

Un cop s'han generat els nous mòduls que amplien el programa original de manera que algunes de les seves tasques depenguin de la lectura o escriptura de fitxers, cal modificar el codi original per tal que s'usin els nous mòduls. Una correcta aplicació de la modularitat hauria de minimitzar el nombre de canvis al codi i fer que aquests estiguin molt ben localitzats. De fet, el cas ideal seria que només cal afegir nou codi, però no modificar, o fins i tot eliminar, codi ja existent.

### 3.4.1 A la classe EntradaTeclat

Noteu que el canvi de la classe `Bestiari` no afecta en absolut aquesta classe, tot i que la fa servir. Això vol dir que s'ha aplicat correctament la modularitat.

**El mètode "trim" elimina els espais a dreta i esquerra d'un text.**

La introducció d'una màxima puntuació al rànquing implica poder demanar a l'usuari, si es compleixen les condicions necessàries, que introdueixi les seves inicials. Si es vol seguir estrictament el plantejament modular de l'aplicació, això vol dir que cal un nou mètode a la classe `EntradaTeclat`. Aquest ha de comprovar que, efectivament, es tracta d'unes inicials (en aquest cas, es considera que tres lletres).

```
1  //Ara cal importar la nova versió
2    import joc.arena.fitxers.Bestiari;
3    //Nou mètode
4    /** Pregunta les inicials del jugador si assoleix una màxima puntuació.
5     * @return Inicials (cadena de text amb 3 caràcters)
6     */
7    public String preguntarInicials() {
8      Scanner lector = new Scanner(System.in);
9      System.out.println("has assolit una màxima puntuació!!");
10     String inicials = "";
11     boolean llegir = true;
12     while (llegir) {
13       System.out.print("Escriu les teves inicials: ");
14       inicials = lector.nextLine();
15       //Aquest mètode elimina espais laterals
16       inicials = inicials.trim();
17       if (inicials.length() == 3) {
18         llegir = false;
19       } else {
20         System.out.print("Entrada incorrecta. ");
21       }
22     }
23     return inicials.toUpperCase();    //Sempre es posaran en majúscula
24   }
```

### 3.4.2 A la classe SortidaPantalla

Un altre nou aspecte del programa és que cal mostrar el rànquing de puntuacions en acabar el joc. Donat el plantejament de la classe `Ranquing`, aquesta tasca és molt simple i es podria dur a terme directament a `JocArena`. Tot i així, per ser coherents amb el fet que els mètodes que mostren diverses línies de text per pantalla, com l'estat del lluitador, són a aquesta classe, se serà estricte en l'aplicació de la modularitat tal com s'ha plantejat a l'aplicació originalment.

Com abans, noteu que el canvi de la classe `Bestiari` no afecta en absolut a aquesta classe, tot i que la fa servir. També observeu ara com es comprova la propagació d'errors des del mètode `llegirRanquing`, mirant si s'ha avaluat `null` o no.

```
1  //Ara cal importar la nova versió
2    import joc.arena.fitxers.Bestiari;
3    //Nou mètode
4    /** Mostra per pantalla la llista de puntuacions
5     */
6    public void mostrarRanking() {
7      Ranquing rnk = new Ranquing();
8      String s = rnk.llegirRanquing();
9      if (s == null) {
10       System.out.println("Hi ha un error en els fitxers de puntuacions!");
11     } else {
12       System.out.println(s);
13     }
14   }
```

### 3.4.3 A la classe JocArena

A l'apartat d'annexos dels materials disposeu del codi font complet amb la solució final d'aquest exemple.

En aquesta classe, només cal modificar el mètode `inici` per tal que, el programa prevegi les noves funcionalitats. D'una banda, ara cal garantir que s'inicialitzen els valors dels adversaris a partir d'un fitxer. D'altra banda, en acabar la partida, cal que es comprovi si pertoca afegir una nova màxima puntuació al fitxer i realitzar les operacions corresponents al teclat i la pantalla (entrar inicials i mostrar rànquing actual).

Tot i que s'ha intentat minimitzar el nombre de canvis necessaris a causa de la nova classe `Bestiari`, cal alguna modificació, ja que almenys cal controlar si el fitxer d'adversaris existeix abans de fer res. A part d'això, el canvi de classe no afecta en res més.

```
1  //Ara cal importar la nova versió
2    import joc.arena.fitxers.Bestiari;
3    public void inici() {
4      //Mirar si hi ha fitxer d'adversaris
5      if (bestiari.existeixFitxer() == false) {
6        System.out.println("No hi ha el fitxer d'adversaris!");
7      } else {
8        //Codi original
9        sortida.mostarBenvinguda();
10       int[] jugador = bestiari.generarJugador();
```

```
11        int numCombat = 0;
12        boolean jugar = true;
13         while (jugar) {
14          numCombat++;
15          //Abans de cada combat es restaura el jugador
16          lluitador.restaurar(jugador);
17          //Inici d'un combat
18          System.out.println("*** COMBAT " + numCombat);
19          System.out.print("Estat actual del jugador: ");
20          sortida.mostrarLluitador(jugador);
21          System.out.println("**************************");
22          //S'obté l'adversari
23          int[] adversari = entrada.triarAdversari(lluitador.llegirNivell(jugador
                ));
24          //Combat
25          combatre(jugador, adversari);
26          //Fi
27          jugar = fiCombat(jugador, adversari);
28          if (numCombat == MAX_COMBAT) {
29            System.out.println("Has sobreviscut a tots els combats. Enhorabona
                  !!");
30          }
31        }
32        System.out.print("Estat final del jugador: ");
33        sortida.mostrarLluitador(jugador);
34        //Nou: comprovar si entra al rànquing (10 posicions)
35        Ranquing rnk = new Ranquing();
36        int punts = lluitador.llegirPunts(jugador);
37        int pos = rnk.cercarRanking(punts);
38        if (pos != -1) {
39          if (pos < 10) {
40            String inicials = entrada.preguntarInicials();
41            err = rnk.entrarPuntuacio(inicials, punts, pos);
42            if (err == -1) {
43              System.out.println("Error accedint al fitxer de puntuacions.");
44            }
45          }
46          sortida.mostrarRanking();
47        } else {
48          System.out.println("Error accedint al fitxer de puntuacions.");
49        }
50      }
51    }
```