CENTRE ESPECÍFIC
D´EDUCACIÓ A DISTÀNCIA DE
LA COMUNITAT VALENCIANA

UNITAT 11

FITXERS

PROGRAMACIÓ
CFGS DAW

Autors:
Carlos Cacho y Raquel Torres
Revisat per:
Lionel Tarazon - lionel.tarazon@ceedcv.es
Fco. Javier Valero – franciscojavier.valero@ceedcv.es
José Manuel Martí - josemanuel.marti@ceedcv.es

2021/2022

Llicència

Nomenclatura

Al llarg d'aquest tema s'utilitzaran diferents símbols per a distingir elements importants dins del contingut. Aquests símbols són:

 Important

 Atenció

 Interessant

ÍNDEX

1. 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, una recopilació de temperatures, les dates d'un calendari, etc. Les possibilitats són il·limitades. Totes aquestes tasques de manipulació i transformació es duen a terme normalment mitjançant l'emmagatzematge de les dades en variables, dins de la memòria de l'ordinador, per la qual cosa es poden aplicar operacions, ja siga mitjançant operadors o la invocació de mètodes.

Desgraciadament, totes aquestes variables només tenen vigència mentre el programa s'està executant. Una vegada 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 del programa. O quan el nombre de dades a tractar és xicotet i es pot preguntar a l'usuari. Ara bé, imaginen-se haver d'introduir les notes de tots els estudiants cada vegada que s'executa el programa per a gestionar-les. No té cap sentit. Per tant, en alguns casos, apareix la necessitat de poder registrar les dades en algun suport de memòria externa, per la qual cosa aquestes es mantinguen de manera persistent entre diferents execucions del programa, o fins i tot si s'apaga l'ordinador.

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

En aquesta unitat didàctica s'expliquen diferents classes de Java que ens permeten crear, llegir, escriure i eliminar fitxers i directoris, entre altres operacions. També s'introdueix la serialització d'objectes com a mecanisme de gran utilitat per a emmagatzemar objectes en fitxers per a després recuperar-los en temps d'execució.

2. GESTIÓ DE FITXERS

Entre les funcions d'un sistema operatiu està la d'oferir mecanismes genèrics per a gestionar sistemes d'arxius. Normalment, dins d'un sistema operatiu modern (o ja no tant modern), s'espera disposar d'alguna mena d'interfície o explorador per a poder gestionar arxius, ja siga gràficament o usant una línia de comandos de text. Si bé la forma en què les dades es guarden 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 d'arxius 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 d'arxius no és exclusiva de la interfície oferida pel sistema operatiu. Molts llenguatges de programació proporcionen biblioteques que permeten accedir directament als mecanismes interns que ofereix el sistema, per la qual cosa és possible crear codi font des del qual, amb les instruccions adequades, es poden realitzar operacions típiques d'un explorador d'arxius. De fet, les interfícies com un explorador d'arxius són un programa com qualsevol altre, el qual, usant precisament aquestes llibreries, permet que l'usuari gestione arxius fàcilment. Però és habitual trobar altres aplicacions amb la seua pròpia interfície per a gestionar arxius, encara que només siga per a poder seleccionar què cal carregar o guardar 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 adequats definits d'aquestes classes és possible dur a terme pràcticament qualsevol tasca sobre el sistema d'arxius.

2.1. La classe File

La peça més bàsica per a poder operar amb arxius, independentment del seu tipus, en un programa Java és la **classe File**. Aquesta classe pertany al *package java.io*. Per tant serà necessari importar-la abans de poder usar-la.

```
import java.io.File;
```

Aquesta classe permet manipular qualsevol aspecte vinculat al sistema de fitxers. El seu nom ("arxiu", en anglés) és una mica enganyós, ja que no es refereix exactament a un arxiu.

> **La classe File representa una ruta dins del sistema d'arxius**.

Serveix per a realitzar operacions tant sobre rutes al sistema d'arxius que ja existisquen com no existents. A més, es pot usar tant per a manipular arxius com directoris.

Com qualsevol altra classe **cal instanciar-la perquè siga possible invocar els seus mètodes**. El constructor de File rep com a argument una cadena de text corresponent a la ruta sobre la qual es volen dur a terme les operacions.

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

Una ruta és la forma general d'un **nom d'arxiu o carpeta**, per la qual cosa identifica únicament la seua localització en el sistema d'arxius.

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

Un aspecte important a tindre present en inicialitzar File és tindre 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, els diferents sistemes operatius usen diferents separadors dins de les rutes. Per exemple, els sistemes Unix usen la barra ("/") mentre que el Windows la inversa ("\").

- Exemple de ruta Unix: /usr/bin
- Exemple de ruta Windows: C:\Windows\System32

De totes maneres Java ens permet utilitzar la barra d'Unix ("/") per a representar rutes en sistemes Windows. Per tant, és possible utilitzar sempre aquest tipus de barra independentment del sistema, per simplicitat.

És important entendre que **un objecte representa una única ruta** del sistema de fitxers. Per a operar amb diferents rutes vegada caldrà crear i manipular diversos objectes. Per exemple, en el següent codi s'instancien tres objectes File diferents.

```
File carpetaFotos = new File("C:/Fotos");
File unaFoto = new File("C:/Fotos/Foto1.png");
File otraFoto = new File("C:/Fotos/Foto2.png");
```

2.2. Rutes absolutes i relatives

En els exemples emprats fins al moment per a crear objectes del tipus File s'han usat rutes absolutes, ja que és la manera de deixar més clar a quin element dins del sistema d'arxius, ja siga arxiu o carpeta, s'està fent referència.

> Una **ruta absoluta** és aquella que **es refereix a un element a partir de l'arrel** del sistema de fitxers. Per exemple "C:/Fotos/Foto1.png"

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 en un sistema d'arxius de Windows:

- C:\Fotos\Viatges (ruta a una carpeta)
- M:\Documents\Unitat11\apartat1 (ruta a una carpeta)
- N:\Documents\Unitat11\apartat1\Activitats.txt (ruta a un arxiu)

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

- /Fotos/Viatges (ruta a una carpeta)
- /Documents/Unitat11/apartat1 (ruta a una carpeta)
- /Documents/Unitat11/apartat1/Activitats.txt (ruta a un arxiu)

Al instanciar objectes 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 a indicar amb tota claredat quin element dins del sistema d'arxius s'està manipulant, hi ha casos que el seu ús comporta unes certes complicacions**. Suposem que s'ha fet un programa en el qual es duen a terme operacions sobre el sistema d'arxius. Una vegada funciona, li deixa el projecte Java a un amic que el copia en el seu ordinador dins d'una carpeta qualsevol i l'obri amb el seu entorn de treball. Perquè el programa li funcione perfectament abans serà necessari que en el seu ordinador hi haja exactament les mateixes carpetes que usa en la seua màquina, tal com estan escrites en el codi font del seu 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 haja de treballar amb una estructura del sistema d'arxius exactament igual allà on s'execute, la qual cosa no és molt còmode.

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

Una **ruta relativa** és aquella que **no inclou l'arrel** i per això es considera que **part des del directori de treball** de l'aplicació. Aquesta carpeta pot ser diferent cada vegada que s'executa el programa.

Quan un programa s'executa per defecte **se li assigna una carpeta de treball**. Aquesta carpeta **sol ser la carpeta des d'on es llança el programa**. En el cas d'un programa en Java executat a través d'un IDE (com NetBeans), la carpeta de treball sol ser la mateixa carpeta on s'ha triat guardar els arxius del projecte.

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

- Viatges
- Unitat11\apartat1
- Unitat11\apartat1\Activitats.txt

Una ruta relativa sempre inclou el directori de treball de l'aplicació com a part inicial malgrat no haver-se escrit. El tret distintiu és que el directori de treball pot variar. Per exemple, l'element al qual es refereix el següent objecte File varia segons el directori de treball.

```
File f = new *File ("Unitat11/apartat1/Activitats.txt");
```

Directori de treball	**Ruta real**
C:/Projectes/Java	C:/Projectes/Java/Unitat11/apartat1/Activitats.txt
X:/Unitats	X:/Unitats/Unitat11/apartat1/Activitats.txt
/Programes	/Programes/Unitat11/apartat1/Activitats.txt

Aquest mecanisme permet facilitar la portabilitat del programari entre diferents ordinadors i sistemes operatius, ja que només és necessari que els arxius i carpetes romanguen en la mateixa ruta relativa al directori de treball. Vegem-ho amb un exemple:

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

Donada aquesta ruta relativa, basta garantir que el fitxer "*Activdades.txt" estiga sempre en el mateix directori de treball de l'aplicació, qualsevol que siga aquest i independentment del sistema operatiu utilitzat (en un ordinador pot ser "C:\Programes" i en un altre "/Java"). En qualsevol de tots aquests casos, la ruta sempre serà correcta. De fet, encara més. Note's com **les rutes relatives a Java permeten crear codi independent del sistema operatiu**, ja que no és necessari especificar un format d'arrel lligada a un sistema d'arxius concret ("C:", "D:", "/", etc.).

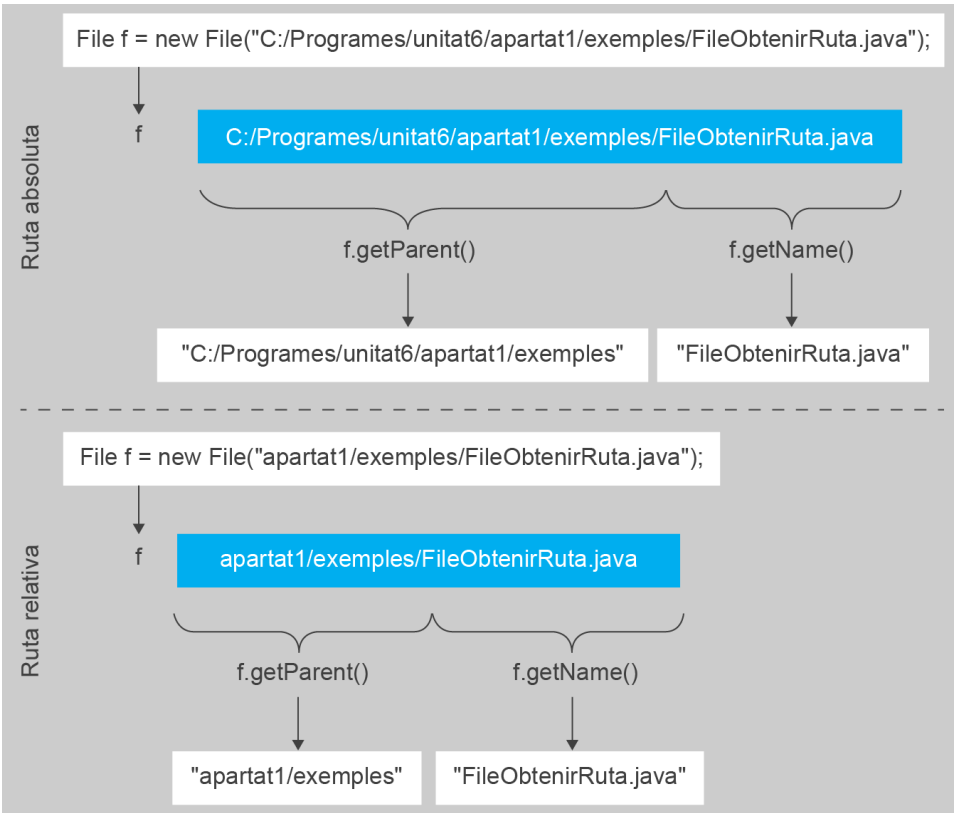
2.3. Mètodes de la classe File

File ofereix diversos mètodes per a poder manipular el sistema d'arxius o obtindre informació a partir de la seua ruta. Alguns dels més significatius per a entendre les funcionalitats es mostren a continuació, ordenats per tipus d'operació.

2.3.1. Obtenció de la ruta

Una vegada s'ha instanciat un objecte de tipus File, pot ser necessari recuperar la informació emprada durant la seua inicialització i conéixer en format text a quina ruta s'està referint, o almenys part d'ella.

- **String getParent()** retorna la ruta de la carpeta de l'element referit per aquesta ruta. Bàsicament la cadena de text resultant és idèntica a la ruta original, eliminant l'últim element. Si la ruta tractada es refereix a la carpeta arrel d'un sistema d'arxius ("C:\", "/", etc.), aquest mètode retorna *null*. En el cas de tractar-se d'una ruta relativa, aquest mètode no inclou la part de la carpeta de treball.
- **String getName()** retorna el nom de l'element que representa la ruta, ja siga una carpeta o un arxiu. És el cas invers del mètode getParent(), ja que el text resultant és només l'últim element.
- **String getAbsolutePath()** retorna la ruta absoluta. Si l'objecte File es va inicialitzar usant una ruta relativa, el resultat inclou també la carpeta de treball.



Vegem un exemple de com funcionen aquests tres mètodes. Observe's que les rutes relatives s'afigen a la ruta de la carpeta de treball (on es troba el projecte):

```
import java.io.File;

public class ProvaFitxers {

public static void main(String[] args) {
      // Dues rutes absolutes
      File carpetaAbs = new File("/home/lionel/fotos");
      File arxiuAbs = new File("/home/lionel/fotos/albania1.jpg");

      // Dues rutes relatives
      File carpetaRel = new File("treballs");
      File fitxerRel = new File("treballs/document.txt");

      // Mostrem les seues rutes
      mostrarRutes(carpetaAbs);
      mostrarRutes(arxiusAbs);
      mostrarRutes(carpetaRel);
      mostrarRutes(fitxerRel);
}

public static void mostrarRutes(File f) {
      System.out.println("getParent() : " + f.getParent());
      System.out.println("getName() : " + f.getName() + "\n");
      System.out.println("getAbsolutePath(): " + f.getAbsolutePath());
}
}
```

Aquest programa produeix l'eixida:

```
getParent() : /home/lionel
getName() : fotos
getAbsolutePath(): /home/lionel/fotos

getParent() : /home/lionel/fotos
getName() : albania1.jpg
getAbsolutePath(): /home/lionel/fotos/albania1.jpg

getParent() : null
getName() : treballs
getAbsolutePath(): /home/lionel/NetBeans/Fitxers/treballs

getParent() : treballs
getName() : document.txt
getAbsolutePath(): /home/lionel/NetBeans/Fitxers/treballs/document.txt
```

2.3.2. Comprovacions d'estat

Donada la ruta emprada per a inicialitzar una variable de tipus File, aquesta pot ser que realment existisca dins del sistema de fitxers o no, ja siga en forma d'arxiu 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. Retornarà *true* si existeix i *false* en cas contrari.

Normalment els arxius incorporen en el seu nom una extensió (.txt, .jpg, .mp4, etc.). Encara així, cal tindre en compte que l'extensió no és un element obligatori en el nom d'un arxiu, només s'usa com a mecanisme perquè tant l'usuari com alguns programes puguen discriminar més fàcilment el tipus d'arxius. Per tant, només amb el text d'una ruta no es pot estar 100% segur de si aquesta es refereix a un arxiu o una carpeta. Per a poder estar realment segurs es poden usar els mètodes següents:

- **boolean isFile()** comprova el sistema de fitxers a la recerca de la ruta i retorna *true* si existeix i és un fitxer. Retornarà *false* si no existeix, o si existeix però no és un fitxer.

- **boolean isDirectory()** funciona com l'anterior però comprova si és una carpeta.

Per exemple, el següent codi fa una sèrie de comprovacions sobre un conjunt de rutes. Per a poder provar-ho pots crear la carpeta "Temp" en l'arrel "C:". Dins, un arxiu anomenat "Document.txt" (pot estar buit) i una carpeta anomenada “Fotos”. Després de provar el programa pots eliminar algun element i tornar a provar per a veure la diferència.

```
public static void main(String[] args) {
    File temp = new File("C:/Temp");
    File fotos = new File("C:/Temp/Fotos");
    File document = new File("C:/Temp/Document.txt");
    System.out.println(temp.getAbsolutePath()+"existeix?"+temp.exists());
    mostrarEstat(fotos);
    mostrarEstat(document);
}

public static void mostrarEstat(File f) {
    System.out.println(f.getAbsolutePath()+"arxiu?"+f.isFile());
    System.out.println(f.getAbsolutePath()+"carpeta?"+f.isDirectory());
}
```

2.3.3. Propietats de fitxers

El sistema de fitxers d'un sistema operatiu emmagatzema diversitat d'informació sobre els arxius i carpetes que pot resultar útil conéixer: els seus atributs d'accés, la seua grandària, la data de modificació, etc. En general, totes les dades mostrades a accedir a les propietats de l'arxiu. Aquesta informació també pot ser consultada usant els mètodes adequats. Entre els més populars hi ha els següents:

- **long length()** retorna la grandària d'un arxiu en bytes. Aquest mètode només pot ser referenciat sobre una ruta que represente un arxiu, en cas contrari no es pot garantir que el resultat siga vàlid.

- **long lastModified()** retorna l'última data d'edició de l'element representat per aquesta ruta. El resultat es codifica en un únic número enter el valor del qual és el nombre de mil·lisegons que han passat des de l'1 de juny de 1970.

L'exemple següent mostra com funcionen aquests mètodes. Per a provar-los crea l'arxiu “Document.txt" en la carpeta "C:\Temp". Primer deixa l'arxiu buit i executa el programa. Després, amb un editor de text, escriu qualsevol cosa, guarda els canvis i torna a executar el programa. Observa com el resultat és diferent. Com a curiositat, fixa't en l'ús de la classe Dona't per a poder mostrar la data en un format llegible.

```
public static void main(String[] args) {
      File document = new File("C:/Temp/Document.txt");
      System.out.println(document.getAbsolutePath());

      long milisegons = document.lastModified();
      Date data = new Data(milisegons);

      System.out.println("Última modificació (ms) : " + milisegons);
      System.out.println("Última modificació (data): " + data);
      System.out.println("Grandària de l'arxiu: " + document.length());
}
```

Primera eixida:

```
C:/Temp/Document.txt
Última modificació (ms) : 1583025735411
Última modificació (data): Sun Mar 01 02.22:15 CET 2022
Grandària de l'arxiu: 0
```

Segona eixida:

```
C:/Temp/Document.txt
Última modificació (ms) : 1583025944088
Última modificació (data): Sun Mar 01 02.25:44 CET 2022
Grandària de l'arxiu: 7
```

2.3.4. Gestió de fitxers

El conjunt d'operacions més habituals en accedir a un sistema de fitxers d'un ordinador són les vinculades a la seua gestió directa: canviar de nom arxius, esborrar-los, copiar-los o moure'ls. Donat el nom d'una ruta, Java també permet realitzar aquestes accions.

- **boolean mkdir()** permet crear la carpeta indicada que no ha d'existir en la ruta en el moment d'invocar el mètode. Per exemple, donada una instància File amb la ruta "C:/Fotos/Albania" que no existeix, crearà la carpeta "Albania" dins de "C:/Fotos". Retorna *true* si s'ha creat correctament, en cas contrari retorna *false* (per exemple si la ruta l'incorrecta, la carpeta ja existeix o l'usuari no té permisos d'escriptura).

- **boolean delete()** esborra l'arxiu o carpeta indicada en la ruta. Es podrà esborrar una carpeta sol si està buida. Retorna *true* o *false* segons si l'operació s'ha pogut dur a terme.

- **boolean createNewFile()** crea un arxiu buit. Java ens obligarà a incloure la instrucció dins d'un context de captura d'excepcions per control intern de Java front a errors crítics.

Per a provar l'exemple que es mostra a continuació de manera que es puga veure com funcionen aquests mètodes, primer assegura't que en l'arrel de la unitat "C:" no hi ha cap carpeta anomenada "Temp" i execute el programa. Tot fallarà, ja que les rutes són incorrectes (no existeix "Temp"). Després, crea la carpeta "Temp" i en el seu interior crea un nou document anomenat "Document.txt" (pot estar buit). Executa el programa i veuràs que s'haurà creat una nova carpeta anomenada "Fotos". Si ho tornes a executar per tercera vegada podràs comprovar que s'haurà esborrat.

```
public static void main(String[] args) {

File fotos = new File("C:/Temp/Fotos");
File doc = new File("C:/Temp/Document.txt");

boolean mkdirFot = fotos.mkdir();

if (mkdirFot) {
     System.out.println("Creada carpeta " + fotos.getName() + "? " +
     mkdirFot);
 }
else {
     boolean delCa = fotos.delete();
     System.out.println("Esborrada carpeta " + fotos.getName() + "? " +
     delCa);
     boolean delAr = doc.delete();
     System.out.println("Esborrat arxiu " + doc.getName() + "? " + delAr);
 }
}
```

Des del punt de vista d'un sistema operatiu l'operació de "moure" un arxiu o carpeta no és més que canviar el seu nom des de la seua ruta original fins a una nova ruta destine. Per a fer això també hi ha un mètode.

- **boolean renameTo(File destí)** el nom d'aquest mètode és una cosa enganyosa ("canviar de nom", en anglés), ja que la seua funció real no és simplement canviar el nom d'un arxiu o carpeta, sinó canviar la ubicació completa. El mètode invoca l'objecte File amb la ruta origen (on es troba l'arxiu o carpeta), i se li dona com a argument altre objecte File amb la ruta destine. Retorna *true* o *false* segons si l'operació s'ha pogut dur a terme correctament o no (la ruta origen i destinació són correctes, no existeix ja un arxiu amb aquest nom en el destí, etc.). S'ha de tindre en compte que, en el cas de carpetes, és possible moure-les encara que continguen arxius.

Una vegada més, vegem un exemple. Dins de la carpeta "C:/Temp" crea una carpeta anomenada "Mitjana" i una altra anomenada "Fotos". Dins de la carpeta "Fotos" crea dos documents anomenats "Document.txt" i "Fotos.txt". Després d'executar el programa, observa com la carpeta "Fotos" s'ha mogut i ha canviat de nom, però manté en el seu interior l'arxiu "Fotos.txt". L'arxiu "Document.txt" s'ha mogut fins a la carpeta "Temp".

```
public static void main(String[] args) {

	File origenDir = new File("C:/Temp/Fotos");
	File destiDir = new File("C:/Temp/Mitjana/Fotografies");
	File origenDoc = new File("C:/Temp/Mitjana/Fotografies/Document.txt");
	File destiDoc = new File("C:/Temp/Document.txt");

	boolean res = origenDir.renameTo(destiDir);
	System.out.println("S'ha mogut i canviat de nom la carpeta? " + res);
	res = origenDoc.renameTo(destiDoc);
	System.out.println("S'ha mogut el document? " + res);
}
```

Com ja s'ha comentat aquest mètode també serveix, implícitament, per a canviar de nom arxius o carpetes. Si l'element final de les rutes origen i destinació són diferents, el nom de l'element, siga arxiu o carpeta, canviarà. Per a simplement canviar de nom un element sense moure'l de lloc, simplement la seua ruta pare seguirà sent exactament la mateixa. El resultat és que l'element de la ruta origen "es mou" en la mateixa carpeta on està ara, però amb un nom diferent.

Per exemple, si utilitzem "C:/Treballs/Doc.txt" com a ruta origen i "C:/Treballs/File.txt" com a ruta destine, l'arxiu "Doc.txt" canviarà de nom a "File.txt" però romandrà en la mateixa carpeta "C:/Treballs".

2.3.5. Llistat d'arxius

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

- **File[] listfiles()** retorna un vector de tipus File (File[]) amb tots els elements continguts en la carpeta (representats per objectes File, un per element). Perquè s'execute correctament la ruta ha d'indicar una carpeta. La grandària del vector serà igual al nombre d'elements que conté la carpeta. Si la grandària és 0, el valor retornat serà *null* i tota operació posterior sobre el vector serà errònia. L'ordre dels elements és aleatori (al contrari que en l'explorador d'arxius del sistema operatiu, no s'ordena automàticament per tipus ni alfabèticament).

Vegem un exemple. Abans d'executar-lo, crea una carpeta "Temp" en l'arrel de la unitat "C:". Dins crea o còpia qualsevol quantitat de carpetes o arxius.

```
public static void main(String[] args) {

        File dir = new File("C:/Temp");
        File[] llista = dir.listFiles();
        System.out.println("Contingut de " + dir.getAbsolutePath() + " :");

        // Recorrem el array i vam mostrar el nom de cada element
        for (int i = 0; i < llista.length; i++) {
              File f = llista[i];
              if (f.isDirectory()) {
                      System.out.println("[DIR] " + f.getName());
               }
              else {
                      System.out.println("[ARX] " + f.getName());
               }
        }
}
```

3. LECTURA I ESCRIPTURA DE FITXERS

Normalment les aplicacions que utilitzen arxius no estan centrades en la gestió del sistema d'arxius del seu ordinador. L'objectiu principal d'usar fitxers és poder emmagatzemar dades de manera que entre diferents execucions del programa, fins i tot en diferents equips, siga possible recuperar-los. El cas més típic és un editor de documents, que mentre s'executa s'encarrega de gestionar les dades relatives al text que està escrivint, però en qualsevol moment pot guardar-lo en un arxiu per a poder recuperar aquest text en qualsevol moment posterior, i afegir altres nous si fora necessari. El fitxer amb les dades del document el pot obrir tant en l'editor del seu ordinador com en el d'un altre company o companya.

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

> Es denomina **accés seqüencial** al processament d'un conjunt d'elements de manera que només és possible accedir a ella d'acord amb la seua ordre d'aparició. Per a processar un element és necessari processar primer tots els elements anteriors.

Java, juntament amb altres llenguatges de programació, diferència entre dos tipus d'arxius segons com es representen els valors emmagatzemats en un arxiu.

> 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 amb el seu format en binari, sense cap separació.

En aquesta unitat didàctica només veurem el processament de fitxers orientats a caràcter.

3.1. Fitxers orientats a caràcter

Un fitxer orientat a caràcter no és més que un document de text, com el que podria generar amb qualsevol editor de text simple. Els valors estan emmagatzemats segons la seua representació en cadena de text, exactament en el mateix format que ha usat fins ara per a 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. Encara que aquests valors es puguen distribuir en línies de text diferents, conceptualment, es pot considerar que estan organitzats un darrere l'altre, seqüencialment, com les paraules en la pàgina d'un llibre.

El següent podria ser el contingut d'un fitxer orientat a caràcter on hi ha deu valors de tipus real, 7 en la primera línia i 3 en la segona:

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

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

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

En un fitxer orientat a caràcter **és possible emmagatzemar qualsevol combinació de dades de qualsevol tipus** (int, double, boolean, String, etc.).

```
7 10 20,5 16,99
Hi havia una vegada...
true false 2020 0,1234
```

El principal avantatge d'un fitxer d'aquest tipus és que resulta molt senzill inspeccionar el seu contingut i generar-los d'acord amb les nostres necessitats.

Per al cas dels fitxers orientats a caràcter, cal usar dues classes diferents segons si el que es vol és llegir o escriure dades en un arxiu. Normalment això no és molt problemàtic, ja que **en un bloc de codi donat, només es duran a terme operacions de lectura o d'escriptura sobre un mateix arxiu, però no els dos tipus d'operació alhora**.

Per a tractar de manera senzilla fitxers orientats a caràcter, Java proporciona les classes **Scanner** (per a lectura) del **package java.util**, i **FileWriter** (per a escriptura) del **package java.io**.

3.2. Lectura de fitxer (classe Scanner)

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: Scanner. Al cap i a la fi, els valors emmagatzemats en els arxius d'aquest tipus es troben exactament en el mateix format que ha usat fins ara per a entrar informació en els seus 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 es troben emmagatzemats en un fitxer.

Per a processar dades des d'un arxiu, **el constructor de la classe Scanner permet com a argument un objecte de tipus File** que continga la ruta a un arxiu.

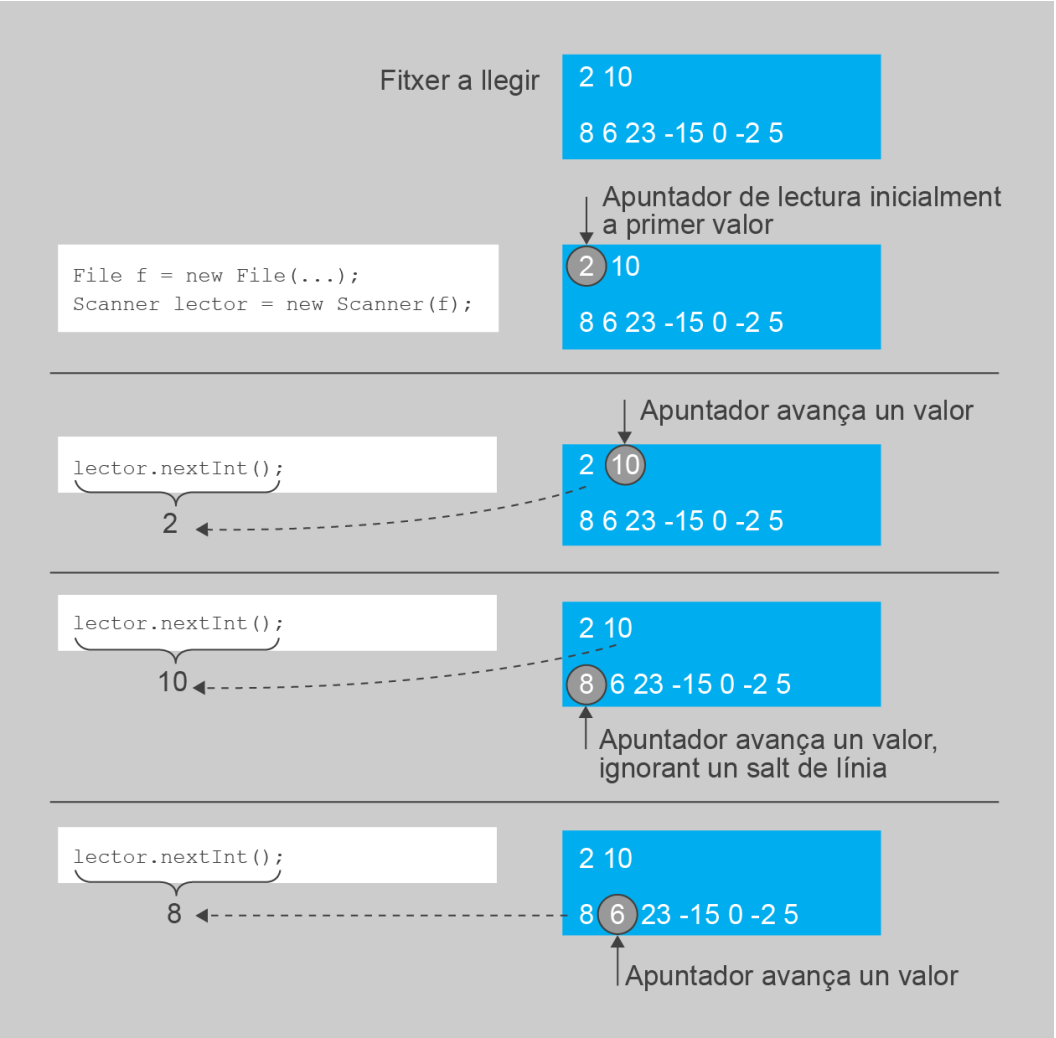
Per exemple, per a crear un objecte de tipus Scanner de manera que permeta llegir dades des de l'arxiu situat en la ruta "C:\Programes\Unitat11\Document.txt", caldria fer:

```
import java.io.File;
import java.util.Scanner;
...
File f = new File("C:\Programes\Unitat11\Document.txt");
Scanner lectorArchivo = new Scanner(f);
...
```

Una vegada instanciat l'objecte Scanner **podem utilitzar els seus mètodes exactament igual que si llegírem de teclat**: hasNext(), next(), nextLine(), nextInt(), nextDouble(), nextBoolean(), etc. L'única diferència és que l'objecte Scanner llegirà seqüencialment el contingut de l'arxiu.

És important entendre que en el cas d'un arxiu, **l'objecte Scanner gestiona internament un apuntador que indica sobre quin valor actuaran les operacions de lectura**. Inicialment l'apuntador es troba en el primer valor dins de l'arxiu. **Cada vegada que es fa una lectura l'apuntador avança automàticament fins al següent valor dins de l'arxiu i no hi ha cap manera de fer-lo retrocedir**. A mesura que invoquem mètodes de lectura l'apuntador continua avançant fins que hàgem llegit tantes dades com vulguem, o fins que no puguem continuar llegint perquè hem arribat al final del fitxer.

A continuació es mostra un xicotet esquema d'aquest procés, recalcant com avança l'apuntador a l'hora de realitzar operacions de lectura sobre un arxiu que conté valors de tipus sencer.



És important recordar la diferència entre el mètode next() i nextLine(), ja que tots dos avaluen una cadena de text. El mètode **next() només llig una paraula individual** (conjunts de caràcters, inclosos dígits, que no estan separats per espais o salts de línia, com per exemple "casa", "hola", "2", "3,14", "1024", etc.). En canvi, **nextLine() llig tot el text que trobe (espais inclosos) fins al següent salt de línia**. En tal cas l'apuntador es posiciona a l'inici de la següent línia.

Una vegada s'ha finalitzat la lectura de l'arxiu, ja siguen totes o només una part, **és imprescindible executar un mètode especial anomenat close()**. Aquest mètode indica al sistema operatiu que l'arxiu ja no està sent utilitzat pel programa. Això és molt important ja que mentre un arxiu es considera en ús, el seu accés pot veure's limitat. Si no s'utilitza *close()* el sistema operatiu pot tardar un temps a adonar-se que l'arxiu ja no està en ús.

> **Sempre cal tancar els arxius amb close()** quan s'ha acabat de llegir o escriure en ells.

És important saber que al instanciar l'objecte Scanner **es llançarà una excepció de tipus java.io.FileNotFoundException si el fitxer no existeix**. Sempre caldrà manejar aquesta excepció mitjançant un try-catch. Scanner també **pot llançar altres excepcions**, per exemple si s'intenta llegir el tipus de dada incorrecta (cridem a nextInt() quan no hi ha un enter, com succeeix en l'entrada per teclat,) o si hem arribat al final del fitxer i intentem continuar llegint (podem comprovar-lo mitjançant el mètode hasNext() de Scanner, que retorna true si encara hi ha algun element que llegir).

El programa següent mostra un exemple de com llegir deu valors sencers d'un arxiu anomenat "Enters.txt" situat en la carpeta de treball (hauria de ser la carpeta del projecte NetBeans). Per a provar-ho, crea l'arxiu i introdueix exactament 10 valors sencers separats per espais en blanc o salts de línia.

```
import java.io.File;
import java.util.Scanner;

public class ProvesFitxers {

     public static final int NUM_VALORS = 10;

     public static void main(String[] args) {

          try {
               // Intentem obrir el fitxer
               File f = new File("Enters.txt");
               Scanner lector = new Scanner(f);

               // Si arriba ací és que ha obert el fitxer :)
               for (int i = 0; i < NUM_VALORS; i++) {
               int valor = lector.nextInt();
               System.out.println("El valor llegit és: " + valor);
                }

                // Cal tancar el fitxer!
                lector.close();

           }
          catch (Exception e) {
                // En cas d'excepció mostrar l'error
                System.out.println("Error: " + e);
                e.printStackTrace();
           }
      }
}
```

Una diferència important a l'hora de tractar amb arxius 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 en l'arxiu i res més. Això té dos efectes sobre el procés de lectura:

1. D'una banda, recorda que **quan es duu a terme el procés de lectura d'una seqüència de valors, sempre cal anar amb compte d'usar el mètode adequat a la mena de valor que s'espera que vinga a continuació**. Quin tipus de valor s'espera és una cosa que haureu decidit vosaltres a l'hora de fer el programa que va escriure aqueix arxiu, per la qual cosa és la vostra responsabilitat saber què cal llegir a cada moment. De totes maneres res garanteix que no s'haja comés algun error o que l'arxiu haja sigut manipulat per un altre programa o usuari. Com operem amb fitxers i no pel teclat, no existeix l'opció de demanar a l'usuari que torne a escriure la dada. Per tant, el programa hauria de dir que s'ha produït un error ja que l'arxiu no té el format correcte i finalitzar el procés de lectura.

2. D'altra banda, **també és necessari controlar que mai es llegisquen més valors dels que hi ha disponibles per a llegir**. En el cas de l'entrada de dades pel teclat el programa simplement es bloquejava i esperava a que l'usuari escriguera nous valors. Però amb fitxers això no succeeix. Intentar llegir un nou valor quan l'apuntador ja ha superat l'últim disponible es considera erroni i llançarà una excepció. Per a evitar-ho, **serà necessari utilitzar el mètode hasNext() abans de llegir, que ens retornarà true si existeix un element a continuació**. Una vegada s'arriba al final de l'arxiu ja no queda més remei que invocar close () i finalitzar la lectura.

Per a veure-ho, intenta executar l'exemple anterior modificant l'arxiu de manera que algun dels valors no siguen de tipus sencer, o hi haja menys de 10 valors.

3.3. Escriptura en fitxer (classe FileWriter)

Per a escriure dades a un arxiu la classe més senzilla d'utilitzar és FileWriter. Aquesta classe té dos constructors que val la pena conéixer:

- `public FileWriter(File file)`
- `public FileWriter(File file, boolean append)`

El primer constructor és molt semblant al del Scanner. Només cal passar-li un objecte File amb la ruta a l'arxiu. En tractar-se d'escriptura la ruta pot indicar un fitxer que pot existir o no dins del sistema. **Si el fitxer no existeix, es crearà un nou**. Però **si el fitxer ja existeix, el seu contingut s'esborra per complet, amb grandària igual a** 0 . Això pot ser perillós ja que si no es maneja correctament pot produir la pèrdua de dades valuoses. Cal estar completament segur que es vol sobreescriure el fitxer.

```
import java.io.File;
import java.io.FileWriter;
...
File f = new File("C:\Programes\Unitat11\Document.txt");
FileWriter writer = new FileWriter(f);
```

El segon constructor té un altre **paràmetre de tipus booleà anomenat "append" (afegir) que ens permet indicar si volem escriure al final del fitxer o no**. És a dir, si li passem "false" farà el mateix que el contructor anterior (si l'arxiu ja existeix, el sobreescriurà), però si li passem "true" obrirà l'arxiu per a escriptura en **mode "append"**, és a dir, **escriurem al final del fitxer sense esborrar les dades ja existents**.

```
import java.io.File;
import java.io.FileWriter;
...
File f = new File("C:\Programes\Unitat11\Document.txt");
FileWriter writer = new FileWriter(f, true);
```

L'escriptura seqüencial de dades en un fitxer orientat a caràcter és molt senzilla. Només és necessari utilitzar el següent mètode **void write(String str)** que escriurà la cadena str en el fitxer. Si es desitja agregar un **final de línia** es pot agregar **"\n"**.

> **Tant el constructor de FileWriter com el mètode write() poden llançar una excepció IOException** si es produeix algun error inesperat.

És important tindre en compte que **perquè el mètode write() escriga text correctament és imprescindible passar-li com a argument un String**. Està permés utilitzar dades o variables diferents a String, però s'escriurà directament el seu valor en bytes, no com a text. Vegem dos exemples il·lustratius.

```
writer.write("8"); // Escriu el caràcter 8
writer.write(8); // Escriu 8 com a byte, és un caràcter no imprimible

writer.write("65"); // Escriu dos caràcters, el 6 i el 5
writer.write(65); // Escriu 65 com a byte, és el caràcter A
```

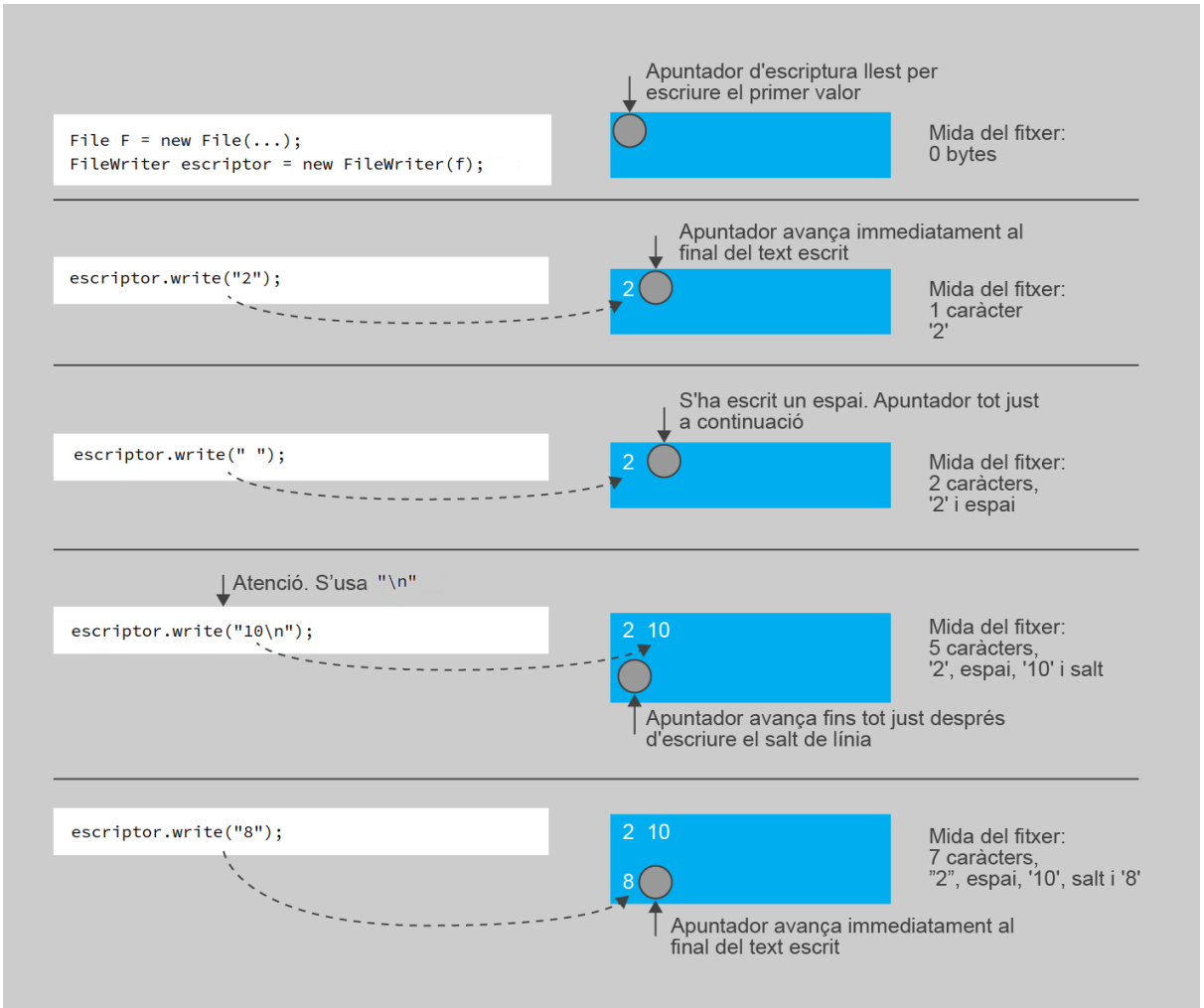
Per tant quan volem escriure el valor de variables que no siguen String serà necessari passar-se-les a write() com String. Això és molt senzill, només cal concatenar un String buit amb la variable (Java sempre converteix a String la concatenació de cadenes de text amb qualsevol altre element): `"" + variable`

```
int edat = 35;
writer.write("" + edat); // escriu el text "35"
```

L'escriptura de dades en fitxer té la particularitat que una vegada s'ha escrit una dada ja no hi ha marxa arrere. No és possible escriure informació abans o enmig de valors que ja estan escrits.

Com en el cas de la lectura, la classe FileWriter també gestiona un apuntador que li permet saber a partir de quina posició del text ha d'anar escrivint. Cada vegada que s'invoca un dels seus mètodes d'escriptura, l'apuntador avança automàticament i no és possible fer-lo retrocedir. A efectes pràctics aquest apuntador sempre està al final de l'arxiu, de manera que a mesura que es van escrivint dades l'arxiu va incrementant la seua grandària.

A continuació es mostra un esquema del funcionament de l'escriptura en fitxer.



L'escriptura no genera automàticament un delimitador entre valors. Els espais en blanc o salts de línia que es desitgen incorporar han d'escriure's esplícitamente. En cas contrari els valors quedaran pegats i en una posterior lectura s'interpretaran com un únic valor. Per exemple, si s'escriu el valor 2 i després el 4, sense espai, en el fitxer s'haurà escrit el valor 24. Si es llegira mitjançant un nextInt() ens retornaria un únic valor, no dos.

En escriure en fitxers el tancament amb close() és encara més important que en la lectura. Això es deu al fet que els sistemes operatius sovint actualitzen les dades de forma diferida. És a dir, el fet d'executar una instrucció d'escriptura no significa que immediatament s'escriga en l'arxiu. Pot passar un interval de temps variable. Només en executar el mètode close() es força al sistema operatiu a escriure les dades pendents (si n'hi haguera).

> En acabar l'escriptura també **és imprescindible invocar el mètode close()** per a tancar-lo i assegurar la correcta escriptura de dades.

El codi següent serveix com a exemple d'un programa que escriu un arxiu anomenat "Enters.txt" dins de la carpeta de treball. S'escriuen 20 valors enters, començant per l'1 i cada vegada el doble de l'anterior. Prova-ho per a veure el seu funcionament. Tingues en compte que si ja existia un arxiu amb aqueix nom, quedarà totalment sobreescrit. Després, pots intentar llegir-ho amb el programa de l'exemple anterior per a llegir 10 valors enters i mostrar-los per pantalla.

```
public static void main(String[] args) {
	try {
		File f = new File("Enters.txt");
		FileWriter fw = new FileWriter(f);

		int valor = 1;

		for (int i = 1; i <= 20; i++) {
		fw.write("" + valor); // escrivim valor
		fw.write(" "); // escrivim espai en blanc
		valor = valor * 2; // calculem pròxim valor
		}

		fw.write("\n"); // escrivim nova línia

		fw.close(); // tanquem el FileWriter

		System.out.println("Fitxer escrit correctament");

	}
	catch (IOException e) {
		System.out.println("Error: " + e);
	}
}
```

Prova d'executar el codi diverses vegades. Veuràs que l'arxiu se sobreescriu i sempre queda igual. Després, modifica la instanciación del FileWriter agregant el segon argument (“append”) a true: `FileWriter fw = new FileWriter(f, true);` Prova-ho i veuràs que ja no se sobreescriu el fitxer, sinó que s'afigen els 20 números al final.

4. BIBLIOGRAFIA

Part del contingut d'aquesta unitat didàctica és un estracte de les anotacions de programació de Joan Arnedo Moreno (Institut Obert de Catalunya, IOC).

També s'ha utilitzat com a referència les següents fonts:

[1] Anotacions de programació de Jose Luis Comesaña (sitiolibre.com).

[2] Anotacions de programació de Natividad Prieto, Francisco Marqués i Javier Piris (E.T.S. d'Informàtica, Universitat Politècnica de València).