Interfícies gràfiques d'usuari. Fluxos i fitxers

Joan Arnedo Moreno

Índex

Introducció

Les classes accessibles a través de l'API del Java van més enllà de tasques genèriques simples i també ofereixen funcionalitats més complexes, però igualment desitjables dins de qualsevol aplicació moderna. Per tant, aprofundir en l'estudi de l'API és un aspecte molt important per a aquells que vulgueu desenvolupar aplicacions en Java de manera habitual. Atesa la seva envergadura, és molt complicat d'explicar fins el seu darrer detall, amb la qual cosa aquesta unitat se centra en el conjunt de classes dins seu que es consideren més útils: aquell vinculat a l'ús dels mecanismes d'entrada/sortida. A fi de comptes, quina utilitat té ser capaç de fer tasques complexes sobre grans quantitats de dades si el programa és incapaç d'obtenir-les d'algun lloc i després poder desar el resultat o mostrar-lo a l'usuari? Si bé l'entrada de dades per consola, o sigui, entrada via teclat i sortida en línies de text per pantalla, és un sistema pel qual és fàcil d'assolir aquesta fita, difícilment es considera un mecanisme satisfactori en la gran majoria d'aplicacions modernes. Tot aquest conjunt de classes es pot dividir en dues parts d'acord amb com s'organitza dins de l'API del Java. D'una banda, les classes associades a la creació d'interfícies gràfiques d'usuari. Si bé actualment ja no es tracta d'un aspecte revolucionari, sí que van més enllà de l'aprenentatge de les funcions bàsiques del llenguatge Java. D'altra banda, les classes associades al mecanisme genèric utilitzat per Java de cara a gestionar l'entrada/sortida de quantitats indeterminades de dades, els fluxos.

A l'apartat "Interfícies gràfiques d'usuari" es presenta a grans trets la llibreria gràfica de Java, anomenada Swing. Aquesta llibreria permet generar entorns gràfics amigables basats en finestres, amb els quals es permet que l'usuari interactuï de manera intuïtiva amb l'aplicació dissenyada. Mitjançant les seves classes, és possible presentar de manera relativament simple elements gràfics en pantalla i transformar les interaccions de l'usuari dutes a terme amb ratolí o teclat en invocacions a mètodes de les classes fruit del procés de disseny. A part, també veureu com treballar amb altres elements gràfics que es distancien del model general de programació d'interfícies gràfiques, amb els quals fer programes és una mica més complex, com fer figures lliures.

A l'apartat "Fluxos i fitxers" s'expliquen les llibreries d'entrada/sortida usades pel Java per gestionar fluxos. Els fluxos són un sistema àmpliament usat en molts llenguatges de programació per gestionar les dades d'una aplicació quan es volen processar d'alguna manera, com pot ser el cas d'assolir la seva persistència, de manera que es puguin desenvolupar programes amb un codi que és pràcticament sempre el mateix sigui quin sigui l'origen o la destinació de les dades: fitxers, *buffers* a memòria, comunicacions en xarxa... A part d'aquest sistema, també es farà un breu incís en algunes operacions útils de cara a emmagatzemar dades dins fitxers.

Un aspecte molt important al llarg de tota la unitat és ser conscient de la impossibilitat de descriure fins a la darrera funcionalitat de totes les classes implicades tant en una interfície gràfica com en la gestió de l'entrada/sortida mitjançant fluxos. Les llibreries de Java són molt extenses i complexes. Per tant, és inevitable haver d'acudir a la documentació oficial per poder estudiar amb detall quins mètodes ofereix cada classe i per a què serveixen.

Resultats d'aprenentatge

En finalitzar aquesta unitat l'alumne/a:

1. Desenvolupa interfícies gràfiques d'usuari simples, utilitzant les llibreries de classes adequades.

- Utilitza les eines de l'entorn de desenvolupament per crear interfícies gràfiques d'usuari simples.
- Programa controladors d'esdeveniments.
- Escriu programes que utilitzin interfícies gràfiques per a l'entrada i sortida d'informació.

2. Realitza operacions bàsiques d'entrada/sortida de informació, sobre consola i fitxers, utilitzant les llibreries de classes adequades.

- Utilitza la consola per realitzar operacions d'entrada i sortida d'informació.
- Aplica formats en la visualització de la informació.
- Reconeix les possibilitats d'entrada / sortida del llenguatge i les llibreries associades.
- Utilitza fitxers per emmagatzemar i recuperar informació.
- Crea programes que utilitzen diversos mètodes d'accés al contingut dels fitxers.

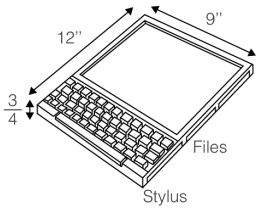
1. Interfícies gràfiques d'usuari

A mesura que els ordinadors i els sistemes operatius han evolucionat, una gran part de les aplicacions desenvolupades han passat d'estar orientades a la línia d'ordres, mostrant la informació en format exclusivament textual, a estar basades en un entorn totalment gràfic, molt més amigable per a l'usuari. Actualment, poques aplicacions no es basen en el que col·loquialment s'anomena una GUI (*graphical user interface*, *interface* gràfica d'usuari).

La immensa majoria de sistemes operatius moderns es basen principalment en una GUI per interactuar amb l'usuari.

Una **GUI** és un tipus d'*interface* en què els mecanismes que utilitza l'usuari per donar ordres al programa o visualitzar qualsevol informació es basa en la manipulació d'icones en lloc de l'entrada d'ordres textuals.

Cal remuntar-se fins als anys seixanta per trobar les primeres petjades de les GUI, basades en el treball dels investigadors de l'empresa Xerox. Ja al final dels anys seixanta i al principi dels setanta, Alan Kay, investigador de la Universitat de Utah, va considerar que la metodologia de l'orientació a objectes era especialment indicada per al disseny d'entorns gràfics. Una GUI és un exemple evident d'elements clarament identificables, amb unes propietats i un comportament, que interactuen per dur a terme una tasca concreta. Les seves idees van ser aprofitades pels investigadors de Xerox per crear un ordinador amb entorn totalment gràfic, el Dynabook. Conjuntament, aquesta feina també va desembocar en un fet important com el naixement de l'SmallTalk, un dels primers llenguatges orientats a objectes. Així, doncs, ja es pot veure que l'orientació a objectes i la generació de GUI són aspectes molt íntimament lligats.



Imatge del Dynabook esbossada per Alan Kay

Avançant fins als anys noranta, sorgeix un nou llenguatge que també aprofita els postulats d'Alan Kay: el llenguatge Java. En els seus orígens, aquest llenguatge estava especialment orientat a poder incloure en pàgines web elements gràfics dinàmics, en forma de petits programes que s'executaven en el navegador: els *applets*. No és casual que el desenvolupador de la primera versió de la biblioteca gràfica fos Netscape Communications, la companyia responsable del navegador principal del moment. Amb les millores successives i l'augment de la potència dels ordinadors, les aplicacions desenvolupades en Java finalment van fer el salt des del navegador a l'escriptori. Per aquest motiu, un dels punts en què el Java ofereix una biblioteca més completa i amb força feina al darrere és per a la generació d'entorns gràfics.

Apple i les GUI

Un mite molt popular és que l'empresa Apple va ser la primera a desenvolupar GUI. En realitat, es va inspirar en les innovacions anteriors de Xerox.

Aquest nucli d'activitat se centra totalment en la generació d'entorns gràfics mitjançant el Java. Per assolir aquesta fita, no solament és necessari entendre quines són les classes relacionades amb elements gràfics, sinó que també cal entendre com es vinculen a les classes especificades en l'etapa de disseny: la lògica interna del programa.

1.1 El paquet Java Swing

El conjunt de classes vinculades a l'entorn gràfic del Java pertanyen a la jerarquia de paquets javax.swing. Familiarment, es coneixen com la biblioteca Java Swing o, simplement, Swing. Aquestes són una extensió d'una biblioteca més antiga anomenada AWT (*Abstract Windows Toolkit*, joc d'eines abstracte de finestres).

Naixement del Java

Cal dir que quan sorgeix el Java, els navegadors més populars avui dia encara estan fent els primers passos. De fet, Internet Explorer no apareix fins al 1995.

La biblioteca original AWT sorgeix l'any 1995 com una primera aproximació a un mecanisme de generació d'entorns gràfics que sigui totalment abstracte, amb un estil homogeni independentment de l'arquitectura sobre la qual s'executi l'aplicació. Aquesta era una tasca molt complicada, ja que cada sistema operatiu disposa del seu propi aspecte (*look and feel*) i primitives per a la gestió d'elements gràfics, normalment molt diferents entre si. En darrera instància, es pot considerar que AWT va assolir la seva fita: mitjançant el seu ús és possible generar *interfaces* molt lletges de manera homogènia, independentment de l'arquitectura. Aquest resultat pot ser comprensible si es té en compte que van ser desenvolupades a corre cuita en un sol mes.

Afortunadament, tot i aquest mal pas, molts dels principis ideats per AWT, fora de l'àmbit purament estètic, van servir per generar una nova versió millorada visualment: **la biblioteca Swing**. Alguns exemples de les solucions que va aportar AWT, i van ser reusats per Swing, són quina estratègia cal usar per vincular la lògica interna del programa a la *interface* gràfica, o com es poden organitzar els elements gràfics dins una finestra. A l'actualitat, pràcticament cap aplicació usa AWT com a biblioteca gràfica, sempre s'utilitza la biblioteca Swing. Tot i així, AWT sempre està present en el rerefons de qualsevol aplicació basada en Swing.

1.1.1 Jerarquia de classes Swing

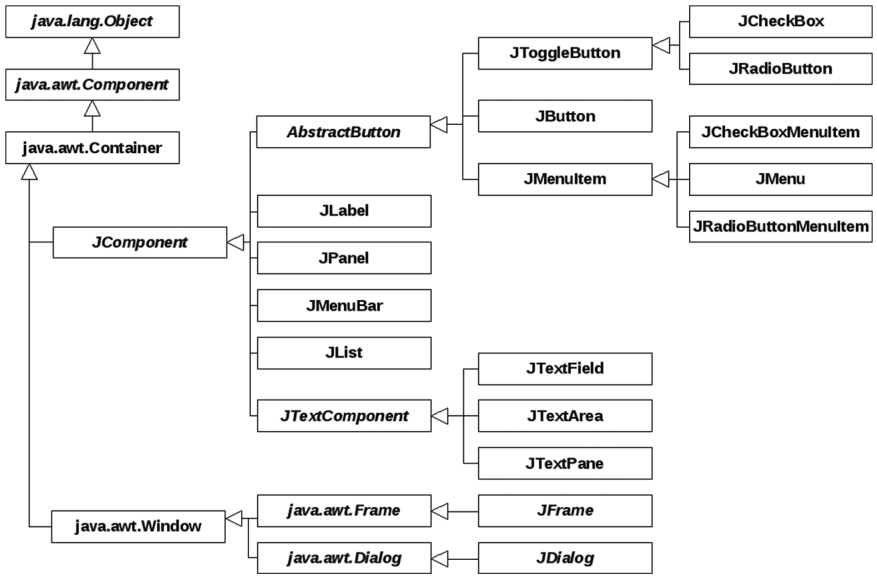
Una jerarquia de classes és la manera com es classifiquen diferents classes amb relacions d'herència entre elles.

La biblioteca Swing pren la forma d'una jerarquia de classes de mida considerable. Cadascuna de les classes que en formen part representa un element típic d'un entorn gràfic: finestres, botons, formularis, menús, etc. Si es vol incloure algun d'aquests elements en la *interface* de l'aplicació, caldrà instanciar la classe pertinent.

> Tot i cada element que compon un entorn gràfic és un **objecte**. Hi haurà tants objectes com elements es vulguin incloure.

Una part petita però prou aclaridora de la jerarquia de classes Swing es presenta en la figura 1.1. Per millorar la llegibilitat de l'esquema, s'ha simplificat el format de les classes.

FIGURA 1.1. Jerarquia de les classes Swing



Qualsevol element gràfic dins la *interface* gràfica s'anomena un **component**, ja que tots són un objecte java.awt.Component.

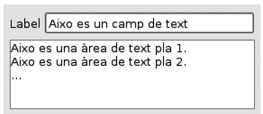
Les classes amb noms que comencen per *J* en la figura són les incloses a Swing, mentre que la resta pertanyen a AWT o a la biblioteca estàndard del Java. Com es pot veure, una part de la biblioteca original AWT continua present dins Swing en forma de les superclasses `java.awt.Component` i `java.awt.Container` (entre d'altres), ja que Swing és una extensió d'aquesta. Aquestes dues classes especifiquen tots els aspectes genèrics del comportament dels elements d'un entorn gràfic. Per fer-ho, defineixen mètodes que les subclasses poden sobreescriue d'acord amb les seves particularitats. Això permet al motor gràfic del Java garantir que cada component sempre té implementat tot el conjunt de mètodes necessaris per visualitzar-los correctament i cridar-los polimòrficament.

Per tenir la llista completa de les classes i tots els seus mètodes cal consultar la documentació del Java.

Específicament, la classe abstracta `java.awt.Component` defineix totes les característiques referents a l'aspecte d'un element gràfic, com la mida, la font del text que conté, si està habilitat o la ubicació en pantalla, com també tot el conjunt d'interaccions que l'element pot rebre (ser pitjat, seleccionat, posar el punter del ratolí a sobre, etc.). En canvi, la classe `java.awt.Container` especifica tot el comportament relatiu a la capacitat d'un element gràfic de contenir-ne d'altres.

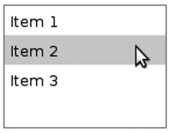
JCheckBox i JRadioButton

En la jerarquia completa de Swing hi ha una gran quantitat de components en forma de subclasses de JComponent, moltes més de les representades en la figura 1.1. Les classes més significatives, tot i que la llista no és ni molt menys completa, són les següents:



Exemple de JLabel, JTextField i JTextArea

- **JButton:** Correspon al botó típic que es pot pitjar per donar ordres a l'aplicació.
- **JToggleButton:** Es tracta d'un botó especial amb ressort, que alterna entre un estat de pitjat o no. Cada cop que es pitja canvia d'estat.



Exemple de JList

- **JCheckBox** i **JRadioButton:** Representen un selector d'opció amb un text associat. En el primer cas, de forma quadrada tipus *checked/ unchecked*(marcat/no marcat), i en el segon de forma rodona. En el fons, és un cas especial d'un botó amb ressort, però amb una representació gràfica diferent. Són útils per fer apartats de configuració o formularis tipus test. La figura 1.3 mostra un exemple de visualització d'un JButton amb icones gràfiques, un JCheckBox i un JRadioButton (de dalt a baix). Noteu la diferència en l'aspecte dels dos darrers.

- **JLabel:** Correspon a una etiqueta en què es pot mostrar text o una imatge.

- **JTextField:** Correspon a un camp de text, en què l'usuari pot escriure. Només es pot escriure una única línia. No permet variar l'estil dins del text (mida o tipus de la font).

- **JTextArea:** Component similar a l'anterior, per en aquest cas especifica una àrea de text en què és possible escriure lliurement, sense limitació a una única línia. La figura 1.4 mostra un exemple de JLabel (dalt a l'esquerra), JTextField (dalt a la dreta) i JTextArea (a baix). Mentre que en la primera no es pot escriure, a la resta si que es pot.

- **JTextPane:** Component de funcionalitat pràcticament idèntica a l'àrea de text, però amb la capacitat afegida de poder editar text amb estil diferent (cursiva, negreta, etc.).

- **JList:** Una llista d'elements o ítems, normalment cadenes de text, d'entre les quals es pot seleccionar un conjunt. La figura 1.5 mostra un exemple d'una JList, una llista d'elements amb opció de selecció múltiple. En aquest cas, es troba seleccionat l'element anomenat "item 2".

- **JFrame:** La finestra principal de la *interface* gràfica en una aplicació d'escriptori.

- **JApplet:** La finestra principal de la *interface* gràfica en una aplicació incrustada en una pàgina web, un *applet*.

- **JDialog:** És un quadre d'alerta o de diàleg amb l'usuari.

- **JPanel:** Representa una àrea específica de la finestra, amb unes propietats concretes diferenciades: color de fons, vora, etc.

- **JScrollPane:** Idèntic a l'anterior, però si en algun moment la mida de la finestra és massa petita per visualitzar tot el contingut d'aquesta àrea, apareix una barra de desplaçament (*scroll*).

- **JTabbedPane:** Un conjunt de panells accessible mitjançant etiquetes (*tabs*) amb una cadena de text, de manera semblant a fitxes de biblioteca. Cada panell està associat a una etiqueta, de manera que quan es pitja, només es visualitza aquest panell.

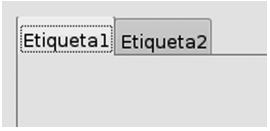
1.1.2 Agregació de components

Tot i que no es mostra representat explícitament dins la jerarquia de classes de Swing, ja que totes les seves classes hereten tant de `java.awt.Component` com de `java.awt.Container`, es considera que hi ha dos tipus de components. Per una banda, els **controls Swing**, que són aquells components amb els quals l'usuari interactua directament: botons, opcions de menú, quadres de text, etc.

Exemples de controls Swing poden ser JButton, JToggleButton, JCheckBox, JRadioButton, JLabel, JList, JTextField, JTextArea, JTextPane.

D'altra banda, els **contenidors Swing**, que són aquells components que no tenen una funció directa d'interacció amb l'usuari, que serveixen exclusivament per encabir i organitzar a dintre qualsevol component, tant controls com altres contenidors. Aquests normalment corresponen a components com finestres, panells o barres de menú. Si bé és possible interactuar-hi (per exemple, redimensionar una finestra o obrir un menú), els resultats de l'acció solen quedar dins l'àmbit de l'entorn gràfic, i no se solen usar per donar ordres a la lògica interna del programa.

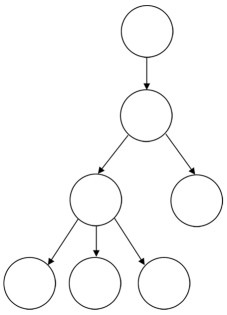
Exemples de contenidors poden ser JFrame, JDialog, JApplet, JPanel, JScrollPane, JTabbedPane.



Exemple de JTabbedPane

La figura 1.6 mostra un exemple de JTabbedPane, un contenidor. Aquest és capaç d'encabir diferents elements gràfics, ordenats segons diferents etiquetes. En aquest cas, hi ha dues etiquetes anomenades "Etiqueta 1" i "Etiqueta 2" la visualització de les quals es pot anar alternant mitjançant la selecció del nom. En la imatge es visualitza "Etiqueta 1".

La relació entre aquests dos tipus de components dins de qualsevol *interface* gràfica és la següent. Dins un contenidor poden haver tant components com d'altres contenidors. En canvi, un component no pot contindre res, és un element final a l'estructura de la interfície Swing. Per tant, aquesta estructura sempre pren la forma d'un arbre *N*-ari. Els objectes ubicats en les fulles corresponen normalment a controls Swing, mentre que la resta són contenidors. Quan un contenidor A conté un altre component qualsevol B, es diu que A és el **contenidor pare** de B.


Un arbre N-ari és un arbre en què cada node pare pot tenir qualsevol nombre de successors. Els nodes sense successors s'anomenen "fulles".

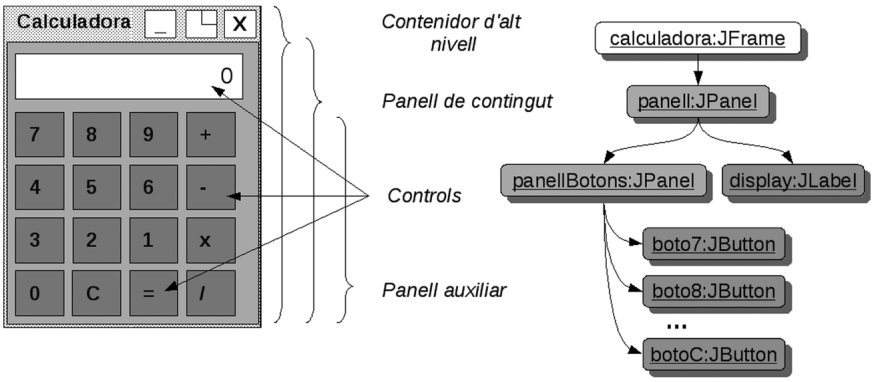
> Per incloure qualsevol **component** dins un contenidor, cal cridar el mètode, `add(Component comp)` sobre el contenidor, passant com a paràmetre el component a afegir-hi. Aquest es troba definit en la classe `java.awt.Container` i totes les classes de la biblioteca Swing l'hereten.

Addicionalment, hi ha un subtipus especial de contenidors Swing, els anomenats **contenidors d'alt nivell** (*top-level containers*). Aquests es consideren els contenidors principals de la *interface* d'usuari, de manera que l'objecte arrel de l'arbre que conforma el mapa d'objectes de la interfície gràfica sempre és un contenidor d'aquest subtipus. Swing en defineix tres, tots subclasses de java.awt.Window: JFrame, JApplet i JDialog.

En el cas dels contenidors d'alt nivell, no és possible afegir-hi directament components cridant el mètode `add`. Només és possible la interacció mitjançant el seu **panell de contingut** (*content pane*), que es pot obtenir amb el mètode `getContentPane()`. Aquest ja és un contenidor normal sobre el qual sí que es pot cridar el mètode `add`.

La figura 1.2 mostra un exemple de com es representaria una *interface* gràfica concreta en forma de mapa d'objectes i la seva classificació per tipus.

FIGURA 1.2. Estructura d'una calculadora simple



El mètode setHorizontalAlignment assigna el tipus de justificació del text. La classe defineix un conjunt de constants estàtiques per cada tipus de justificació.

Cal instanciar les diferents classes associades a cada component del panell de contingut i afegir cada control obtingut en els contenidors corresponents, de manera que al final tots estiguin vinculats a un contenidor d'alt nivell: la finestra principal, instància de JFrame. Un fragment prou significatiu del codi necessari per generar el panell de contingut és el següent:

```
1  //Contenidor d'alt nivell: finestra principal
2  JFrame calculadora = new JFrame("Calculadora");
3  //Panell de contingut
4  Container panell = calculadora.getContentPane();
5  ...
6  //Display de la calculadora
7  JLabel display = new JLabel();
8  display.setHorizontalAlignment(SwingConstants.RIGHT);
9  panell.add(display);
10 ...
11 //Panell auxiliar on posar els botons
12 JPanel panellBotons = new JPanel();
13 JButton boto7 = new JButton("7");
14 JButton boto8 = new JButton("8");
15 ...
16 JButton botoC = new JButton("C");
17 //Afegir botons a panell auxiliar
18 panellBotons.add(boto7);
19 panellBotons.add(boto8);
20 ...
21 panellBotons.add(botoC);
22 //Afegir panell auxiliar de botons a interface
23 panell.add(panellBotons);
24 calculadora.setVisible();
```

El mètode setVisible fa visible el component. Fins que no es crida, tot i estar creat, és invisible per a l'usuari.

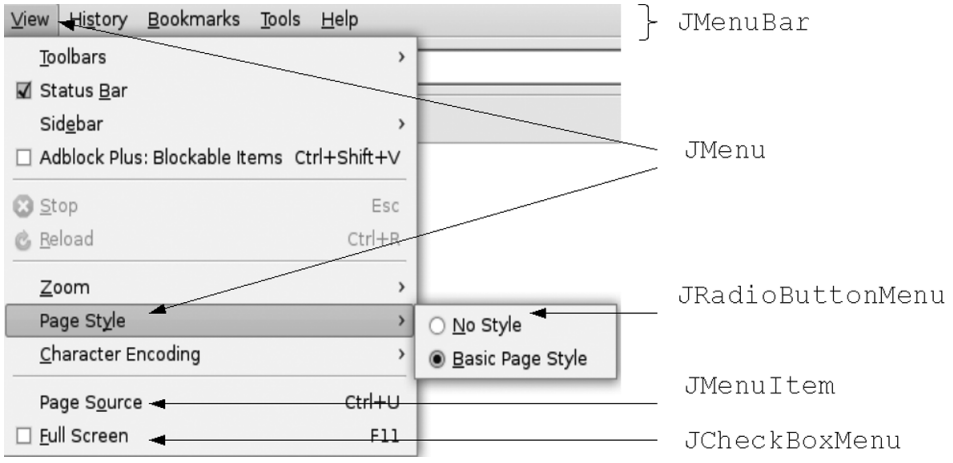
1.1.3 La barra de menús

Un tret característic força típic de les aplicacions d'escriptori és la utilització de menús en la franja superior de la finestra principal, amb l'objectiu de donar accés a moltes opcions sense atapeir la pantalla. Els components principals vinculats als menús dins la biblioteca Swing són els següents:

- **JMenuBar:** Contenidor que representa la barra de menús. Només n'hi pot haver un per finestra (JFrame).
- **JMenu:** Contenidor que representa un menú individual d'entre els diferents que es poden incloure dins la barra de menús. L'usuari visualitza el seu nom i es desplega en pitjar amb el ratolí. Es poden incloure menús dins d'altres menús, que es despleguen consecutivament.
- **JMenuItem:** Control associat a una opció individual de menú, que l'usuari selecciona.
- **JCheckBoxMenuItem:** Control que combina el JMenuItem i elJCheck-Box.
- **JRadioButtonMenuItem:** Control que combina el JMenuItem i elJRadio-ButtonMenu.

La figura 1.3 mostra un exemple amb tots els components possibles d'una barra de menús. No es pot afegir cap altre tipus de component o contenidor dins un menú.

Figura 1.3. Barra de menús amb tots els elements possibles.



Els menús també usen el mètode `add` per afegir components als contenidors. Es visualitzaran dins el menú exactament en el mateix ordre en què s'han afegit. L'única excepció sobre el mecanisme general és assignar l'única barra de menús a la finestra principal, que es fa cridant el mètode següent:

```
1 public void setJMenuBar(JMenuBar menubar)
```

A continuació es presenta un fragment del codi que generaria una barra de menús com la que s'ha mostrat en la figura 1.3, centrant-se en el menú *View*. Analitzeu detingudament com s'afegeixen els elements del menú mitjançant crides successives del mètode add.

El mètode addSeparator permet afegir línies de separació entre els elements d'un menú.

El mètode setSelected serveix per marcar com a seleccionat o deseleccionat un control tipus o .

El mètode setMnemonic serveix per establir una drecera de teclat per a un control donat. La classe defineix constants per a totes les tecles.

El mètode setEnabled habilita o deshabilita un control.

```
1  JFrame navegador = new JFrame();
2  JMenuBar barraMenu = new JMenuBar();
3  JMenu viewMenu = new JMenu("View");
4  JMenu toolbarSubmenu = new JMenu("Toolbars");
5  ...
6  viewMenu.add(toolbarSubmenu);
7  JCkeckBoxMenuItem statusbarItem =
8  new JCkeckBoxMenuItem("Status Bar");
9  statusBar.setSelected();
10 viewMenu.add(statusbarItem);
11 ...
12 viewMenu.addSeparator();
13 JMenuItem stopItem =
14 new JMenuItem("Stop", new ImageIcon("Stop.gif"));
15 stopItem.setMnemonic(KeyEvent.VK_ESCAPE);
16 stopItem.setEnabled(false);
17 viewMenu.add(stopItem);
18 ...
19 JMenu pagestyleSubmenu = new JMenu("Page Style");
20 pagestyleSubmenu.add (new
21 JMenuCheckBoxMenuItem("No Style"));
22 JMenuCheckBoxMenuItem basicStyle = new
23 JMenuCheckBoxMenuItem("Basic Page Style");
24 basicStyle.setSelected();
25 pagestyleSubmenu.add (basicStyle)
26 viewMenu.add(pagestyleSubmenu);
27 ...
28 barraMenu.add(viewMenu);
29 ...
30 navegador.setJMenubar(barraMenu);
```

1.1.4 Layouts

Un dels objectius més importants de la biblioteca d'entorn gràfic del Java és poder generar aplicacions amb un comportament homogeni independentment de la plataforma en què s'executin. Aquesta fita té molt sentit si es recorda que el Java es va pensar inicialment per al desenvolupament d'aplicacions executades en un navegador, els *applets*. En un entorn heterogeni com és Internet, és impossible establir per endavant els paràmetres sota els quals s'executa el navegador: maquinari, sistema operatiu, resolució de la pantalla, dimensions de la finestra del navegador, etc. Per tant, no es pot generar un entorn gràfic en què els components s'ubiquin d'una manera preestablerta i tinguin una mida fixa, suposant unes dimensions concretes de la finestra principal.

El mètode usat per afegir controls a un contenidor, `add (Component comp)`, i les seves sobrecàrregues no disposen de cap paràmetre vinculat a les coordenades en què es pugui ubicar el component, només s'indica el component que s'ha d'afegir. El motiu és el mecanisme que el Java aporta per solucionar la circumstància que és impossible establir per endavant els paràmetres sota els quals s'executa el navegador. En una aplicació Swing (o AWT), en realitat, no és possible establir la

ubicació i mides exactes de cada component dins la *interface* gràfica. El que es fa és, per a cada contenidor Swing (principalment, els JPanel i JFrame), especificar una política d'ubicació de components, de manera que el motor gràfic del Java escull automàticament la millor opció d'acord amb les dimensions reals de la finestra principal. Cada cop que la finestra principal canvia de dimensions, els components es reubiquen i redimensionen dinàmicament. Aquestes polítiques no les ha de generar el desenvolupador -el Swing ja defineix un conjunt predeterminat disponible-, entre les quals tan sols cal triar-ne una per a cada contenidor en la *interface*. Cada una és el que s'anomena un *layout*.

Un **layout** és una política d'ubicació i dimensionament de components, de manera que el motor gràfic del Java escull automàticament on s'ha de visualitzar i quina ha de ser la seva mida.

Concretament, els *layouts* disponibles a Swing són totes les classes que implementen la *interface* `java.awt.LayoutManager`.

El **mètode** que assigna un *layout* a un contenidor és `setLayout (LayoutManager manager)`.

setPreferredSize

Aquest mètode permet suggerir quina mida es vol que tingui un component. Tot i així, no hi ha cap garantia que el layout l'obeeixi sempre.

Sota el sistema de *layouts*, tots els components tenen com a mida per defecte la mínima necessària per encabir tot el contingut. En principi, les seves dimensions no es poden establir de manera estàtica.

Malauradament, els *layouts* són un de tants casos d'idees que sonen molt més bé del que realment funcionen a l'hora de la veritat. En la immensa majoria dels casos, el motor gràfic del Java mai tindrà el mateix concepte de "millor ubicació i mida" que la que té al cap el desenvolupador. En conseqüència, l'ús de *layouts* sense l'ajut d'un IDE que proporcioni un editor d'interfícies gràfiques és una tasca complicada i que normalment necessita la inversió de força temps i esforç. Tot i així, si es vol veure el costat positiu de tot plegat, alguns dels defensors incondicionals del Java consideren que això no és necessàriament dolent, ja que força el desenvolupador a cenyir-se a entorns gràfics senzills, no gaire recarregats i, per tant, més usables. Fer una *interfaces* massa complexa es pot arribar a convertir en un exercici de paciència si no s'és un desenvolupador experimentat en l'ús de *layouts*.

L'única excepció en aquesta característica són els contenidors vinculats a menús, que no usen *layouts*, ja que un menú mai no es redimensiona ni els seus components canvien d'ubicació. Sempre té el mateix aspecte al llarg de l'execució de l'aplicació.

Els *layouts* més significatius són FlowLayout, BorderLayout, GridLayout, BoxLayout, GridBayLayout i GroupLayout.

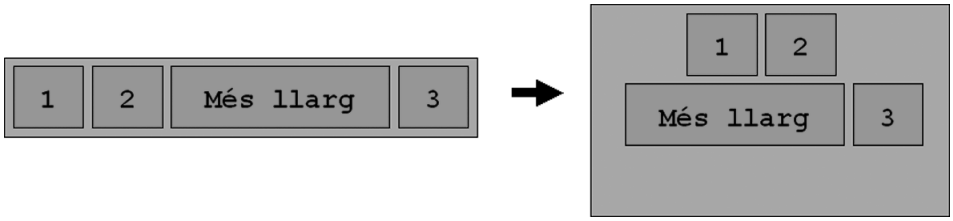
FlowLayout

El FlowLayout és el *layout* que hi ha per defecte en tots els contenidors si no se n'assigna cap altre mitjançant el mètode setLayout. S'anomena així perquè es considera que els elements "flueixen de manera natural" dins el contenidor. La política que defineix és que tots els components es mantenen en la seva mida per defecte i es van ubicant per línies, d'esquerra a dreta i de dalt a baix, centrats horitzontalment. Si en un moment donat un component no cap en la línia actual, s'ubica en la línia immediatament inferior. L'ordre en què s'afegeixen al contenidor és el mateix en què s'ha cridat el mètode add.

El seu constructor és `public FlowLayout()`.

La figura 1.4 mostra un exemple de com varia la ubicació dels components en cas de redimensionar el contenidor. En tots els casos, es considera que la seva mida és la mínima per encabir-ne el contingut (en aquest cas, el mateix text que apareix representat).

FIGURA 1.4. Redimensionat d'un FlowLayout



BorderLayout

En un component en què s'aplica el BorderLayout es defineixen cinc zones diferenciades: nord, sud, est, oest i centre, que corresponen als punts cardinals del contenidor. En cada zona només es pot ubicar un component, que l'ocupa totalment i mai no varia de posició. Els components de les zones nord i sud ocupen el màxim espai possible horitzontal i el mínim indispensable en vertical. Per a les zones est i oest es dóna el cas invers. La zona central és l'única que varia de mida quan es redimensiona el contenidor.

El seu constructor és `public BorderLayout()`.

Aquest *layout* és una excepció respecte a la resta, ja que sobre un contenidor que l'usa es pot cridar una sobrecàrrega del mètode add en què es passa un paràmetre addicional que indica la zona en què es pot ubicar el component:

```
1 public void add(Component comp, Object constraints)
```

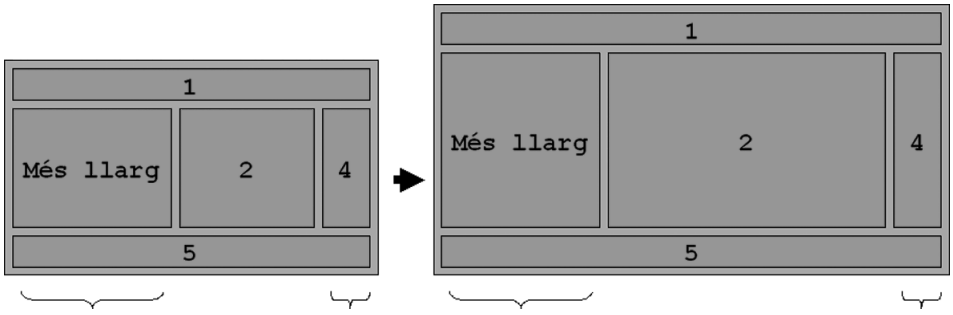
La classe BorderLayout defineix cinc constants que serveixen per indicar cada zona en el paràmetre `constraints`:

- `BorderLayout.NORTH`
- `BorderLayout.SOUTH`

- `BorderLayout.EAST`
- `BorderLayout.WEST`
- `BorderLayout.CENTER`

La figura 1.5 mostra un exemple de redimensionat d'un contenidor amb aquest *layout*. Fixeu-vos que els components a est i oest no varien de mida tot i que el contenidor augmenta d'amplada.

Figura 1.5. Redimensionat d'un BorderLayout



GridLayout

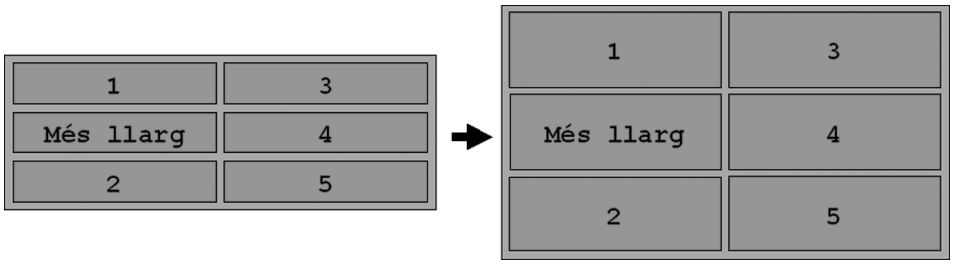
El GridLayout organitza el component com una matriu amb cel·les de mida idèntica. En cada cel·la només hi pot haver un component, que ocupa tot l'espai disponible independentment de quina en seria la mida per defecte. Els components s'ubiquen per files, d'esquerra a dreta en el ma-add teix ordre en què es crida el mètode add.

No es poden saltar cel·les en anar cridant add. Si es vol deixar una cel·la en blanc, n'hi ha prou d'afegir un panell buit.

El seu constructor és `public GridLayout(int rows, int cols)`.

El paràmetre `rows` indica el nombre de files i `cols` el nombre de columnes. La figura 1.6 mostra un exemple de redimensionat d'un GridLayout.

Figura 1.6. Redimensionat d'un GridLayout



BoxLayout

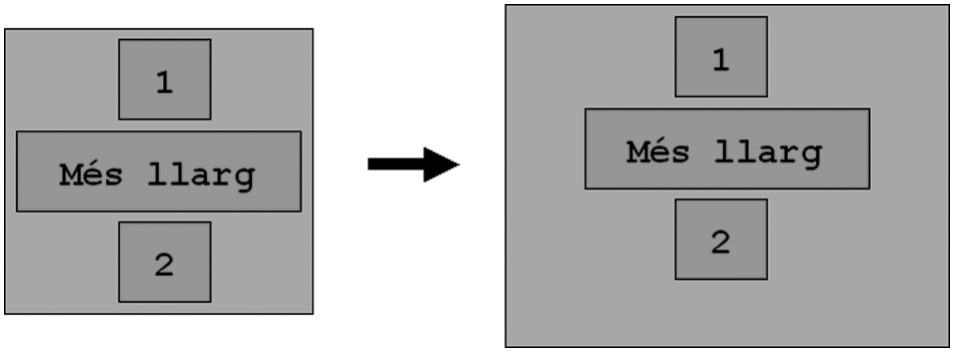
El BoxLayout s'acostuma a aplicar sobre un contenidor específic, el Box, que ja porta incorporat aquest *layout* per defecte en ser instanciat i no es pot canviar. Els components que conté mantenen la mida per defecte i s'ubiquen en una sola línia horitzontal o vertical, centrada.

Les instàncies de la classe Box no es generen mitjançant un mètode constructor, en els seu lloc s'utilitza un dels dos mètodes estàtics definits en la mateixa classe Box. El mètode a usar depèn de si es volen alinear els components horitzontalment o verticalment:

```
1 Box alineaVertical = Box.createVerticalBox();
2 Box alineaHoritzontal = Box.createHorizontalBox();
```

La figura 1.7 mostra un exemple de redimensionat d'un contenidor Box d'alineació vertical.

Figura 1.7. Redimensionat d'un BoxLayout vertical



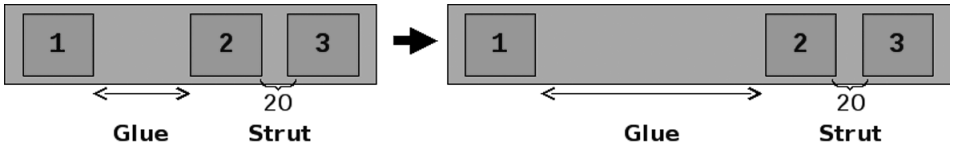
Una altra de les particularitats dels contenidors Box és la seva capacitat d'incloure, usant el mètode add, dos components especials que cap altre tipus de *layout* pot usar: les *Struts* i les *Glues*, verticals i horitzontals. Donada una instància de Box, només es poden afegir *Glues* o *Struts* de la seva mateixa alineació (vertical o horitzontal). Novament, aquests components s'instancien mitjançant mètodes estàtics definits en la classe Box:

- `Component Box.createVerticalGlue()`
- `Component Box.createVerticalStrut(int height)`
- `Component Box.createHorizontalGlue()`
- `Component Box.createHorizontalStrut(int width)`

Les *Struts* són espais en blanc d'un nombre concret de píxels (indicat amb els paràmetres `height` o `width`). Independentment de les dimensions de la Box, aquest espai es manté sempre invariable. Les *Glue* són exactament el contrari al que podria donar a entendre la seva traducció, "cola". Dos components separats per una *Glue* sempre s'ubiquen el més separat possible segons l'espai que hi ha. Es pot considerar que es comporta com una molla en expansió.

En la figura 1.8 es mostra un exemple d'aplicació de *Glues* i *Struts*.

Figura 1.8. Exemple d'ús de "Struts" i "Glues".



CardLayout

Aquest *layout* organitza els components com una pila de cartes, on tots estan ubicats però a cada moment només se'n pot visualitzar un, el qual ocupa el màxim espai possible. L'ordre amb què s'afegeixen a la pila és el mateix en què es crida el mètode add sobre el contenidor.

El CardLayout se sol usar per sobreposar panells.

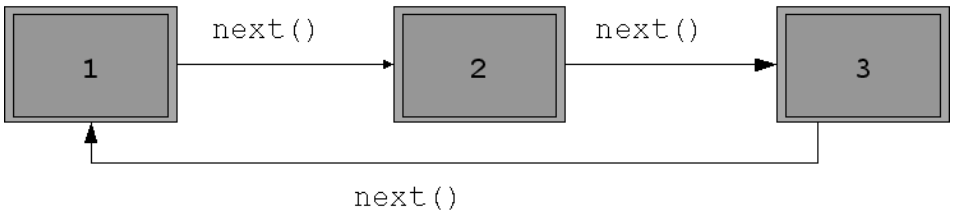
El seu constructor és `public CardLayout()`.

Per anar canviant entre els diferents components de la pila, la classe `CardLayout` disposa d'un seguit de mètodes. En tots el paràmetre es refereix al contenidor en què s'ha aplicat el *layout*:

- `public void first(Container parent)`: Salta al primer component afegit.
- `public void last(Container parent)`: Salta al darrer component afegit.
- `public void next(Container parent)`: Salta al component afegit a continuació del visualitzat actualment.
- `public void previous(Container parent)`: Salta al component afegit tot just abans del visualitzat actualment.

La figura 1.9 mostra com s'alternen els components d'un CardLayout.

FIGURA 1.9. Esquema de funcionament del CardLayout.



GridBagLayout

El GridBagLayout és el més versàtil i complex de tots els *layouts*. Justament per la seva complexitat, ens limitarem a donar una idea general del seu funcionament, ja que una explicació detallada seria molt extensa.

Aquest *layout* és conceptualment similar al GridLayout, i divideix el contenidor en una matriu de cel·les. La particularitat és que en aquest cas les diferents files i columnes poden ser de mida desigual i els components inclosos poden ocupar diverses cel·les contigües, tant en diferents files com columnes. Els components sempre ocupen tot l'espai possible de les cel·les assignades.

El seu constructor és `public GridBagLayout()`.

Totes les propietats especials de les cel·les es defineixen amb l'ajut de la classe auxiliar `GridBagConstraints`. Les crides al mètode `add` al contenidor que usa

aquest *layout* contenen tant el component a afegir com una instància d'aquesta classe:

```
1 add(Component comp, GridBagConstraints constraints)
```

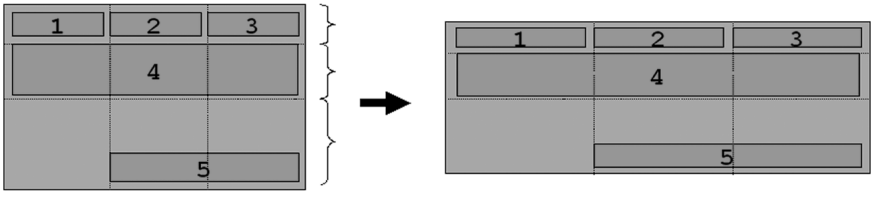
La classe `GridBagConstraints` té un conjunt de propietats amb les quals definim com s'ubica l'element afegit dins la graella i quantes caselles ocupa per cada eix. Algunes de les més significatives són:

- **gridx, gridy**. Especifica la fila i columna en l'extrem superior dret del component. La primera fila i columna de la graella són les posicions zero. Si no s'especifica, l'element s'ubicarà tot just després de l'afegit tot just abans.
- **gridwidth, gridheight**. Indica el nombre de cel·les horitzontals o verticals que ocupa el component. Per defecte és 1. Si s'usa la constantGridBagConstraints.REMAINDER, significa que es volen ocupar totes les files o columnes restants fins al final.
- **ipadx, ipady**. Especifica un farcit extra al voltant del component, internament, en píxels. Per tant, el component mai serà més petit que aquest valor. Per defecte és zero.
- **insets**.Similar al cas anterior, però el farcit és extern, per la qual cosa es crea un cert espai de separació entre aquest component i la resta que l'envolten. Per defecte és zero.
- **anchor**. Usat quan el component és més petit que la cel·la, de manera que indica a quin extrem d'aquesta s'ha d'ajustar. GridBagConstraints especifica un seguit de constants per a aquest valor: `CENTER` (per defecte), `PAGE_START`, `PAGE_END`, `LINE_START`, `LINE_END`, `FIRST_LINE_START`, `FIRST_LINE_END`, `LAST_LINE_END`, i `LAST_LINE_START`.

Tot i que res no impedeix reusar objectes GridBagConstraints en crides successives del mètode add per als casos de components amb característiques idèntiques, és millor usar instàncies diferents per evitar confusions.

La figura 1.10 mostra un exemple de redimensionat de GridBagLayout. Les línies de separació entre cel·les no es visualitzen en realitat, només apareixen en la figura per facilitar la comprensió del seu funcionament. El component 5 és un exemple de l'element més petit que les cel·les que ocupa, i per tant ha està afegit amb un *anchor* de LAST_LINE_END.

FIGURA 1.10. Exemple de redimensionat de GridBagLayout



GroupLayout

Des de la versió 1.6, el Java incorpora el *layout* GroupLayout, molt orientat al desenvolupament automàtic d'*interfaces* gràfiques mitjançant eines auxiliars. Res no impedeix usar-lo manualment, tot i que, com el GridBagLayout, té un cert grau de complicació.

La filosofia d'aquest *layout* és desplegar els elements al llarg dels dos eixos de coordenades (vertical i horitzontal). Al llarg d'aquests eixos, els elements s'agrupen mitjançant grups jeràrquics, de manera que donat un grup, hi pot haver continguts, components, altres grups, o espais en blanc. L'addició de grups és el que permet fer una organització jeràrquica, mentre que els espais són espais en blanc, són separacions buides entre elements. Els grups estan representats per la classe `GroupLayout.Group`, en la qual hi ha un conjunt de mètodes que permeten afegir-hi elements:

- `addComponent (Component comp)`
- `addGroup(GroupLayout.Group group)`
- `addGap(int size)`

Al mateix temps, hi ha dos tipus de grups: seqüencials i paral·lels. En el primer cas els elements s'ubiquen un darrera l'altre al llarg de l'eix de coordenades corresponent, mentre que en el segon cas s'ubiquen en paral·lel. De manera similar a les Box, hi ha una classe concreta per a cada subtipus: `SequentialGroup` i `ParallelGroup`. Per crear algun d'aquest grups cal cridar els mètodes definits en la classe GroupLayout:

- `GroupLayout.Group createSequentialGroup()`
- `GroupLayout.Group createParallelGroup()`

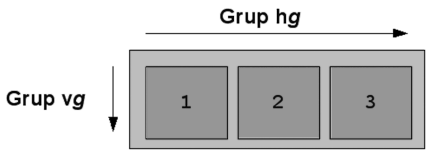
La clau d'aquest *layout* està en el fet que almenys hi ha d'haver un grup associat a l'eix vertical i un altre a l'horitzontal, i tots els components han de pertànyer a dos grups, un de l'eix vertical i un de l'horitzontal. A partir del grup on estan associats, se'n pot calcular la ubicació.

La millor manera d'entendre com es fa aquest càlcul d'ubicació és mitjançant un exemple. La figura 1.11 mostra un GroupLayout en què hi ha un únic grup paral·lel associat a l'eix vertical i un de seqüencial a l'eix horitzontal. Cada component visualitzat (1, 2 i 3) pertany a tots dos grups i s'ha afegit al grup en ordre de numeració incremental. Els espais entre elements en realitat no existirien, s'han posat en la imatge per fer-la més clara.

```
1 GroupLayout layout = new groupLapout(panel)
2 GroupLayout.SequentialLayout hg =
3 layout.createSequentialGroup();
4 GroupLayout.ParallelLayout vg = layout.createParallelGroup();
5 JLabel l1 = new JLabel("1");
6 JLabel l2 = new JLabel("2");
7 JLabel l3 = new JLabel("3");
```

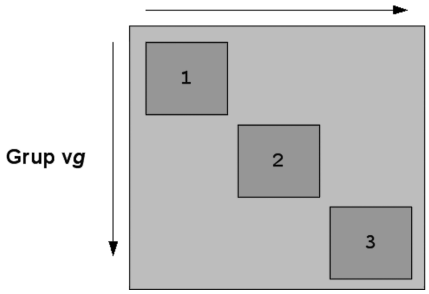
```
8  hg.addComponent(l1);
9  hg.addComponent(l2);
10 hg.addComponent(l3);
11 vg.addComponent(l1);
12 vg.addComponent(l2);
13 vg.addComponent(l3);
14 layout.setHorizontalGroup(hg);
```

FIGURA 1.11. GroupLayout seqüencial (eix horitzontal) i paral.lel (eix vertical).



En contraposició, en la figura 1.12 es mostra com s'ubicarien els components si els dos grups fossin seqüencials en els dos eixos. Els elements s'han afegit als grups igual que en el cas de la figura 1.11, però com es pot apreciar, en lloc d'ubicar-se paral·lelament en l'eix vertical avancen una posició perquè són un grup seqüencial.

FIGURA 1.12. GroupLayout seqüencial (eix horitzontal) i seqüencial (eix vertical).



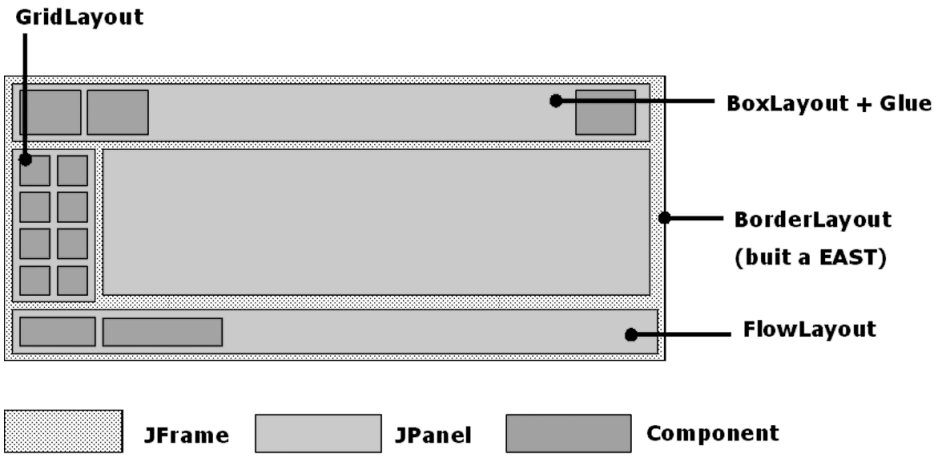
Una de les particularitats més importants del GroupLayout és que permet crear composicions d'elements sense haver de dependre de subpanells auxiliars. Els grups ja fan aquest paper.

1.1.5 Creació d'interfícies complexes

A partir de l'estudi dels diferents *layouts* existents, es pot arribar a la conclusió que molts, per si mateixos, no són suficients per generar una *interface* gràfica d'una certa complexitat. Aquest fet és especialment evident en casos com el BorderLayout, en què en cada zona només hi pot haver un component i, per tant, només hi podria haver cinc components en tota la *interface*.

Això es deu al fet que, normalment, un entorn gràfic no es pot resoldre amb un únic *layout*, sinó que cal la combinació de diferents *layouts* usats en diversos panells dins la finestra principal, tal com mostra la figura 1.13. De fet, els *layouts* és el que dóna sentit a l'existència de la classe JPanel, un contenidor que defineix àrees dins la finestra principal.

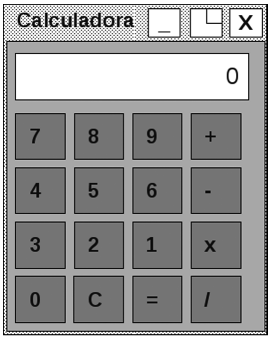
FIGURA 1.13. Combinació de layouts per generar una interfície gràfica complexa.



El fet d'haver d'estudiar com es combinen diferents *layouts* per assolir el resultat desitjat és un dels factors que donen una certa complexitat al disseny d'una *interface* gràfica i el que fa recomanable l'ús d'un IDE.

A continuació es mostra un exemple de com es poden combinar diferents *layouts* per generar la calculadora que es mostra en la figura 1.14. Concretament, es combina un BorderLayout amb un GridLayout per aconseguir l'efecte final.

```
1  //Contenidor d'alt nivell: finestra principal
2  JFrame frame = new JFrame("Calculadora");
3  JPanel panell = calculadora.getContentPane();
4  panell.setLayout(new BorderLayout());
5  ...
6  JLabel display = new JLabel();
7  display.setHorizontalAlignment(SwingConstants.RIGHT);
8  ...
9  //Display de la calculadora a la zona superior
10 panell.add(display, BorderLayout.NORTH);
11 ...
12 JPanel panellBotons = new JPanel();
13 panellBotons.setLayout(new GridLayout(4, 4));
14 JButton boto7 = new JButton("7");
15 ...
16 panell.add(boto7);
17 ...
18 //Panell de botons a la zona central
19 panell.add(panellBotons, BorderLayout.CENTER);
20 ...
```



Calculadora creada combinant layouts.

El Netbeans i layouts absoluts

El Netbeans disposa d'un *layout* de posicionament absolut: la classe `org.netbeans.lib.awtextra.AbsoluteLayout`.

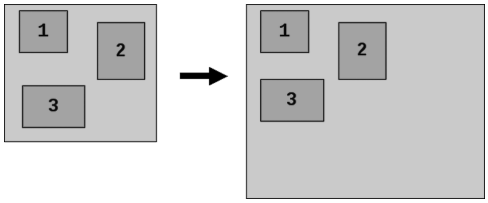
Afortunadament, a l'hora de generar *interfaces* amb composicions complexes, en què cal ubicar els components de manera molt especial, hi ha una alternativa molt útil: els *layouts* de posicionament absolut (o, simplement, *layouts* absoluts). Algunes biblioteques externes a les estàndard del Java proporcionen un tipus especial de *layout* que permet el posicionament absolut dels components. D'aquesta manera, es pot indicar la ubicació i mides exactes de cada component, que mai varia al llarg de l'execució de l'aplicació independentment que es redimensioni el contenidor.

setResizable

Aquest mètode serveix per habilitar/deshabilitar la capacitat de redimensionar una finestra. Evitant que es pugui redimensionar la finestra podem eludir el problema del redimensionat del layout absolut.

Si bé aquest tipus de *layout* pot estalviar molta feina, cal ser molt conscient de les implicacions del seu ús. Per una banda, al redimensionar el contenidor d'alt nivell, tot segueix igual, i per tant, tot el nou espai extra queda buit, tal com mostra la figura 1.14

FIGURA 1.14. Redimensionat d'un layout absolut.



D'altra banda, mai no s'ha d'oblidar que els *layouts* absoluts són aliens a les biblioteques estàndard del Java. Cal incloure'ls com a classes addicionals dins de les aplicacions que els usen en desplegar-les, o aquestes no funcionaran.

1.2 Connexió de la interfície a l'aplicació

Mitjançant la combinació d'objectes de les classes definides en la biblioteca Java Swing és possible generar finestres amb tots els components correctament ubicats i visualitzats, però qualsevol interacció per part de l'usuari no fa res. Per interconnectar la *interface* gràfica generada amb la lògica interna del programa, quan l'usuari dóna una ordre, aquesta es tradueix en una interacció directa amb els objectes que componen la lògica interna del programa, i en canvia l'estat.

Aquesta tasca d'interconnexió no és trivial si és vol fer la feina ben feta, ja que hi ha problemes que, si no es preveuen, tenen un impacte greu en l'escalabilitat de l'aplicació, i incrementen la possibilitat que qualsevol lleugera modificació impliqui canvis en moltes altres classes. El més important de tot és no respectar el principi d'encapsulació, barrejant el codi vinculat exclusivament a la gestió de la *interface* gràfica amb el de la lògica del programa. Si això succeeix, el disseny generat quedarà lligat per sempre a aquesta *interface*, i adaptar-lo a una altra implicarà modificacions. Les classes deixen de ser directament reusables. Per tant, l'objectiu principal és separar les classes vinculades a la *interface* amb les vinculades a la lògica interna, o estat, de l'aplicació.

Per lògica interna del programa s'entén les instàncies de les classes generades en el procés de disseny de l'aplicació.

El **cas ideal d'interconnexió és** aquell en què les classes del model estàtic UML, resultat del procés de disseny, es poden integrar dins de qualsevol *interface*, sigui quina en sigui l'aparença, sense haver de fer absolutament cap canvi sobre aquestes.

Perquè l'aplicació final sigui escalable i reusable, s'ha d'establir una estratègia que es pugui usar en qualsevol llenguatge de programació.

1.2.1 El patró Model-Vista-Controlador

Per a interconnectar la lògica interna de l'aplicació, generada en el procés de disseny en forma de diagrama UML, amb la *interface* d'usuari, una immensa majoria de llenguatges, incloent-hi el Java, es decanta pel patró de disseny anomenat **Model-Vista-Controlador** (MVC).

Un **patró de disseny** és una estratègia a seguir per resoldre un problema determinat dins el procés de disseny del programari, de manera es pugui emprar en un ampli ventall de situacions. De totes maneres, sempre cal adaptar aquesta estratègia als detalls de cada cas concret.

El patró MVC és aplicable a qualsevol aplicació que permet la interacció amb l'usuari. No és exclusiu d'aplicacions amb interfaces gràfiques.

En aquest cas, el problema que hi ha en el procés de disseny de programari que es vol resoldre és l'esmentat anteriorment: com es pot separar de manera efectiva la lògica interna del programa de la *interface* d'usuari, de manera que modificacions en una part impliquin canvis mínims en l'altra.

El patró MVC divideix les diferents classes de l'aplicació en tres conjunts diferenciats, segons el rol. Aquesta diferenciació és exclusivament conceptual i no s'ha de traduir en algun tipus de relació entre classes a nivell de diagrama UML (associació, herència, etc.).

MVC i SmallTalk

El patró MVC té els orígens en el llenguatge SmallTalk, cosa que no és gens estranya si es té en compte que aquest llenguatge va ser dissenyat per fer la *interface* gràfica del Dynabook.

Les classes del **Model** representen la lògica interna del programa i contenen l'estat de l'aplicació, i proporcionen totes les funcionalitats exclusives a l'aplicació, independents de la *interface*. Aquest grup està compost principalment per les classes que el dissenyador ha reflectit en el model estàtic UML.

Les classes de la **Vista** representen l'aspecte purament vinculat a la *interface* d'usuari, gràfica o no. Aquestes s'encarreguen tant de capturar les interaccions de l'usuari com d'accedir a les dades emmagatzemades en el model, de manera que l'usuari les pugui visualitzar i manipular correctament. Per tant, una de les seves responsabilitats principals és mantenir la consistència entre les dades internes i el que visualitza l'usuari. Qualsevol classe usada per gestionar un element on visualitzar la informació o donar ordres a l'aplicació forma part d'aquest grup. Per exemple, la classe que gestiona una impressora o un panell de LED també es consideraria part de la Vista. Tots els components gràfics Swing de l'aplicació pertanyen exclusivament a aquest grup.

La torre Agbar de Barcelona és un enorme panell de LED (Light Emitting Diodes, diodes emissors de llum) que forma part de la Vista del programa que la controla.

Les classes del **Controlador** representen la capa intermèdia entre dades i *interface* que s'encarrega de traduir cada interacció de l'usuari, capturada per la Vista, en crides a mètodes definits en el Model, de manera que s'executi la lògica interna adequada a l'ordre donada per l'usuari. En aquest grup hi haurà una classe per a cada funcionalitat que es vulgui incorporar a la *interface*.

Mitjançant les interaccions dels objectes de classes dels diferents conjunts es genera una aplicació que respon a les ordres de l'usuari. Atès que aquestes interaccions, en darrera instància, sempre es tradueixen en crides a mètodes,

aquest patró de disseny estableix els mecanismes necessaris perquè un usuari pugui cridar mètodes sobre les instàncies de les classes del diagrama UML resultat del disseny de l'aplicació.

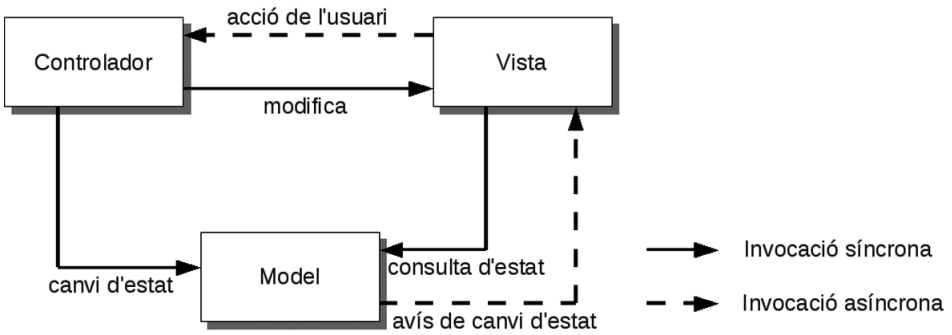
S'entén per fer doble clic pitjar un botó del ratolí dues vegades molt ràpidament.

L'esquema d'actuació dels elements del patró MVC és el següent:

1. L'usuari actua sobre una instància d'una classe de la Vista, per exemple, un botó.
2. L'objecte rep l'acció i la passa a una instància d'una classe del Controlador. En aquest procés es transmet tota la informació addicional necessària per al tractament correcte de l'acció. Per exemple, si s'ha fet amb el botó dret o esquerre del ratolí, o si s'ha fet doble clic.
3. L'objecte Controlador crida els mètodes que pertoqui del Model per aconseguir el resultat associat a l'acció de l'usuari.
4. L'estat dels objectes del Model canvia.
5. El Model avisa la Vista que hi ha hagut canvis.
6. Els objectes de la Vista que mostren la informació associada a l'estat del Model criden els mètodes de consulta necessaris per mostrar correctament el nou estat.

La figura 1.15 mostra un resum de les interaccions entre els objectes dels tres conjunts. Les diferents interaccions normalment es divideixen entre les que es tradueixen en crides a mètodes en moments clarament identificables durant l'execució de l'aplicació, les crides síncrones, i les que no ho són, ja que poden sorgir en qualsevol moment, les crides asíncrones.

FIGURA 1.15. Esquema d'interaccions Model-Vista-Controlador.



Entorns heterogenis

Alguns exemples típics són la portabilitat a dispositius mòbils o sistemes encastats (caixers automàtics, caixes enregistradores, etc.).

A part dels beneficis esmentats, tot seguit es mostren algunes altres aportacions d'usar el patró MVC. El preu a pagar per obtenir tots aquests beneficis és un cert increment en la complexitat de l'aplicació. Aquest, però, és un preu que val la pena pagar. Són:

- **Més reusabilitat de les classes**. La separació entre les classes del Model i la Vista permet implementar més fàcilment aplicacions en què hi ha diferents mecanismes per visualitzar la informació en paral·lel. Això també permet

facilitar el test i el manteniment d'aquestes classes, ja que tot l'accés a l'estat de l'aplicació sempre es realitza per mitjà d'aquestes.

- **Millor portabilitat en entorns heterogenis**. L'adaptació de l'aplicació a nous sistemes amb diferents capacitats per visualitzar la informació només implica la implementació d'una nova Vista. El Model es pot mantenir íntegre sense que calgui cap modificació.

Tot i els passos enumerats, val la pena comentar que hi ha situacions en què és factible no obeir el model directament i fer que sigui el Controlador el que forci l'actualització de la Vista. Si bé aquesta no és una aproximació pura, no es perd cap dels avantatges descrits del patró MVC.

1.2.2 Control d'esdeveniments

Una de les particularitats de les aplicacions interactives és que, un cop es posen en marxa, aquestes es queden parcialment o totalment inactives a l'espera que l'usuari faci alguna acció. Fins que això no succeeix, no s'executa cap codi associat a les funcionalitats de l'aplicació. Aquesta circumstància es plasma dins el patró MVC amb les crides asíncrones, també anomenades esdeveniments, entre els objectes de la Vista i el Controlador. És per això que la biblioteca Java Swing implementa aquest patró mitjançant el mecanisme anomenat **control d'esdeveniments**. Els esdeveniments són generats pel motor gràfic del Java en resposta a accions de l'usuari. El programador no s'ha de preocupar de com es generen realment.

Sempre s'executa el codi vinculat al motor gràfic que gestiona la visualització de la interfície.

Les accions de l'usuari sobre components Swing generen **esdeveniments**. Aquests esdeveniments són associats a fragments de codi, que s'executen cada cop que tenen lloc.

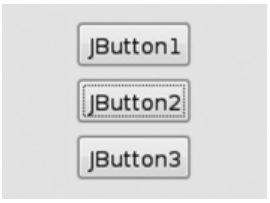
Reflexionant una mica, el concepte d'esdeveniment asíncron amb un codi associat no és un aspecte totalment nou, ja que conceptualment no és gens diferent de l'usat en el Java per al control d'errors mitjançant excepcions.

Swing defineix un conjunt de classes que representen cada tipus d'interacció, genèricament, que l'usuari pot realitzar sobre un component. Com en el cas de les excepcions, els esdeveniments són objectes, resultants de la instanciació d'alguna d'aquestes classes, que el programador pot manipular per obtenir informació més detallada respecte a les particularitats de l'acció que l'han generat. Totes elles pertanyen al paquet java.awt.event.

Gestió de focus

Un control guanya el focus sempre que és usat. Normalment, el focus també es pot desplaçar entre controls amb la tecla de tabulador.

Alguns dels tipus d'esdeveniments més típics són els següents:



El control JButton2 té el focus en aquesta imatge.

- **ActionEvent**: Es genera en realitzar l'acció més típica, o estàndard, sobre un control. Cada control estableix quina considera que és la seva acció estàndard, que pot ser diferent per a cada cas. Per exemple, en el cas d'un botó, es genera en pitjar-lo.

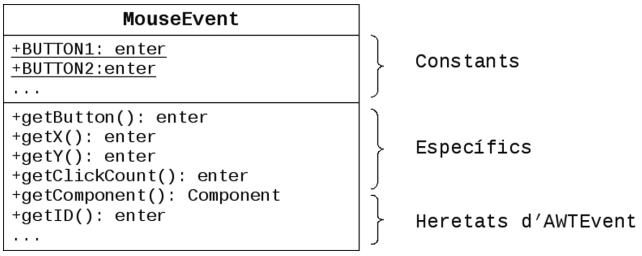
- **MouseEvent**: Generat davant qualsevol acció vinculada exclusivament al ratolí. Per exemple, en pitjar qualsevol botó del ratolí, moure l'apuntador dins una àrea concreta, etc.
- **KeyEvent**: Associat a accions exclusivament relatives al teclat. Per exemple, pitjar una tecla o deixar-la anar.
- **WindowEvent**: Qualsevol esdeveniment relatiu a l'estat d'una finestra. Per exemple, minimitzar-la, maximitzar-la, redimensionar-la o tancar-la.
- **FocusEvent**: Aquest esdeveniment ve donat per accions vinculades al focus de controls. Amb focus es refereix al fet que un control queda remarcat dins la *interface*, de manera que s'hi pot interactuar directament mitjançant el teclat. Exemples d'aquest tipus d'esdeveniment són guanyar o perdre el focus.
- **TextEvent**: Generat en realitzar accions relatives a camps de text. Per exemple, modificar un camp de text.

No tots els components Swing poden generar absolutament tots els tipus d'esdeveniments, sinó que només generen els associats a interaccions que realment poden rebre. De fet, alguns estan molt vinculats a components molt específics: per exemple, en una aplicació d'escriptori, només els JFrame generen WindowEvent i només els JTextComponent generen TextEvent. Per veure amb detall quins esdeveniments llença cada component Swing davant cada tipus d'acció, és necessari mirar la documentació de Java.

La figura 1.16 mostra un exemple d'esdeveniment: la classe `MouseEvent`. Com es pot veure, aquest disposa d'un conjunt de mètodes que permeten consultar-ne els detalls: quin botó s'ha pitjat, quantes pulsacions s'han fet, quines són les coordenades de la seva posició sobre la finestra principal, etc. Alguns d'aquest mètodes són específics d'un esdeveniment generat pel ratolí, i d'altres són aplicables a qualsevol, heretats de la superclasse `AWTEvent`.

El mètode getComponent retorna el component que ha generat l'esdeveniment.

FIGURA 1.16. Exemple d'esdeveniment: la classe "MouseEvent".



1.2.3 Captura d'esdeveniments

La captura d'esdeveniments, de manera que es puguin tractar dins l'aplicació, es realitza mitjançant uns objectes especials anomenats *Listeners*. Aquests objectes conformen la part de Controlador del patró MVC plasmat en la biblioteca gràfica del Java.

Listeners

Hi ha un tipus de *Listener* diferent per cada tipus d'esdeveniment: objectes ActionListener, que capturen esdeveniments tipus ActionEvent, objectes MouseListener per capturar els MouseEvent, etc. Cada tipus de *Listener* només pot capturar el tipus d'esdeveniment al qual esta associat, i absolutament cap altre.

Nomenclatura dels Listeners

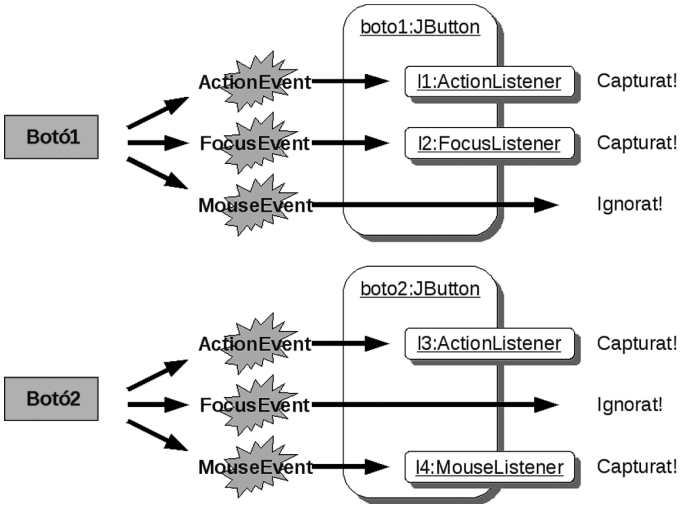
Donat un tipus d'esdeveniment "xxxEvent", normalment, el seu "Listener" associat té el nom "xxxListener". El mètode per assignar-lo a un component es diu "addxxxListener".

Un *Listener* captura els esdeveniments que genera un únic component dins la *interface* gràfica. Perquè pugui acomplir aquesta tasca cal **registrar-lo** en el component. Si per un tipus concret d'esdeveniment el component no té cap *Listener* associat registrat, aquests s'ignoraran, independentment del fet que l'usuari pugui fer l'acció sobre el component. No passarà res a l'aplicació en fer-la. Per fer el registre, cada component disposa d'un mètode diferent per cada tipus d'esdeveniment que pot generar. Per exemple, els components que generen ActionEvent disposen d'un mètode addActionListener que permet registrar-hi un ActionListener. En contraposició, els components que no poden generar aquest tipus d'esdeveniment no disposen d'aquest mètode i, per tant, no poden tenir registrat cap ActionListener.

> Els objectes **Listener** són els encarregats de capturar els diferents tipus d'esdeveniment, adoptant el rol de Controlador del patró MVC. Per poder fer-ho, cal que estiguin **registrats** en els components que generen els esdeveniments a capturar.

Cada component té la seva llista individualitzada d'objectes *Listener*, tants com tipus d'esdeveniment calgui capturar pel component. La figura 1.17 mostra un esquema d'aquest mecanisme per dos botons diferents dins una *interface* gràfica. D'acord amb la seva llista de *Listeners*, cada botó respon de manera diferent davant els diferents esdeveniments que generen individualment. Val la pena remarcar que cada *Listener* que apareix en la figura és un objecte diferent.

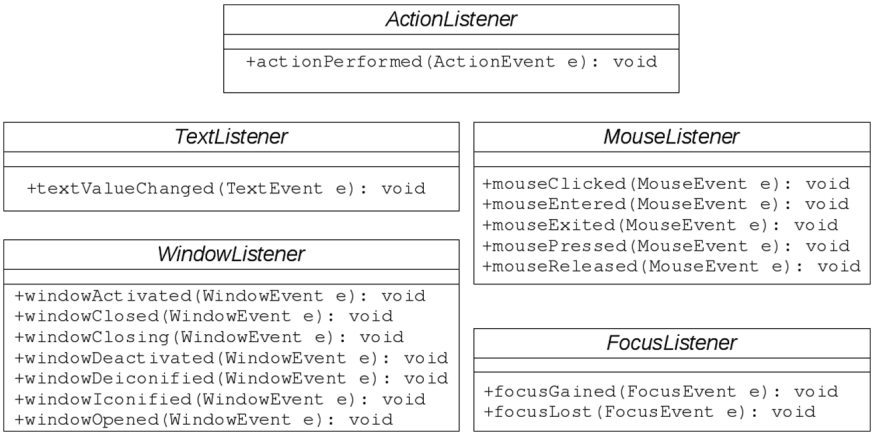
FIGURA 1.17. Exemple de registre de Listeners i captura d'Events.



La biblioteca gràfica del Java defineix dins de cada tipus de *Listener* un conjunt de mètodes, cadascun dels quals correspon a una interacció més concreta del que marca l'esdeveniment. Per exemple, la generació d'un MouseEvent per si mateixa només indica que ha passat alguna cosa amb el ratolí, però res més. El nombre d'aquests mètodes varia segons el tipus d'interaccions més especifiques que pot significar cada tipus d'esdeveniment.

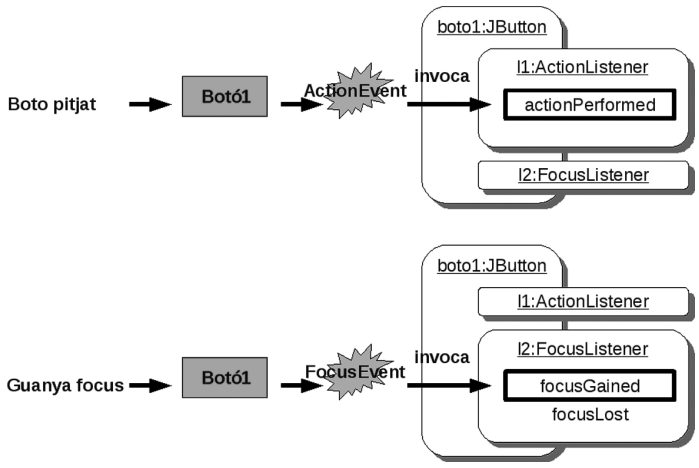
La figura 1.18 mostra els *Listeners* més usats d'entre els definits pel Java, amb tots els seus mètodes.

FIGURA 1.18. Exemple d'alguns Listeners típics, amb els seus mètodes definits.



Quan un *Listener* captura un esdeveniment, immediatament crida el mètode corresponent a l'acció més concreta que ha dut a terme la generació de l'esdeveniment. Per exemple, si es genera un MouseEvent per la pulsació d'un botó del ratolí, el MouseListener, en capturar-lo, executarà immediatament el mètode `mouseClicked`. D'aquesta manera, és possible fitar quina acció concreta ha dut a terme l'usuari sobre el component que ha generat l'esdeveniment. El mateix esdeveniment es passa com a paràmetre en aquesta crida, de manera que és possible consultar-ne els detalls (quin botó s'ha pitjat, la seva posició, etc.). Tot aquest comportament es produeix automàticament, gestionat pel motor gràfic del Java.

FIGURA 1.19. Crida dels mètodes als Listeners.



Un cop s'ha entès com es porta a terme la captura d'esdeveniments i què passa quan es porta a terme, és el moment de veure la darrera peça que falta: com s'executa un tros codi concret quan un usuari realitza una acció sobre la *interface* gràfica Swing.

Interfaces Java són classes abstractes pures, sense cap mètode codificat a l'interior. S'hereta d'aquestes amb la paraula clau implements.

Dins la biblioteca gràfica del Java, tots els diferents tipus de *Listener* es troben definits com a *interfaces* Java. Concretament, tots hereten de la *interface* comú EventListener. Per tant, en realitat, no és possible instanciar directament un *Listener* a partir d'ells, ja que aquests només proporcionen la definició dels mètodes mostrats en la figura 1.19, però no contenen cap codi executable. Per poder instanciar un *Listener* d'un tipus concret, el desenvolupador ha de crear una classe pròpia que implementi la *interface* corresponent i, per tant, codificar mitjançant sobreescriptura tots i cadascun del mètodes definits en la *interface*. L'objecte *Listener* que realment es registra en un component és una instància d'aquesta classe creada pel desenvolupador.

És justament en aquest punt, en la implementació dels mètodes definits en la *interface*, que s'assigna el codi que es vol executar davant l'acció d'un usuari. Pel mecanisme de polimorfisme, quan el *Listener*, prèviament registrat en un component, capturi un esdeveniment i cridi el mètode corresponent a l'acció de l'usuari, el codi que s'executarà serà l'assignat a aquest mètode en la classe creada pel desenvolupador. Així, doncs, es pot veure com gràcies a l'ús de mètodes polimorfes ha estat possible crear la biblioteca gràfica del Java de manera que pugui ser adaptada a qualsevol aplicació.

El **desenvolupador** ha de generar una classe per cada *Listener* que vulgui usar dins l'aplicació. Cadascuna d'aquestes classes implementa la *interface Listener* associada al tipus d'esdeveniment a controlar. Les seves instàncies són les que realment es registren en els components.

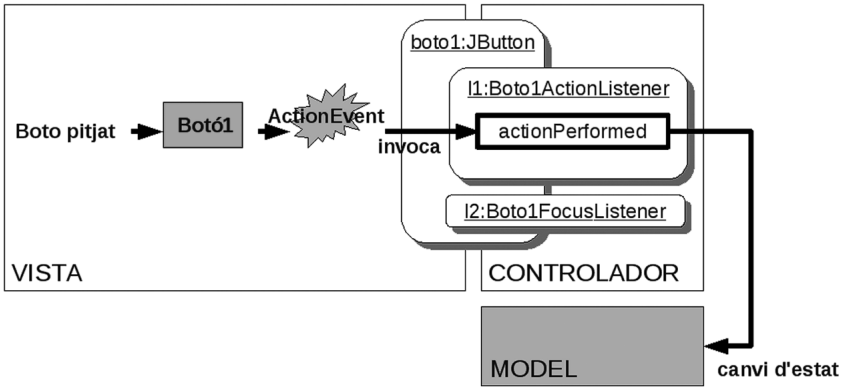
Donats els rols dels elements del patró MVC, és justament dins els mètodes sobreescrits a aquests *Listeners* personalitzats des d'on cal fer les crides cap a objectes del Model. La figura 1.20 mostra una visió general de tot el sistema, associant ja cada part implicada al patró MVC. Recordem que, pel mecanisme d'herència, un objecte Boto1ActionListener també és un ActionListener.

Una de les conclusions d'aquest mecanisme és que, atès que el codi associat a una mateixa acció en diferents components normalment també serà diferent, això implica que el desenvolupador ha de crear una classe pròpia *Listener* per cada tipus d'esdeveniment i cada component dins la *interface* gràfica. Per exemple, si hi ha tres botons i es vol assignar codi diferent a l'acció de pitjar cada un, caldrà definir tres classes diferents que implementin la *interface* ActionListener. A cada una s'assignarà el codi que correspongui, diferent dels altres, als mètodes `actionPerformed` respectius. Un cop fet, ja només resta crear una instància de cadascuna d'aquestes tres classes i assignar una a cada botó, d'acord amb el codi que es vol executar en pitjar cada un. La figura 1.20 esquematitza aquest comportament.

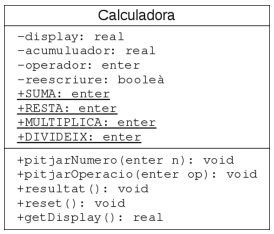
FIGURA 1.20. Assignació de codi als esdeveniments usant mètodes polimórfics i usant el patró MVC.

```
import java.awt.event.ActionListener;

public class Boto1ActionListener implements ActionListener{
  void actionPerformed(ActionEvent e){
    //Canvi d'estat del model invocant mètodes als seus objectes
    ...
  }
}
```



Tot i que normalment es registra un objecte *Listener* diferent en cada component per cada tipus d'esdeveniment, res no impedeix reusar el mateix objecte per a un mateix tipus d'esdeveniment en diferents components. La condició per fer-ho és que el codi que cal executar en capturar l'esdeveniment sigui exactament el mateix per a tots els casos.



Esquema de la classe Calculadora.

Prenent l'exemple de la calculadora, a continuació es mostra un fragment de codi que serveix per il·lustrar aquest fet. Un dels aspectes que val la pena destacar de l'exemple és el fet que tots els *Listeners* es defineixen com a classes privades. Aquest és un fet molt habitual per dos motius. D'una banda, i el menys important, pel fet que els *Listeners* propis es poden considerar classes auxiliars de la *interface* gràfica. D'aquesta manera s'evita tenir una quantitat enorme de classes en fitxers .java separats. D'altra banda, i com a motiu principal, això permet que des del codi dels *Listeners* es pugui accedir directament als atributs de la*interface* gràfica. Això és important, ja que no és possible cridar mètodes d'objectes del model sense tenir-hi una referència. En l'exemple, això es veu en la crida de mètode als objectes `calculadora:Calculadora` i `display: JTextField`.

```
1  public class CalculadoraGUI {
2
3    //Model
4    private Calculadora calculadora = new Calculadora();
5    //Display consulta estat del Model per inicialitzar-se
6    private JLabel display =
7                   new JLabel(Float.toString(calculadora.getDisplay()));
8
9    private class B1ActListener implements ActionListener {
10     public void actionPerformed(ActionEvent e){
11       calculadora.pitjarNumero(1);
12       display.setText(
13       Float.toString(calculadora.getDisplay()));
14     }
15   }
16   ...
17   private class SumActListener implements ActionListener {
18     public void actionPerformed(ActionEvent e){
```

```
      calculadora.pitjarOperacio(Calculadora.SUMA);
      display.setText(
      Float.toString(calculadora.getDisplay()));
    }
  }
  ...
  private class ResActListener implements ActionListener {
    public void actionPerformed(ActionEvent e){
      calculadora.reset();
      display.setText(
      Float.toString(calculadora.getDisplay()));
    }
  }
  ...
  JButton boto1 = new JButton("1");
  boto1.addActionListener(new B1ActListener());
  ...
  JButton botoSuma = new JButton("+");
  botoSuma.addActionListener(new SumActListener());
  ...
  JButton botoC = new JButton("C");
  botoC.addActionListener(new ResActListener());
  ...
}
```

Un exemple de reutilització d'un mateix objecte *Listener* en diferents components es mostra a continuació pels diferents botons numèrics. Si el mètode `actionPerformed` es codifica d'una manera adequada, en realitat tots els botons han de fer el mateix: cridar el mètode `pitjarBoto` sobre el model (l'objecte calculadora).

parseInt és mètode estàtic de la classe Integer que permet transformar una cadena de text en enter.

```
private class BotoNumeroListener implements ActionListener {
  public void actionPerformed(ActionEvent e) {
    JButton botoPitjat = (JButton)e.getSource();
    calculadora.pitjarNumero(
    Integer.parseInt(botoPitjat.getText()));
    display.setText(
    Float.toString(calculadora.getDisplay()));
  }
}
...
//S'instancia un únic Listener
ActionListener botoNumeroListener = new BotoNumeroListener();
//Es generen els 10 botons numèrics amb un bucle
for (int i=0;0<10;i++) {
  JButton boto = new JButon(Integer.toString(i));
  //S'assigna sempre el mateix objecte Listener per tots
  boto.addActionListener(botoNumeroListener);
  panellBotons.add(boto);
  ...
}
```

Adaptadors

Atès que els adaptadors són classes normals, s'hereta dels adaptadors usant la sentència "extends".

Atès que els *Listeners* són *interfaces*, el desenvolupador està obligat a sobreescriure tots els seus mètodes sense cap excepció, o en cas contrari el compilador retornarà un error. Això vol dir que si per algun dels mètodes estesos definits no es vol realitzar realment cap acció, serà necessari deixar el mètode buit, sense codi. En els casos de *Listeners* amb diversos mètodes, per exemple, el WindowListener, això pot ser pesat i fer el codi més confús. Per aquest motiu, la biblioteca gràfica del Java defineix un conjunt de classes *Listener* no abstractes amb tots els mètodes

ja sobreescrits, però de contingut buit: els **adaptadors** (*adapters*). Per tant, el desenvolupador també pot generar diferents tipus de *Listener* heretant-ne d'ells i només sobreescrivint els mètodes que realment vol usar.

Tots els adaptadors també estan definits en el paquet `java.awt.event`. Hi ha un adaptador per cada tipus de *Listener* amb més d'un mètode: FocusAdapter, WindowAdapter, KeyAdapter, etc. No hi ha adaptadors per a *Listeners* amb un únic mètode, com l'ActionListener o el TextListener, ja que en aquest sentit no aporten res.

Un error típic: el problema de les majúscules i minúscules

Un error típic del programador inexpert és, en decidir sobreescriure un mètode, equivocar-se en una majúscula o minúscula en el nom del mètode respecte a l'original de la superclasse. Atès que el Java és sensible a majúscula/minúscula (és *case-sensitive*), quan això passa, en realitat no s'ha sobreescrit el mètode, sinó que se n'ha generat un de nou amb un nom molt semblant. El compilador ho accepta i no retorna cap error. Atès que el mètode original es manté invariable, que en el cas dels adaptadors vol dir sense codi, l'aplicació no farà absolutament res en realitzar aquella acció, ja que en capturar l'esdeveniment es crida el mètode original buit.

El *Listener* següent no fa res quan el component en què s'ha registrat guanya el focus:

```
1 public class ElMeuListener extends FocusAdapter {
2   void focusgained(FocusEvent e) {
3    //Realitzar acció...
4   }
5 }
```

Classes anònimes

En la immensa majoria de casos, els *Listeners* que codifica el desenvolupador són classes relativament senzilles i curtes, amb pocs mètodes, i de les quals només s'usa una instància dins de tota l'aplicació, ja que cada *Listener* se sol registrar a un únic component. Atès aquest fet, és útil conèixer un mecanisme que ofereix el Java per definir classes auxiliars de manera encara més simple que les classes privades: les classes anònimes.

Una **classe anònima** no té nom. Es caracteritza perquè en lloc de definir-se com a entitat diferenciada amb una capçalera class, es defineix dins el codi just en el moment precís d'instanciar-la.

Les classes anònimes són un mecanisme del llenguatge Java que es pot usar per a qualsevol situació, però són especialment útils a l'hora de generar *Listeners*. Com les classes privades, les classes anònimes tenen accés directe a qualsevol atribut de la classe en què es defineixen.

Per usar-les, és prerequisit que la classe que es vol definir sigui subclasse d'una altra ja existent. La seva sintaxi és la següent:

```
1 new NomSuperclasse() {
2   //Definició normal de la classe (atributs + mètodes)
3 }
```

Per exemple, els *Listeners* usats en exemples anteriors es poden definir de la manera següent, en lloc de mitjançant classes privades. Fixeu-vos com les classes són definides just en el moment d'instanciar-les (en fer un "new"), en lloc de fer-ho prèviament en un bloc a part. Els fragments de codi en què es defineixen classes anònimes es troben remarcats en negreta.

windowClosing

Perquè una aplicació realment acabi en tancar la finestra principal, cal registrar-hi un WindowListener i fer que el seu mètode finalitzi l'aplicació. Per exemple, cridant la crida System.exit(0).

```
1  public class CalculadoraGUI {
2
3  //Model
4  private Calculadora calculadora = new Calculadora();
5
6  //Contenidor d'alt nivell: finestra principal
7  JFrame frame = new JFrame("Calculadora");
8
9  //Display consulta estat del Model per inicialitzar-se
10 private JLabel display =
11                new JLabel(Float.toString(calculadora.getDisplay()));
12 ...
13
14 JButton botoSuma = new JButton("+");
15
16 //Classe anònima per Listener del botó de sumar
17 botoSuma.addActionListener(new ActionListener() {
18   public void actionPerformed(ActionEvent e) {
19     calculadora.pitjarOperacio(Calculadora.SUMA);
20     display.setText(
21     Float.toString(calculadora.getDisplay()));
22   }
23 });
24 ...
25 JButton botoC = new JButton("C");
26
27 //Classe anònima per Listener del botó de reset
28 botoC.addActionListener(new ActionListener() {
29   public void actionPerformed(ActionEvent e) {
30     calculadora.reset();
31     display.setText(
32     Float.toString(calculadora.getDisplay()));
33   }                              windowClosing
34 });
35 ...
36 //Gestio el tancament de la finestra principal
37 frame.addWindowListener(new WindowAdapter() {
38   public void windowClosing(WindowEvent e) {
39     System.exit(0);
40   }
41 });
42 ...
43 }
```

1.3 Altres elements gràfics

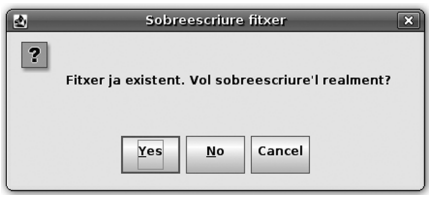
La biblioteca gràfica Swing és molt extensa i ofereix un conjunt de funcionalitats que, si són conegudes pel desenvolupador, poden estalviar-li molta feina. Swing permet realitzar tasques habituals dins la generació d'una *interface* gràfica de manera flexible i amb poc esforç. Per veure com funcionen amb detall absolut les classes implicades és necessari consultar la documentació del Java, en què s'enumeren tots els seus mètodes i s'explica com cridar-los correctament.

1.3.1 Panells d'opcions

Un element molt usat en les *interfaces* gràfiques són els panells d'opcions, que serveixen per avisar l'usuari d'algun esdeveniment o demanar-li confirmació davant alguna acció. L'aplicació queda bloquejada fins que es respon en el panell d'opcions amb alguna de les opcions presentades: Sí/No, Acceptar/Cancel·lar, etc. La figura 1.21 mostra un exemple de panell d'opcions.

L'idioma dels botons sempre està associat a la configuració d'idioma del sistema operatiu.

FIGURA 1.21. Un panell d'opcions.



A Swing, per realitzar un **panell d'opcions** no és necessari crear una finestra i afegir tots els components un a un, ja hi ha una classe que els construeix automàticament i els mostra per pantalla: la classe JoptionPane.

La classe `JOptionPane` presenta una particularitat important respecte a tota la resta de mecanismes emprats a Swing, una gran part dels seus mètodes són estàtics. Cada un està vinculat a la generació d'un tipus de panell d'opcions diferent. Així, doncs, en contraposició amb tots els altres components, els panells d'opcions no se solen generar mitjançant la instanciació d'una classe. Només s'usen constructors per generar panells personalitzats.

Component pare

Un paràmetre que apareix en tots els casos és el component pare del panell d'opcions, normalment, el contenidor d'alt nivell corresponent.

Existeixen quatre blocs de mètodes estàtics, tots amb diferents sobrecàrregues, que permeten generar diferents tipus de panells d'opcions:

- **showConfirmDialog**. Mostra un panell de confirmació, en què es poden establir diferents possibilitats de resposta: Sí/No, Sí/No/Cancel·lar, Acceptar/Cancel·lar.
- **showInputDialog**. Mostra un panell en què l'usuari pot introduir una única línia de text.
- **showMessageDialog**. Mostra un missatge a l'usuari, que només pot acceptar.
- **showOptionDialog**. Permet mostrar qualsevol tipus de panell. La llista de paràmetres permet personalitzar-ne totes les propietats.

Tots els mètodes retornen el valor corresponent a la resposta donada per l'usuari. La classe `JOptionPane` defineix un conjunt de constants que permeten al desenvolupador establir el comportament del panell. Les constants que permeten esbrinar la resposta donada per l'usuari són:

- `JOptionPane.YES_OPTION`, si l'usuari ha pitjat Sí.
- `JOptionPane.NO_OPTION`, si l'usuari ha pitjat No.
- `JOptionPane.CANCEL_OPTION`, si l'usuari ha pitjat Cancel.lar.
- `JOptionPane.OK_OPTION`, si l'usuari ha pitjat *OK*.
- `JOptionPane.CLOSED_OPTION`, si l'usuari ha tancat el panell.

En el cas del mètode `showInputDialog`, no es retorna cap d'aquestes constants, sinó un `String` amb el valor introduït per l'usuari. En cas que l'acció es cancel·li, retorna `null`.

Les que permeten indicar quin tipus de panell (*messageType*) es mostra, reflectit en una icona diferent (un signe d'exclamació, un senyal d'informació, etc.), són:

- `JOptionPane.ERROR_MESSAGE`
- `JOptionPane.INFORMATION_MESSAGE`
- `JOptionPane.WARNING_MESSAGE`
- `JOptionPane.QUESTION_MESSAGE`
- `JOptionPane.PLAIN_MESSAGE`

Cadascuna de les imatges que es mostren correspon a les diferents tipus de constants enumerades, en el mateix ordre. La constant `PLAIN_MESSAGE` no genera cap icona, deixa l'espai buit.

Icones possibles a un JOptionPane.

Finalment, les constants que permeten establir quines combinacions de botons han d'aparèixer en el panell (*optionType*) són:

- `JOptionPane.YES_NO_OPTION`
- `JOptionPane.YES_NO_CANCEL_OPTION`
- `JOptionPane.OK_CANCEL_OPTION`

A continuació es mostra el fragment de codi que generaria el panell que es mostra en la figura 1.21.

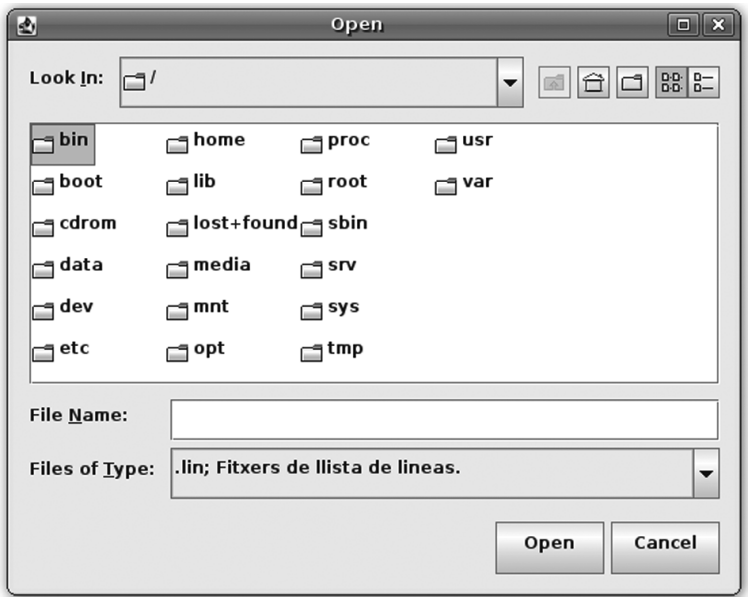
```
1  JFrame finestra = new JFrame();
2  ...
3  int resposta = JOptionPane.showConfirmDialog(finestra,
4    "Fitxer ja existent. Vol sobreescriure'l realment?",
5    "Sobreescriure fitxer",
6    JOptionPane.YES_NO_CANCEL_OPTION,
7    JOptionPane. QUESTION_MESSAGE);
8
9  switch(resposta) {
10   case JOptionPane.YES_OPTION:
11     //Acció per SÍ
12     break;
13   case JOptionPane.NO_OPTION:
14     //Acció per NO
```

```
15      break
16    default:
17      //Acció per CANCELAR
18  }
```

1.3.2 Selectors de fitxers

Un altre dels elements que Swing ja té pregenerat, la qual cosa fa innecessari haver de crear-lo des de zero agregant components individuals, són els panells de selecció de fitxers emprats per obrir o desar dades. La figura 1.22 mostra un exemple d'aquest tipus de panell, concretament d'un per obrir dades.

FIGURA 1.22. Un panell selector de fitxers.



La **classe responsable** de mostrar panells de selecció de fitxers és la `JFileChooser`.

En aquest cas, aquesta classe sí que s'ha d'instanciar. No es basa en mètodes estàtics com `JOptionPane`. Un cop creat l'objecte, hi ha dos mètodes per generar els dos tipus diferents de panells:

- **public int showOpenDialog (Component parent)**. Mostra un panell per obrir un fitxer o directori.
- **public int showSaveDialog (Component parent)**. Igual que l'anterior, però per desar-lo.

En realitat, ambdós panells són pràcticament idèntics, i únicament es diferencien pel títol de la finestra. Com en el JOptionPane, aquests mètodes retornen un valor que es pot comparar amb un conjunt de constants definides per veure si l'operació s'ha realitzat correctament:

- `JFileChooser.CANCEL_OPTION`, si l'usuari ha pitjat *Cancelar*.
- `JFileChooser.APPROVE_OPTION`, si la selecció ha estat correcta.
- `JFileChooser.ERROR_OPTION`, en cas d'error.

Per obtenir el fitxer seleccionat, cal usar el mètode `getSelectedFile`. Si la selecció ha estat correcta, s'obté una instància de la classe `java.io.File`, que és la que el Java usa per fer operacions amb fitxers. En cas contrari, aquest retorna `null`.

Una funcionalitat dels JFileChooser que val la pena comentar és la possibilitat de filtrar els fitxers visualitzats en la finestra, de manera que només es mostren els que tenen una extensió determinada. Aquesta funció se sol usar molt en les aplicacions que usen selectors de fitxers. Els mètodes principals implicats per assolir-la són:

- **public void setFileFilter(FileFilter filter)**. Assigna el filtre actiu.
- **public void addChoosableFileFilter (FileFilter filter)**. Afegeix un filtre a la llista desplegable del selector. Això permet filtrar per diferents tipus d'extensió.

Els filtres són objectes del tipus `javax.swing.filechooser.FileFilter`. Aquesta és una classe abstracta, de manera que és el desenvolupador qui en realitat ha de definir la seva pròpia subclasse i codificar els mètodes abstractes d'acord amb el seu criteri de si, donat un nom de fitxer, aquest s'ha de visualitzar o no. Per sort, existeix una subclasse que proporciona una implementació per defecte de tots els mètodes: la classe `javax.swing.filechooser.FileNameExtensionFilter`. Tot i ser senzilla, serveix per a la majoria de casos.

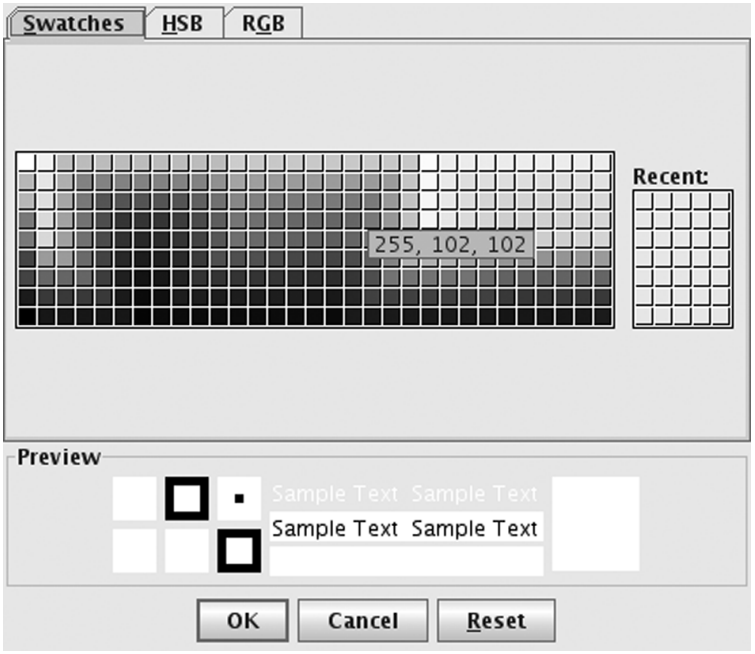
A continuació es mostra el codi que genera el selector de fitxers representat en la figura 1.22.

```
1 JFrame frame = new JFrame();
2 ...
3 JFileChooser selector = new JFileChooser();
4 selector.setFileSelectionMode(JFileChooser.FILES_ONLY);
5 FileNameExtensionFilter filtre =
6   new FileNameExtensionFilter(".lin; Fitxers de llista de lineas.", "lin");
7 selector.setFileFilter(filtre);
8 selector.showOpenDialog(frame);
```

1.3.3 Selectors de colors

De la mateixa manera que hi ha una classe predissenyada que permet obrir un diàleg per obrir o tancar un fitxer, Swing també en proporciona una que permet triar entre una paleta de colors: la classe `JColorChooser`. La figura 1.23 en mostra l'aspecte.

FIGURA 1.23. Un panell selector de colors.



Com en el cas del selector de fitxers, per poder fer-ne ús, primer cal crear una instància, que no es mostra en pantalla fins a cridar el mètode `public Color showDialog (Component parent, String titol, Color color)` que mostra un panell per seleccionar un color, amb un títol donat a la seva capçalera i un color per defecte preseleccionat.

El component pare es queda bloquejat fins que es dóna una resposta al selector de color. Aquesta crida retorna `nul` si es cancel·la la selecció, o una instància de la `classe java.awt.Color` amb el color seleccionat. Aquesta classe representa un color a partir d'una combinació de valors vermell-verd-blau (RGB, *red-green-blue*) i s'utilitza en totes les classes gràfiques sempre que cal definir algun color.

A fi de facilitar la feina del desenvolupador, la classe Color defineix un seguit de constants que es poden usar en qualsevol situació:

- `Color.BLACK`, negre.
- `Color.BLUE`, blau.
- `Color.WHITE`, blanc.
- `Color.GREEN`, verd.
- `Color.DARK_GRAY`, gris fosc.
- etc.

A continuació es mostra un exemple de crida de codi per mostrar un selector de colors:

```
1 JFrame frame = new JFrame();
2 ...
3 JColorChooser chooser = new JColorChooser();
```

```
Color res = chooser.showDialog(frame, "Tria el color",
          Color.WHITE);
JLabel label = new JLabel("Un text amb fons de colors");
if (null != res) {
  label.setBackground(res);
}
```

1.3.4 Classes basades en models

Tots els components de Swing disposen d'un ampli ventall de mètodes que permeten modificar directament el seu aspecte d'acord amb els designis del desenvolupador: el text que mostren, el color, etc. Tot i així, hi ha un conjunt de controls que integren directament el patró MVC en el seu codi, ja que es vinculen a un Model i per variar-ne l'aspecte cal interactuar amb aquest Model, en lloc de cridar mètodes directament sobre el control. Per aquest motiu s'anomenen **classes basades en models**. Els exponents principals d'aquest conjunt són les classes `JList`, `JTable` i `JTree`.

> Per modificar l'aspecte d'una **classe basada en model**, cal assignar un Model al seu constructor, o mitjançant el mètode `setModel`, i interactuar amb aquest Model. Cada una defineix la pròpia classe a usar com a Model i quins mètodes s'hi poden cridar.

Una particularitat dels Models d'aquestes classes és que poden emmagatzemar *Listeners*, de manera que és possible avisar el motor gràfic del Java sempre que hi ha alguna modificació. Quan això passa, aquest crida automàticament els mètodes consultors necessaris per veure quina ha estat la modificació i actualitzar els valors presentats per pantalla.

JList

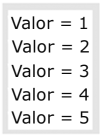
La classe `JList` correspon a una llista d'elements, de la qual se'n pot seleccionar un o més. Si es mira la documentació, es pot apreciar que no té cap mètode amb el qual afegir aquests elements, ja que cal usar el seu Model per fer-ho. El Model definit per interactuar-hi és la *interface* ListModel. Per tant, és responsabilitat del desenvolupador crear una implementació, d'acord amb les seves necessitats, que és la que realment s'instancia i assigna a la JList.

Aquesta defineix els mètodes públics següents:

- **Object getElementAt(int index)**. Obté l'element emmagatzemat en la posició *i* de la llista. Fixeu-vos que es pot desar qualsevol tipus d'objecte. El text que es visualitza en la JList és el resultat de cridar el mètode toString sobre l'objecte retornat.

- **int getSize()**. Obté el nombre d'elements emmagatzemats en la llista.
- **void removeListDataListener(ListDataListener l)**. Elimina un *Listener* associat.
- **void addListDataListener(ListDataListener l)**. Els mètodes associats a la gestió dels *Listeners* registrats.

Com es pot veure, la *interface* només defineix els mètodes consultors de lectura, que són els mínims indispensables per al motor gràfic del Java a fi de poder esbrinar quina informació ha de representar en pantalla. El desenvolupador disposa de llibertat absoluta per afegir tots els mètodes addicionals que consideri necessaris a la seva implementació (normalment, accessoris d'escriptura per poder desar o modificar les dades de la llista).



Exemple d'una JList amb un Model associat que retorna cinc cadenes de text diferents (Valor = 1...5). Per tant, es visualitzen cinc elements.

Afortunadament, per estalviar feina al desenvolupador, Swing proporciona una parell de subclasses en què ja es proporciona una implementació per defecte dels mètodes definits en la *interface* `ListModel`. D'una banda, hi ha la classe abstracta `AbstractListModel`, en què ja estan codificats tots els aspectes vinculats al registre de *Listeners,* de manera que si s'hereta directament d'ella només cal implementar els mètodes `getElementAt` i `getSize` i cridar el mètode `fireContentsChanged` sempre que les dades del model es modifiquin. En fer-ho, el motor gràfic del Java ja s'encarrega d'actualitzar el component gràfic d'acord amb els valors emmagatzemats en el Model. D'altra banda, la classe `DefaultListModel` encara va més enllà i implementa absolutament tota la *interface*, de manera que en proporciona un amb un comportament per defecte igual al d'un vector. Normalment, n'hi ha prou amb aquesta *interface* per a la majoria de casos.

A continuació es mostra un exemple de codi per generar una JList correctament a partir d'un Model, en aquest cas, partint de la classe `AbstractListModel`. Sempre que es crida el mètode incrementar, l'aspecte del component varia en pantalla.

```
1  ...
2  private class MyListModel extends AbstractListModel {
3    int[] valors = {1,2,3,4,5};
4
5    public Object getElementAt(int i) {
6      return "Valor = " + valors[i];
7    }
8
9    public int getSize() {
10     return valors.length;
11   }
12
13   public void incrementar() {
14     for (int i=0;i < valors.length; i++) valors[i]++;
15     //Avisar que s'ha modificat el model
16     fireContentsChanged(this, 0, valors.length);
17  }
18 }
19 ...
20 JList list = new JList(new MyListModel);
```

JTable

La classe `JTable` mostra una taula com la que es podria generar en un full de càlcul. El seu Model associat és la *interface* `TableModel` i la seva filosofia és exactament igual que el cas JList, si bé, evidentment, en aquest cas els mètodes definits en el Model són diferents. Aquests són els següents:

- **int getRowCount()**. Retorna el nombre de files de la taula.
- **int getColumnCount()**. Retorna el nombre de columnes de la taula.
- **Object getValueAt(int row, int column)**. Retorna l'element emmagatzemant en la cel·la de la fila `row` i columna `column`. Novament, es visualitza el resultat de cridar el mètode `toString` sobre l'objecte retornat.

La figura 1.24 mostra un exemple d'una JTable amb un model associat que indica que es compon de tres files i quatre columnes. Per cada cel.la, el model retorna una cadena de text del tipus "Valor = xx".

FIGURA 1.24. Exemple de taula amb model associat.

Valor = 1	Valor = 2	Valor = 3	Valor = 4
Valor = 5	Valor = 6	Valor = 7	Valor = 8
Valor = 9	Valor = 10	Valor = 11	Valor = 12

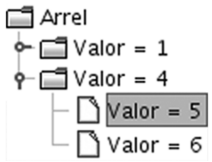
En aquest cas, també hi ha implementacions parcials, definides juntament amb el Model dins el paquet `javax.swing.table`. Aquestes són la classe `AbstractTableModel`, que proporciona una implementació parcial, i la classe `DefaultTableModel`, que en proporciona una de completa amb un comportament per defecte. Tot seguit es mostra un exemple basat en el primer cas per la taula de la figura 1.24.

```
1  import javax.swing.table.*;
2  ...
3  private class MyTableModel extends AbstractTableModel {
4    //Per una taula de 3 files * 4 columnes
5    int[][] valors =
6    {{1,2,3,4}, {5,6,7,8}, {9,10,11,12}};
7
8    public int getRowCount() {
9      return valors.length;
10   }
11   public int getColumnCount(){
12     return valors[0].length;
13   }
14   public Object getValueAt(int row, int column){
15     return "Valor = " + valors[row][column];
16   }
17   public void incrementar() {
18     for (int i=0;i < valors.length; i++)
19       for (int j=0;j < valors[i].length; j++)
20         valors[i][j]++;
21     //Avisar que s'ha modificat el model
22     fireTableDataChanged();
23   }
24 }
25 ...
26 JTable table = new JTable(new MyTableModel);
```

JTree

La classe `JTree` mostra un arbre desplegable d'elements, de manera que en pitjar sobre un es mostren o s'oculten tots els seus fills de manera commutada. El Model que utilitza és la *interface* `TreeModel`, definida en el paquet javax.swing.tree. Aquest Model és més complex que els anteriors i utilitza un seguit de classes auxiliars (`TreeNode` i les seves subclasses), per la qual cosa no es llistaran tots els seus mètodes. És molt recomanable usar la implementació per defecte que ofereix el Java, que en aquest cas només és una classe: `DefaultTreeModel`. Novament, els elements es visualitzen d'acord amb el valor retornat per la crida del mètode `toString`.

FIGURA 1.25



Exemple de JTree amb un model associat compost per una jerarquia de nodes. Cada node té associada una cadena de text, que és la que es visualitza en la interface gràfica.

A títol il·lustratiu, el codi següent mostra el codi que generaria el JTree i el model associat a la figura 1.25.

```
1  import javax.swing.tree.*;
2  ...
3  private class MyTreeModel extends DefaultTreeModel {
4    int[] valors = {1,2,3,4,5,6};
5    DefaultMutableTreeNode[] nodes = new DefaultMutableTreeNode[6];
6    public MyTreeModel() {
7      super(new DefaultMutableTreeNode("Arrel"));
8      DefaultMutableTreeNode arrel = (DefaultMutableTreeNode)getRoot();
9      for (int i=0; i< nodes.length; i++)
10       nodes[i] = new DefaultMutableTreeNode("Valor = " + valors[i]);
11     arrel.add(nodes[0]);
12     nodes[0].add(nodes[1]);
13     nodes[0].add(nodes[2]);
14     arrel.add(nodes[3]);
15     nodes[3].add(nodes[4]);
16     nodes[3].add(nodes[5]);
17   }
18   public void incrementar() {
19     for (int i=0; i<nodes.length; i++) {
20       valors[i]++;
21       nodes[i].setUserObject("Valor = " + valors[i]);
22       //Avisar que s'ha modificat el model
23       this.nodeChanged(nodes[i]);
24     }
25   }
26 }
27 ...
28 JTree tree = new JTree(new MyTreeModel());
```

1.3.5 Dibuix lliure

Hi ha situacions en què el desenvolupador vol poder dibuixar lliurement sobre una part de la pantalla, normalment sobre un panell, i modificar-ne directament l'aspecte. Per a això, la biblioteca gràfica del Java proporciona una API en què s'estableixen dos mecanismes diferents, però molt relacionats, per controlar com es dibuixen els components a pantalla, un associat als d'AWT i un altre als de Swing. De totes maneres, en ser una API d'una certa extensió i amb moltes funcionalitats, es limita a donar una visió general per al segon cas, però més que suficient per conèixer els aspectes bàsics amb els quals es pot començar a treballar.

API són les inicials d'Application Programming Interface, interface de programació d'aplicacions.

El primer pas per entendre com es pot modificar l'aspecte dels components Swing és veure quin és el mecanisme que en gestiona la visualització correcta en pantalla. El més important d'aquest mecanisme és que quan es detecta que el dibuix d'un component qualsevol en pantalla s'ha d'actualitzar, cal cridar-ne el mètode `repaint`, definit a `java.awt.Component` i heretat per tots els elements gràfics Aquest és el responsable de proporcionar la imatge que cal mostrar en pantalla i se sobreescriu per a cada cas. Hi ha dos motius pels quals cal actualitzar la visualització d'un component:

- **Actualització iniciada pel sistema.** Es tracta d'un cas iniciat automàticament pel motor gràfic del Java quan detecta que la regió de la pantalla que ocupa el component ha variat i, per tant, ha de ser redibuixada. El desenvolupador no ha de fer res, per exemple, en redimensionar una finestra, de manera que el component deixa de ser visible o ho torna a ser.

- **Actualització iniciada per l'aplicació.** Es tracta del cas en què el desenvolupador força l'actualització explícitament, cridant `repaint` al seu codi, ja que el Model associat al component ha variat i considera que ha de canviar la manera com es visualitzen les dades. Per exemple, el cas d'una gràfica en què en un moment donat varien els valors que cal mostrar.

En els components AWT, els mètodes a sobreescriure són "paint" i "update".

El mètode `repaint` realitza internament un conjunt de tasques diferents per garantir que el component es visualitzarà correctament. Per exemple, també ha de gestionar l'actualització d'altres components que conté, entre d'altres coses. El conjunt de tasques concretes varia segons si el component és de la biblioteca AWT o Swing. En el cas de Swing, el punt clau d'aquestes tasques és la crida a un nou mètode, `paintComponent`, que és el que realment defineix com s'ha de dibuixar el component. Aquest rep com a paràmetre un objecte de tipus `java.awt.Graphics`, a partir del qual es pot dibuixar directament sobre el component.

Per modificar directament l'aspecte gràfic d'un component Swing, cal sobreescriure'n el mètode **paintComponent** , per així poder manipular-ne l'objecte `Graphics` associat.

Per assolir un component amb un aspecte personalitzat, el desenvolupador ha de generar una nova subclasse pròpia a partir del component original en què se sobreescrigui el mètode `paintComponent`. Per fer dibuixos lliurement en una àrea de la pantalla, cal crear una subclasse de `JPanel`.

La classe `Graphics` defineix un ampli ventall de mètodes per dibuixar lliurament sobre el panell, per la qual cosa, un cop obtingut l'objecte d'aquest tipus associat al component, tot es limita ja a cercar dins la documentació el mètode més adequat per a la tasca a realitzar.

Càrrega d'imatges

La càrrega d'imatges sobre un panell es realitza mitjançant alguns dels mètodes `drawImage` ,oferts per la classe `Graphics`. Per gestionar imatges, la biblioteca AWT ofereix una classe auxiliar, Toolkit, que permet obtenir objectes Image a partir del nom del fitxer o un bloc de dades.

```
1  public class MyPanel extends JPanel {
2    //Classe auxiliar per gestionar imatges
3    Toolkit t = new Toolkit.getDefaultToolkit();
4    Image imatge;
5
6    public MyPanel(String nom) {
7      imatge = t.getImage(nom);
8    }
9
10   public void paintComponent(Graphics g) {
11     //Mantenim el comportament original
12     super.paintComponent(g);
13     //Dibuixem la imatge
14     g.drawImage(imatge,0,0,this);
15   }
16 }
```

Dibuixar figures geomètriques

De la mateixa manera que es poden carregar imatges, la classe `Graphics` disposa d'un mètode diferent per a pràcticament qualsevol tipus de figura, en dues o tres dimensions. A continuació es mostren alguns dels més significatius:

- **void drawLine(int x1, int y1, int x2, int y2)**. Dibuixa una línia recta des de les coordenades (x1,y1) fins a les coordenades (x2,y2). Es considera la coordenada (0,0) la cantonada superior esquerra del component.
- **void drawArc(int x, int y, int width, int height, int startAngle, int arcAngle)**. Dibuixa una línia corba, en forma d'arc, partint de les coordenades (x,y). La seva alçària i amplada, en píxels, són widthi height de manera que l'àrea coberta és igual a un rectangle. L'arc d'inclinació parteix destartAngle graus i es manté durant arcAngle graus.
- **drawRect(int x, int y, int width, int height)**. Dibuixa un rectangle de coordenades quadrades, partint de les coordenades (x,y), amb una amplada i alçària de width i heightpíxels.

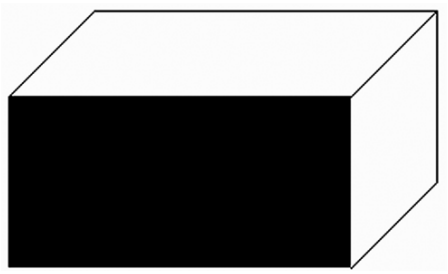
- **drawRoundRect(int x, int y, int width, int height, int arcWidth, int arcHeight)**. Dibuixa un rectangle de cantonades arrodonides, partint de les coordenades (x,y), amb una amplada i alçària de width i heightpíxels. El grau d'arrodoniment de les cantonades està determinat pels valors arcWidthi arcHeightde manera similar a com es defineixen les línies corbes en el mètodedrawArc.

- **drawOval(int x, int y, int width, int height)**. Dibuixa un oval amb centre en les coordenades (x,y) i amb una amplada i alçària de width i heightpíxels. Si aquest dos valors són iguals, el resultat és un cercle.

Tots els mètodes descrits dibuixen figures "buides", de manera que el seu interior és transparent i es veu el color del fons del component sobre el qual s'han dibuixat. Per crear figures opaques, per a cada mètode hi ha una versió anomenada `fillXXX`, en lloc de `drawXXX` que realitza aquesta tasca (`fillOval`, `fillRect`,etc.).

Per establir el color amb què es vol dibuixar, cal cridar prèviament el mètode `setColor`. Cada cop que es vol usar un color diferent cal tornar a cridar aquest mètode amb el nou color.

FIGURA 1.26. Resultat del codi d'exemple.



Tot seguit es mostra un exemple de dibuix com el de la figura 1.26 usant diverses crides al mètode `drawLine`:

```
1  public class MyPanel extends JPanel {
2
3    public void paintComponent(Graphics g) {
4      super.paintComponent(g);
5      g.setColor(Color.BLACK);
6      g.fillRect(100, 100, 200, 100);
7      g.drawLine(100,100,150,50);
8      g.drawLine(150,50,350,50);
9      g.drawLine(300,100,350,50);
10     g.drawLine(350,50,350,150);
11     g.drawLine(300,200,350,250);
12   }
13   //Sobreescrivint getPreferredSize es pot definir
14   //una mida per defecte.
15   public Dimension getPreferredSize() {
16     return new Dimension(400, 400);
17   }
18 }
```

Dibuixar cadenes de caràcters

Per escriure lletres directament en unes coordenades concretes del component hi ha el mètode `drawString`.

```
1  public class MyPanel extends JPanel {
2
3    Color colors[] = { Color.BLACK, Color.BLUE,
4      Color.RED, Color.GREEN, Color.YELLOW};
5    public void paintComponent(Graphics g) {
6      super.paintComponent(g);
7      for (int i=0;i<5;i+=50) {
8        g.setColor(colors[i]);
9        g.drawString( "Hola Java!", i+50, i+50);
10     }
11   }
12
13   //Sobreescrivint getPreferredSize es pot definir
14   //una mida per defecte.
15   public Dimension getPreferredSize() {
16     return new Dimension(400, 400);
17   }
18 }
```

1.3.6 Applets

El codi HTML

És la sintaxi usada per representar una pàgina web, de manera que pugui ser interpretada pel navegador i visualitzada per l'usuari.

El llenguatge Java ofereix alternatives a les aplicacions d'escriptori a l'hora de generar *interfaces* gràfiques. De fet, el tipus d'aplicació que va impulsar el llenguatge Java com un entorn ideal per generar aplicacions gràfiques a Internet van ser els anomenats *applets*.

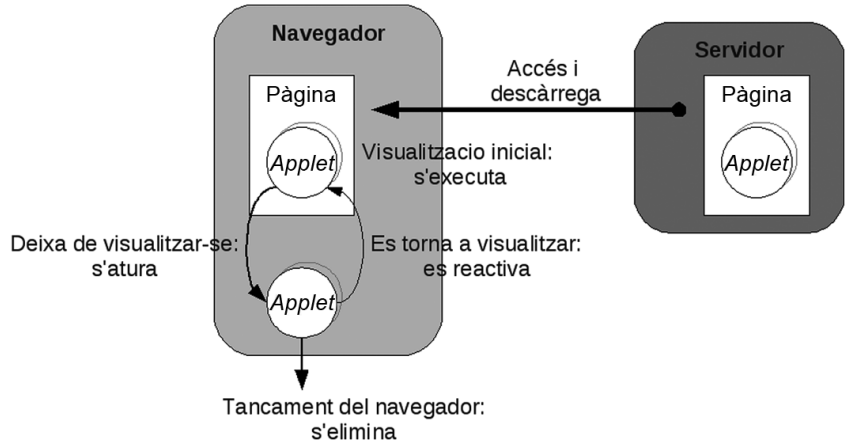
Provant applets

Per tant, si es vol provar un *applet* quan s'està desenvolupant, cal tancar totalment el navegador cada cop que es vulgui executar una nova versió. No n'hi ha prou sortint de la pàgina que el conté i tornant a entrar-hi.

Un **applet** és una aplicació Java que s'executa dins una altra aplicació, en lloc de directament sota el sistema operatiu. Normalment, s'executen dins els navegadors web, i permeten crear pàgines dinàmiques o interactives d'una complexitat que el codi HTML per si mateix no permet.

La figura 1.27 esquematitza el funcionament d'un *applet* i quin és el seu cicle de vida quan un usuari accedeix a la pàgina web que el conté des del seu navegador. L'*applet* es troba instal·lat en el servidor web i es descarrega dins el navegador quan l'usuari accedeix a la pàgina web que el conté. Un cop descarregat, el navegador l'emmagatzema en la memòria de l'ordinador i immediatament l'executa. L'aspecte més destacat d'aquest cicle de vida és que la descàrrega de l'*applet* només es produeix la primera vegada que s'accedeix a la seva pàgina. Al llarg de l'execució del navegador, l'*applet* roman en la memòria independentment que l'usuari accedeixi a altres pàgines web diferents i deixi de visualitzar-lo. Només s'elimina realment de la memòria quan es tanca el navegador. La propera vegada que l'usuari posi en marxa el navegador i torni a accedir a la pàgina de l'*applet*, tot el procés es repetirà des del principi.

FIGURA 1.27. Cicle de vida d'un applet.



L'avantatge principal de generar aplicacions en forma d'*applet* és la facilitat de desplegament, en contrast amb una aplicació estàndard d'escriptori. No cal copiar el fitxers de l'aplicació en cada ordinador que l'ha d'usar, només cal copiar-los en el servidor web i garantir que en cada client hi ha algun navegador instal·lat. Avui en dia, es pot garantir que això és cert per a pràcticament qualsevol ordinador. En cas de fer actualitzacions, novament, només cal canviar les dades en el servidor web. Evidentment, el seu desavantatge principal és que cada client ha de descarregar l'*applet* del servidor quan el vol executar, procés més lent que si la instal·lació és local. Si el servidor no funciona per algun motiu, ningú no pot executar l'aplicació.

Adobe Flash

O simplement Flash, és una plataforma multimèdia per a la creació d'animacions i aplicacions gràfiques riques, executables sobre navegadors. Aquestes solen prendre la forma de fitxers amb extensió .swf.

Cal dir que, a mesura que els navegadors han evolucionat i l'amplada de banda disponible per a les connexions a Internet ha crescut, els *applets* Java han vist disminuir parcialment la popularitat en favor d'altres tecnologies més modernes, amb capacitats gràfiques i multimèdia superiors, com pot ser la tecnologia Flash. Tot i així, els *applets* encara tenen l'avantatge de permetre a un desenvolupador del Java traduir aplicacions d'escriptori a executables sobre navegadors sense haver d'aprendre un nou llenguatge de programació absolutament diferent. Un altre avantatge important és que, en l'actualitat, pràcticament el 100% dels navegadors moderns disposen de la màquina virtual del Java preinstal·lada, cosa que no és el cas per a altres tecnologies, que normalment requereixen la instal·lació de complements addicionals (connectors o *plug-ins*) per part de l'usuari. Per tant, encara no ha arribat l'hora d'obviar el desenvolupament d'*applets*.

La classe JApplet

La classe que representa el contenidor d'alt nivell dels *applets* dins la biblioteca Swing és `JApplet`. Aquesta pertany a una branca de la jerarquia de classes diferent de la resta de components Swing explicats fins ara (`JFrame`, `JPanel`, etc.), per la qual cosa no comparteixen totes les mateixes funcionalitats exactament.

Per generar un *applet*, cal crear una classe que hereti de `JApplet` i sobreescriure, d'acord amb les tasques que es vol que realitzi, els mètodes associats al seu cicle de vida:

- **public void init**(). S'executa la primera vegada que l'*applet* es descarrega en el navegador i es posa en marxa. És l'equivalent al mètode main d'una aplicació Java d'escriptori, per la qual cosa és obligatori sobreescriure'l si es vol arribar a algun resultat.
- **public void stop**(). S'executa cada cop que l'*applet* es deixa de visualitzar en pantalla, per exemple, perquè el navegador es minimitza o surt de la seva pàgina i es visita una pàgina diferent. Resulta útil amb vista a aturar l'execució d'elements associats que impliquen una certa càrrega, però que no té sentit que estiguin en marxa si l'*applet* no s'està visualitzant com, per exemple, animacions o qualsevol element dinàmic que canviï constantment. No és imprescindible sobreescriure'l si no hi ha cap element d'aquest tipus.
- **public void start**(). S'executa cada cop que l'*applet* es visualitza en pantalla. Per tant, es crida sempre immediatament després del mètode init, però també sempre que l'usuari retorna a la pàgina que conté l'*applet*, després que per algun motiu es deixa de visualitzar. Només sol tenir sentit sobreescriure'l si també s'ha sobreescrit el mètode stop.
- **public void destroy**(). S'executa en tancar el navegador de manera ordenada, moment en què l'*applet* es descarrega definitivament de la memòria de l'ordinador, per la qual cosa cal alliberar qualsevol recurs important en el seu codi. Només se sol sobreescriure en *applets* complexos.

Els *applets* no es poden obrir directament amb el navegador, sempre es troben encastats dins una pàgina normal en codi HTML, a la qual s'accedeix per mitjà del navegador web. Per incloure un *applet* dins una pàgina web cal usar l'etiqueta o *tag* HTML:

```
<APPLET CODE = "NomApplet" WIDTH = "500" HEIGHT = "500">
  <PARAM NAME=nom1 VALUE=valor1 />
  <PARAM NAME=nom2 VALUE=valor2 />
  ...
</APPLET>
```

Els indicadors `WIDTH` i `HEIGHT` permeten indicar quines són les dimensions que ha d'ocupar dins la finestra del navegador, l'amplada i alçària respectivament. Opcionalment, és possible incloure un conjunt d'etiquetes `PARAM` per definir parells nom-valor que indiquen paràmetres amb vista a l'execució de l'*applet*, de manera similar als arguments associats al mètode `main` d'una aplicació d'escriptori. Els valors d'aquests paràmetres es poden recuperar mitjançant el mètode definit en la mateixa classe `JApplet`:

```
public String getParameter(String nomParametre)
```

A part d'aquestes particularitats, el comportament de la classe JApplet és pràcticament igual que un JFrame per a generar la *interface* gràfica. Mitjançant el mètode `getContentPane()` se'n pot obtenir el panell de contingut, i per aquest es pot afegir qualsevol contenidor o control Swing cridant el mètode `add`.

Timer i TimerTask

Mitjançant aquestes classes del paquet `java.util` permeten executar tasques cada cert temps, alhora que s'executa el codi del programa principal.

Tot seguit es mostra un exemple de codi per generar un *applet* dinàmic, en què un comptador avança cada segon a partir del valor establert en un paràmetre.

```
public class Test extends JApplet {
  private JLabel display = null;
  java.util.Timer timer = null;
  private int valor = 0;

  //Classe que defineix una tasca de temporitzador
  private class Task extends java.util.TimerTask {
    public void run() {
      valor++;
      display.setText("Comptador :" + valor);
      //Reprogramar el temporitzador de nou
      timer.schedule(new Task(),1000);
    }
  }

  private int getValor() {
    try {
      String valStr = getParameter("INICI");
      return Integer.parseInt(valStr);
    } catch (Exception ex) {
      return 0;
    }
  }

  public void init() {
    valor = getValor();
    JPanel panell = (JPanel)getContentPane();
    display = new JLabel("Comptador :" + valor);
    display.setHorizontalAlignment(SwingConstants.CENTER);
    display.setOpaque(true);
    panell.add(display);
    //Es genera un temporitzador
    timer = new java.util.Timer();
  }

  public void start() {
    //Activar el temporitzador a 1 segon
    timer.schedule(new Task(),1000);
  }
}
```

La capsa de sorra

Tot i que, a grans trets, la programació del codi d'un *applet* és molt semblant a la d'una aplicació d'escriptori, sí que hi ha un aspecte important que tot desenvolupador ha de tenir ben present. Els *applets* sempre s'executen dins un navegador que s'anomena *capsa de sorra* (*sandbox*).

> Una **capsa de sorra** (sandbox) és un entorn d'execució segur en què no es permet que les aplicacions realitzin un conjunt d'operacions considerades inherentment insegures, que poden deixar la porta oberta a malifetes per part de codi creat amb males intencions.

Malware

S'anomena així tot tipus de programari maliciós amb l'únic objectiu d'executar-se sense el permís de l'usuari amb intencions funestes.

En el cas específic dels *applets*, es tracta amb un tipus d'aplicació amb totes les funcionalitats de les biblioteques completes del Java i que s'executa automàticament en el navegador tan bon punt l'usuari es connecta a la pàgina en què s'ha inclòs. Això implica una porta d'entrada per a tota mena de programa maliciós o *malware*. Per exemple, suposem que es genera un *applet* que formata el disc

de l'ordinador, o cerca tots els fitxers amb informació personal i els envia a una adreça de correu electrònic, tot sota l'aparença d'una aplicació lícita (per exemple, unes divertides animacions amb coloraines). En un entorn d'execució sense cap límit, tan bon punt qualsevol usuari es connecti a la pàgina amb aquest *applet* ja es pot dur a terme la malifeta. Quan s'adoni del que ha passat, ja serà massa tard.

Runtime és una classe de les biblioteques del Java que permet executar altres programes usant el seu mètode .

Per aquest motiu, hi ha algunes tasques que un *applet* no pot realitzar per defecte. En fer-les, es llança una excepció des del mètode que ho ha intentat. Aquestes tasques són, a grans trets:

- Qualsevol mena d'interacció amb el sistema de fitxers, tant de lectura com d'escriptura.
- Qualsevol mena d'interacció per xarxa amb una màquina que no sigui la mateixa en què l'*applet*està instal·lat (el servidor web).
- Consultar o modificar qualsevol propietat del sistema operatiu en què s'executa l'*applet*.
- Executar altres programes.
- Instanciar altres contenidors d'alt nivell.
- Sortir directament de l'*applet* amb la crida `System.exit()`.

Per signar un *applet*, s'usa l'eina "jarsigner", distribuïda amb l'entorn estàndard del Java.

Per tant, en desenvolupar un *applet* cal tenir en compte que hi ha aquestes limitacions dins les tasques que pot realitzar. Tot i així, hi ha mecanismes per generar *applets*que poden saltar-se algunes d'aquestes restriccions, com per exemple el que s'anomena *signar l'applet.*

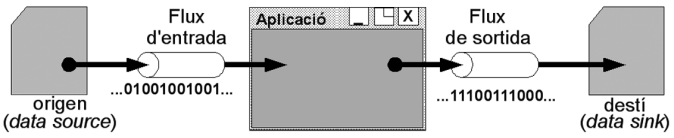
2. Fluxos i fitxers

Per poder dur a terme operacions d'entrada i de sortida, de manera que sigui possible llegir o escriure dades, el Java disposa d'un mecanisme unificat, independent de l'origen o la destinació de les dades: els fluxos (*stream*). Aquest sistema no és exclusiu del Java, sinó que està suportat en altres llenguatges també, perquè es tracta en realitat d'una funcionalitat dels sistemes operatius. Aquest apartat se centrarà en l'ús de fluxos per al cas de l'accés a dades dins fitxers, en ser el més senzill i intuïtiu. Ara bé, cal tenir present que el mecanisme de fluxos no està vinculat exclusivament a interaccions amb el sistema de fitxers, sinó que és extrapolable a qualsevol operació en què s'efectuen operacions de lectura o d'escriptura seqüencial de dades. Per exemple, operacions amb *buffers* de memòria o comunicacions en xarxa.

Un **flux** (*stream*) és el terme abstracte usat per referir-se al mecanisme que permet a un conjunt de dades **seqüencials** transmetre's des d'un origen de dades (*data source*) a una destinació de dades (*data sink*).

Des del punt de vista de l'aplicació, es poden generar dos tipus de fluxos: d'entrada i de sortida. Els fluxos d'entrada serveixen per llegir dades des d'un origen (per exemple, seria el cas de llegir un fitxer), per tal de ser processades, mentre que els de sortida són els responsables d'enviar les dades a una destinació (per a un fitxer, seria el cas d'escriure-hi). La figura 2.1 resumeix com una aplicació transmet o rep dades mitjançant fluxos.

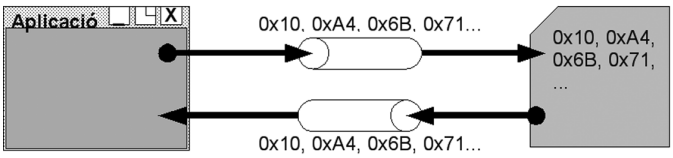
FIGURA 2.1. Fluxos d'entrada i de sortida



L'aspecte més important del funcionament dels fluxos és que les dades sempre es transmeten i es processen seqüencialment. En el cas dels fluxos de lectura, això vol dir que un cop s'ha llegit un conjunt de dades, ja no és possible tornar enrere per llegir-lo de nou. En el cas dels fluxos d'escriptura, aquest fet implica que les dades s'escriuen, en la destinació, exactament en el mateix ordre que es transmeten, no es poden fer salts. Aquest comportament seqüencial també fa que l'ordre en què es llegeixen les dades d'un origen sigui sempre exactament el mateix que l'ordre en què es van escriure en el seu moment. El primer *byte* que es llegeix és el primer que es va escriure, després el segon... Tot el sistema es comporta sempre com una estructura FIFO. La figura 2.2 mostra un esquema d'aquest fet.

FIFO són les inicials de first in first out (primer a entrar, primer a sortir).

FIGURA 2.2. Les dades s'escriuen i es llegeixen seqüencialment.



Un flux de dades és com un tub, on, en lloc d'aigua, es transmeten dades entre dos extrems. Un cop passa l'aigua, ja no es pot fer enrere.

Dins el Java, hi ha dos tipus de flux. D'una banda, hi ha els **fluxos orientats a dades**, que són aquells en què les dades que es transmeten són purament binàries, amb una interpretació totalment dependent de l'aplicació. D'altra banda, hi ha els **fluxos orientats a caràcter**, en què les dades processades sempre es poden interpretar com a text. Tot i que estrictament es pot considerar que els segons són més aviat un subconjunt dels primers, ja que en ordinador, en darrera instància, sempre s'acaba representant tot en cadenes de *bytes*, fer aquesta diferenciació val la pena per entendre algunes de les particularitats del sistema de fluxos que proporciona el Java. Un exemple d'aquestes particularitats és que mentre que els primers operen sempre amb el tipus primitiu *byte*, els segons ho fan amb el tipus `char`.

Totes les classes vinculades a l'entrada/sortida en Java es troben definides dins el *package* `java.io`. Dins aquest mateix paquet també es defineix el tipus d'excepció vinculada a errors sorgits durant el procés de lectura i escriptura de dades: `IOException`. Pràcticament tots els mètodes més importants poden generar aquesta excepció, per la qual cosa és imprescindible capturar-la. Complementant `IOException`, també hi ha definit un gran nombre de subclasses que aporten informació més concreta sobre l'error que ha tingut lloc.

2.1 Gestió de fitxers

Abans d'aprofundir en el funcionament dels fluxos de dades, val la pena veure com es gestiona el sistema de fitxers, ja que en la majoria de casos els fluxos s'utilitzaran per accedir-hi. Dins la biblioteca java.io, la classe que representa un fitxer a Java és `File`. Aquesta permet al desenvolupador manipular qualsevol aspecte vinculat al sistema de fitxers. Es pot usar tant per manipular fitxers de dades com directoris.

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

Si bé disposa de diversos constructors, el més típicament usat és:

```
1 public File (String ruta)
```

El format de la ruta

Cal tenir sempre present que el format que ha de tenir 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 Windows inicia les rutes per un nom d'unitat (C:, D:, etc.), mentre que l'Unix sempre usa una barra ("/"). A més a més, els diferents sistemes operatius usen diferents separadors dins les rutes. Per exemple, l'Unix usa la barra ("/") mentre que el Windows la contrabarra ("\").

Ruta Unix: /usr/bin

Ruta Windows: C:\Windows\System32

Per generar aplicacions portables a diferents sistemes, la classe File ofereix una constant que és recomanable usar per especificar separadors de ruta dins una cadena de text: File.separator. Aquesta sempre pren la forma adequada d'acord amb el sistema operatiu en què s'estigui executant l'aplicació en aquell moment.

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

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

Un altre aspecte molt important que també cal tenir sempre present és si el sistema operatiu distingeix entre majúscules i minúscules o no. El Java és totalment neutral en aquest aspecte, actuant tal com especifiqui el sistema operatiu.

L'element final de la ruta pot existir realment o no, però això no impedeix de cap manera poder instanciar File. Les seves instàncies es comporten 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 es s'hi escriuen o llegeixen dades, que realment s'accedeix al sistema de fitxers i es processa la informació. Si s'intenten llegir dades des d'una ruta que en realitat no existeix, es produeix un error, i es llança una FileNotFoundException.

La classe File ofereix tot un seguit de mètodes que permeten realitzar operacions amb la ruta especificada. Alguns dels més significatius per entendre'n les funcionalitats són:

- `public boolean exists()`. Indica si la ruta especificada realment existeix en el sistema de fitxers.
- `public boolean isFile()/isDirectory()`. Aquests dos mètodes serveixen per identificar si la ruta correspon a un fitxer, o bé a un directori.

Els mètodes següents només es poden cridar sobre rutes que especifiquen fitxers o, en cas contrari, no faran res:

- `public long length()`. Retorna la mida del fitxer.
- `public boolean createNewFile()`. Crea un nou fitxer buit en aquesta ruta, si encara no existeix. Retorna si l'operació ha tingut èxit.

Gestionant el sistema de fitxers

Per exemple, suposem que es vol crear un nou fitxer, sempre que aquest encara no existeixi. El codi que realitzaria aquesta acció seria:

```
File file = new File(ruta);
if (!file.exists())
 file.createNewFile();
```

El fitxer no es crea realment en el sistema de fitxers fins a executar el mètode createNewFile. Fins llavors, l'objecte referenciat per file només indica una ruta dins el sistema de fitxers amb la qual es pot operar, però no un fitxer real.

En contraposició, els mètodes següents només es poden cridar sobre rutes que especifiquen directoris:

- `public boolean mkdir()`. Crea el directori, si no encara existeix. Retorna si l'operació ha tingut èxit.

- `public String[] list()`. En retorna el contingut en forma d'*array* de cadenes de text.

- `public String[] list(FilenameFilter filter)`. Mitjançant el paràmetre addicional filter, és possible filtrar el resultat, de manera que només es retorna el conjunt de fitxers i directoris que compleixen certs criteris. `FilenameFilter`és una *interface*, per la qual cosa és responsabilitat del desenvolupador proporcionar la implementació adequada d'acord amb les condicions en les quals es vol llistar el contingut del directori. Només té un mètode:

- `boolean accept(File dir, String name)`. Cada cop que es crida el mètode list(), aquest crida internament accept per cada fitxer o directori contingut. El paràmetre dir indica el directori en què està ubicat el fitxer o directori processat, mentre que name n'indica el nom. Retorna cert o fals segons si es vol, o no, que sigui inclòs en la llista retornada per la crida al mètode list().

Llistant fitxers .png

Un exemple de com s'usa un FilenameFilter per llistar fitxers amb una extensió concreta dins un directori podria ser:

```
file.list( new FileNameFilter() {
 public boolean accept(File f, String name) {
  if (name.endsWith(".png"))
   return true;
  else
   return false;
 }
});
```

Fixeu-vos que en aquest exemple s'ha usat una classe anònima per definir el filtre. Atès que els filtres no se solen reusar molt dins el codi, és un altre cas en què val la pena usar classes anònimes.

2.2 Fluxos orientats a dades

El Java ofereix un sistema d'accés homogeni al mecanisme de fluxos orientats a dades mitjançat, per descomptat, una jerarquia de classes.

Les superclasses **InputStream** i **OutputStream** especifiquen els mètodes relatius al comportament comú a qualsevol flux, i cada subclasse s'encarrega llavors de sobreescriure'ls, o afegir-ne de nous, segons les seves particularitats. Tots els fluxos a Java hereten d'alguna d'aquestes dues classes.

El fet de que tots els fluxos a Java hereten alguna de les superclasses `InputStream` i `OutputStream` té sentit, ja que, per exemple, independentment del format de l'origen de les dades, sempre hi ha l'opció de llegir, però les tasques que cal fer internament per a això són totalment diferents segons l'origen de dades concret. No és el mateix llegir d'un fitxer que d'un *buffer* de memòria. Hi ha una subclasse per cada tipus d'origen o destinació de dades.

La classe `InputStream` ofereix els mètodes descrits a continuació per llegir dades des de l'origen mitjançant un flux d'entrada. Un aspecte a destacar és que en una operació de lectura mai no es pot garantir quants bytes es llegiran realment, independentment que es conegui per endavant el nombre de bytes disponibles a l'origen (i, per tant, *a priori*, es pugui suposar aquesta garantia). Sempre cal crear algoritmes que tinguin en compte el valor de retorn dels diferents mètodes:

- **int available()**. Retorna tots els bytes que hi ha en el flux pendents de ser llegits. En un fitxer, seria el nombre de bytes que ocupa i encara no s'han processat.
- **int read()**. Llegeix exactament un byte. Aquest mètode retorna un enter, ja que fa ús del valor de retorn -1 per indicar que ja no queden més dades per llegir en l'origen. Per obtenir realment el byte llegit cal fer un *cast* del valor retornat sobre una variable de tipus byte.
- **int read(byte[] b)**. Intenta llegir tants bytes com la longitud de l'*array* passat com a paràmetre, on els emmagatzema a partir de l'índex 0. Retorna el nombre de bytes llegits realment. Cal tenir molt present que si s'ha llegit un nombre de bytes N, inferior a `b.length`, les dades emmagatzemades entre `N` i `b.length - 1` no són vàlides, ja que no corresponen a la lectura. Retorna -1 si no queden més dades per llegir en l'origen.
- **int read (byte[] b, int offset, int len)**. Intenta llegir len bytes, que emmagatzema dins de l'*array* `b` a partir de l'índex indicat pel valor `offset`. Com en el cas anterior, retorna el nombre real de bytes llegits i -1 si no queden més dades per llegir en l'origen.

Al mateix temps, la classe `OutputStream` ofereix els mètodes complementaris per escriure dades cap al destinació mitjançant un flux de sortida:

- **void write(int b)**. Escriu exactament un byte.
- **void write(byte[] b)**. Escriu tots els bytes emmagatzemats a `b`, de manera ordenada des de l'índex 0 a `b.length - 1`.

Flush

La traducció literal de "flush" és "tirar de la cadena". Una manera molt explícita de dir que es fa net al flux, i una nova referència a l'analogia entre un flux i un tub.

- **void write (byte[] b, int offset, int len)**. Escriu tots els bytes emmagatzemats a `b`, de manera ordenada des de l'índex `offset` a `offset + length - 1`.

Quan les operacions de lectura o escriptura sobre un flux han finalitzat, és imprescindible tancar-lo. En fer-ho, s'informa el sistema operatiu que ja no s'hi vol realitzar cap operació més i que pot alliberar tot un seguit de recursos que li ha calgut reservar prèviament per gestionar el flux. En el cas dels fluxos de sortida, tancar-lo també serveix per forçar l'escriptura real de les dades cap a la destinació, el que s'anomena **fer un flush**.

Per tancar qualsevol flux de dades es disposa del mètode `close()`.

En els sistemes operatius moderns, les escriptures són asíncrones. Això vol dir que no es pot garantir que en el mateix instant en què es fa un `write`, les dades realment s'hagin enviat a la destinació. Hi pot haver un retard, més o menys llarg. Només en tancar un flux es pot garantir que absolutament qualsevol dada escrita ja es troba realment a la destinació. Això implica que es pot donar el cas que durant el procés d'escriptura en un fitxer, immediatament després de retornar d'una crida a un mètode `write` s'obri el fitxer per veure'n el contingut (per exemple, amb un editor), però tot i així, les dades encara no hi siguin.

2.2.1 Origen i destinació en fitxers

Dins la jerarquia de fluxos orientats a dades, les classes responsables de crear fluxos vinculats a fitxers, aquestes són les classes `FileInputStream` (per lectura, origen) i `FileOutputStream` (per escriptura, destinació). Aquestes dues classes no afegeixen gaires mètodes addicionals respecte als definits per InputStream i OutputStream. Ambdues disposen de constructors que tenen com a paràmetre d'entrada o bé una instància de File, o directament una cadena de text amb la ruta del fitxer:

```
1 FileInputStream(File fitxer)
2 FileInputStream(String ruta)
3 FileOutputStream(File fitxer)
4 FileOutputStream(String ruta)
5 FileOutputStream(File fitxer, boolean append)
6 FileOutputStream(String ruta, boolean append)
```

Sempre que es genera un flux de sortida sobre un fitxer ja existent, aquest se sobreescriu, de manera que es perden absolutament totes les dades emmagatzemades anteriorment. L'única excepció d'aquest comportament són els constructors amb el paràmetre append. Aquest permet indicar, si es crida amb el valor true, que es volen concatenar les dades tot just a partir del final del fitxer actual.

Còpia d'un fitxer

Com exemple del funcionament dels fluxos orientats a dades vinculats a fitxers, a continuació es mostra el fragment de codi que realitzaria una còpia del fitxer ubicat a ruta, escrivint-lo a novaRuta exactament igual. En l'exemple, les dades es llegeixen en blocs de 100 bytes consecutius, si bé cal tenir molt present que mai es pot donar per garantit el nombre de bytes llegits realment en una crida del mètode read. Per aquest motiu, és

imprescindible controlar que només s'escrigui a la destinació exactament el mateix nombre de bytes que s'ha llegit.

```
InputStream in = new FileInputStream(ruta);
OutputStream out = new FileOutputStream(novaRuta);
copiaDades(in, out);
...
public void copiaDades(InputStream in, OutputStream out) {
  try {
    byte[] dades = new byte[100];
    int llegits = 0;
    while (-1 != (llegits = in.read(dades)))
     out.write(dades,0,llegits);
    out.close();
    in.close();
  } catch (IOException) { ... }
}
```

IOException

En operar amb fluxos, és imprescindible capturar les possibles excepcions en el procés d'entrada/sortida.

2.2.2 Origen i destinació en buffers de memòria

Un altre parell de classes molt útils quan es processen dades mitjançant fluxos són les relatives a orígens i destinacions de dades vinculats a *buffers* de memòria dinàmics: `ByteArrayInputStream` i `ByteArrayOutputStream`.

En el cas del flux d'entrada, permet llegir dades des d'un *array* de bytes (`byte[]`) seqüencialment, en lloc d'haver d'accedir per índex. Per aquest motiu, en el seu constructor cal indicar quin és l'*array* origen de les dades:

- **ByteArrayInputStream(byte[] buf)**. En el cas del flux de sortida, les dades escrites s'emmagatzemen en un bloc indeterminat de la memòria del programa, que augmenta la mida dinàmicament a mesura que s'escriuen noves dades. No hi ha cap límit excepte la memòria física de l'ordinador, si bé es recomana no usar-los per a quantitats molt grans de dades. Un cop ha finalitzat l'escriptura, és possible obtenir totes les dades emmagatzemades mitjançant el mètode específic:
- **byte[] toByteArray()**. En la posició 0 de l'*array* hi ha el primer byte escrit en el flux, i així successivament fins a trobar el darrer escrit en la posició `length - 1`.

Canviant la destinació o l'origen de les dades

Per observar els avantatges que ofereix l'abstracció mitjançant fluxos, suposem que es vol canviar la destinació de les dades del tros de codi que serveix per copiar un fitxer i, en lloc d'un fitxer, es vol escriure sobre un *buffer* de memòria dinàmic. En aquest cas, l'única modificació en el fragment de codi seria, en crear el flux de sortida a la segona línia, simplement instanciar `ByteArrayOutputStream` classe en lloc de `FileOutputStream`. La resta del codi queda exactament igual. Passa el mateix si es canvia l'origen.

El codi següent escriuria un *array* de bytes en un fitxer:

```
byte[] array = ...

InputStream in = new ByteArrayInputStream(array);
```

```
4 OutputStream out = new FileOutputStream(novaRuta);
5 copiaDades(in, out);
```

El codi següent llegiria el contingut en un array de bytes:

```
1 InputStream in = new FileInputStream(ruta);
2 OutputStream out = new ByteArrayOutputStream();
3 copiaDades(in, out);
4 byte[] array = out.toByteArray();
```

Per tant, el codi del mètode `copiaDades` es manté exactament igual. Amb aquest sistema, el processament de les dades és absolutament transparent en el seu origen o destinació real. I un cop més, tot plegat gràcies a l'aplicació correcta del polimorfisme.

2.3 Fluxos orientats a caràcter

La particularitat principal dels fluxos orientats a caràcter, en contraposició amb els orientats a dades, és que els mètodes de les seves classes operen amb el tipus primitiu char en lloc de byte. La decisió de crear un subconjunt de classes amb aquesta propietat no va ser arbitrària, i depèn d'un seguit de motius de pes.

ASCII és l'acrònim d'*american standard code for information interchange* (codi estàndard americà per a l'intercanvi d'informació).

En menor mesura, al contrari que amb els bytes, hi ha caràcters amb un cert significat especial. Saber que les dades que s'estan transmetent són caràcters permet processar-les correctament i poder detectar ubicacions concretes dins els text. El cas més clar d'aquest fet és el salt de línia, que permet distingir entre línies diferents dins un text.

En major mesura, la diferenciació entre bytes i caràcters permet la internacionalització d'aplicacions. Tradicionalment, el sistema per codificar caràcters ha estat, i en moltes aplicacions encara ho és, el sistema ASCII, formalitzat inicialment l'any 1963, que es basa en caràcters d'un sol byte. Aquest sistema conforma una taula en què a cada valor possible representable amb un byte s'assigna un caràcter concret (per exemple, la *A* majúscula es representa amb el valor hexadecimal 41).

Dec	Hex	Oct	Char
64	40	100	@
65	41	101	A
66	42	102	B
67	43	103	C
68	44	104	D
69	45	105	E

Part de la taula ASCII.

A pesar de l'enorme acceptació, aquest sistema té un problema molt important: es basa totalment en l'alfabet llatí i, encara més concretament, en el llenguatge anglès. Per tant, qualsevol llenguatge no representable en aquest alfabet no es pot representar en ASCII: grec, rus, pràcticament totes les llengües orientals, etc. En aquests països es van desenvolupar altres sistemes de codificació totalment incompatibles amb l'ASCII, en assignar a un byte determinat una simbologia diferent, cosa que feia molt complicat fer aplicacions fàcilment portables independentment de l'idioma del sistema en què s'executi. Per resoldre aquest problema es va crear la codificació Unicode al final de la dècada dels vuitanta. Aquesta codificació es basa en 16 bits i és capaç d'englobar una gran quantitat d'alfabets. Per permetre la retrocompatibilitat amb el codi ASCII, els valors Unicode 0 0000-0 007F, quan el byte de més pes és 0, es corresponen exactament amb els valors definits per ASCII. La figura 2.3 mostra un bocí de la seva gran taula, en aquest cas per caràcters japonesos.

Figura 2.3. Rang de la taula Unicode pel sil.labari japonès hiragana.

Taula Unicode per Hiragana

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
U+304x		ぁ	あ	ぃ	い	ぅ	う	ぇ	え	ぉ	お	か	が	き	ぎ	く
U+305x	ぐ	け	げ	こ	ご	さ	ざ	し	じ	す	ず	せ	ぜ	そ	ぞ	た
U+306x	だ	ち	ぢ	っ	つ	づ	て	で	と	ど	な	に	ぬ	ね	の	は
U+307x	ば	ぱ	ひ	び	ぴ	ふ	ぶ	ぷ	へ	べ	ぺ	ほ	ぼ	ぽ	ま	み
U+308x	む	め	も	ゃ	や	ゅ	ゆ	ょ	よ	ら	り	る	れ	ろ	ゎ	わ
U+309x	ゐ	ゑ	を	ん	ゔ	ゕ	ゖ			゙	゚	゛	゜	ゝ	ゞ	

El Java es basa totalment en l'Unicode, els tipus primitius char ocupen 2 bytes, cosa que permet que una aplicació Java s'executi sobre qualsevol plataforma, independentment de l'idioma. Novament, es pot veure com el Java es va crear pensant en Internet.

Les classes **Reader** i **Writer** representen les superclasses associades a fluxos orientats a caràcter. Sempre que es treballa amb caràcters cal usar la seva jerarquia de classes.

La filosofia dels fluxos orientats a caràcter és exactament igual que en els fluxos orientats a dades, i només canvia tota ocurrència de byte a `char`.

Per cada origen o destinació de dades també hi ha una classe concreta, `FileReader` i `FileWriter` per processar fitxers. Fins i tot el nombre, nom i format dels mètodes són idèntics (`write`, `read`).

Còpia de fitxers de text

Les diferències de la còpia de fitxers de text són mínimes respecte a la manera d'operar dels fluxos orientats a dades.

```
1  Reader in = new Filereader(ruta);
2  Writer out = new FileWriter(novaRuta);
3  copiaDades(in, out);
4  ...
5  public void copiaDades(Reader in, Writer out) {
6    try {
7      char[] dades = new char[100];
8      int llegits = 0;
9      while (-1 != (llegits = in.read(dades)))
10       out.write(dades,0,llegits);
11     out.close();
12     in.close();
13   } catch (IOException) { ... }
14 }
```

De manera homònima als fluxos relatius a *buffers* de memòria, també hi ha classes per gestionar *buffers* de caràcters: `CharArrayReader` i `CharArrayWriter`. En aquest cas, el constructor del flux d'entrada té com a paràmetre una variable de tipus `char[]` i el de sortida permet obtenir les dades emmagatzemades usant el mètode `toCharArray()`.

2.4 Modificadors de fluxos

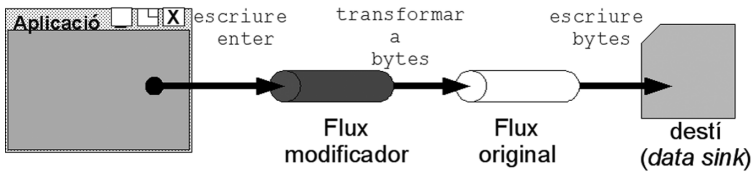
Hi ha situacions en les quals haver de processar qualsevol informació binària directament a un nivell tan baix com de byte o de caràcter pot ser una feina pesada per al desenvolupador. Suposem que es vol emmagatzemar un valor enter, un tipus primitiu `int`. Amb el funcionament per defecte dels fluxos orientats a dades, això implica haver de fer un conjunt de tasques prèvies abans que no es pugui escriure realment: conèixer la mida exacta d'un enter en Java (32 bits), dividir-lo d'alguna manera en bytes independents, i tot seguit escriure'l com una cadena de bytes. Llavors, en llegir-lo, cal fer el procés invers.

Per a casos com aquest, en què les dades requereixen una transformació, la biblioteca `java.io` proporciona un conjunt de classes anomenades*modificadores*.

> Una **classe modificadora d'un flux** altera el seu funcionament per defecte, i proporciona mètodes addicionals que permeten el pre-procés de dades complexes abans d'escriure o llegir-les del flux. Aquest preprocés el realitza de manera transparent el desenvolupador.

La figura 2.4 mostra un esquema del comportament d'aquestes classes partint de la problemàtica exposada a l'inici. El que vol el desenvolupador és simplement poder disposar d'un mecanisme per escriure enters en el flux, i deixar-lo que s'encarregui de totes les transformacions necessàries per convertir-lo en una cadena de bytes.

FIGURA 2.4. Modificació del comportament d'un flux de sortida.



Les classes modificadores més significatives són els fluxos de tipus de dades, els fluxos amb *buffer* intermedi, la sortida amb format, la compressió de dades, la transformació del flux orientat a caràcter de dades i la lectura per línies. En tots els casos, el seus constructors tenen com a paràmetre el flux que es vol modificar.

2.4.1 Fluxos de tipus de dades

Les classes `DataInputStream` i `DataOutputStream` serveixen per resoldre exactament el problema proposat a l'inici d'aquest apartat. Proporcionen un seguit de mètodes addicionals que permeten escriure directament tipus primitius sense que el desenvolupador s'hagi de preocupar de com cal codificar-los en bytes:

- `void writeInt(int i)`
- `int readInt()`
- `void writeBoolean(boolean b)`
- `boolean readBoolean()`
- `void writeDouble(double d)`
- `double readDouble()`
- etc.

Escriptura i lectura de tipus primitius

Un exemple d'utilització d'aquest modificador, en què s'escriu directament un valor enter, seria:

```
1 DataOutputStream dos = new DataOutputStream(new FileOutputStream(
      ruta));
2 dos.writeInt(enter);
3 dos.writeBoolean(boolea1);
4 dos.writeBoolean(boolea2);
5 dos.writeDouble(doble);
6 dos.close();
```

En usar aquest tipus de flux cal anar amb molt de compte de llegir dades en exactament l'ordre invers en què s'han escrit, ja que en cas contrari el programa serà erroni. Així, doncs, per llegir el fitxer anterior correctament cal fer:

```
1 DataInputStream dis = new DataInputStream(new FileInputStream(
      ruta));
2 double d = dis.readDouble();
3 boolean b1 = dis.readBoolean();
4 boolean b2 = dis.readBoolean();
5 int i = dis.readInt();
6 dis.close();
```

2.4.2 Fluxos amb buffer intermedi

La classe `BufferedInputStream` proporciona la capacitat de disposar d'un *buffer* de memòria intermedi entre l'aplicació i un flux d'entrada orientat a dades. A efectes pràctics, això significa que permet tornar enrera en qualsevol moment de la lectura o l'escriptura, al contrari del que normalment es permet. Per assolir aquesta fita, disposa de mètodes addicionals:

- **void mark(int limit)**. Marca una posició del flux. La posició marcada es conserva sempre que no es llegeixin més de `limit` bytes (que correspondria a la mida del *buffer* intermedi). En cas que això succeeixi, la marca es perd. Si bé aquest mètode es pot cridar diverses vegades al llarg de la vida del flux, com a màxim hi pot haver una única marca vàlida. Aquesta sempre serà la corresponent a la darrera crida d'aquest mètode.

- **void reset()**. El flux retrocedeix el processament de dades de nou fins a la posició marcada. En les lectures següents, es tornaran a obtenir exactament les mateixes dades que es van obtenir tot just després de cridar mark.

Retrocedint en la lectura d'un flux

Un tros de codi mostrant el seu funcionament seria el següent. En aquest es llegeixen els primers 200 bytes i llavors, tot seguit, es tornen a llegir. El paràmetre del mètode mark indica quina és la mida del *buffer* i, per tant, el límit de bytes que es poden llegir fins que la marca deixa de ser vàlida i es perd:

```
1  BufferedInputStream bis = new BufferedInputStream(new
       FileInputStream(ruta));
2  byte[] dades = new int[100];
3  bis.mark(500);   //Marca. El buffer serà de 500 bytes
4  bis.read(dades); //Llegim els bytes 0-99 del fitxer
5  bis.read(dades); //Llegim els bytes 100-199 del fitxer
6  bis.reset();     //Tornem a la marca
7  bis.read(dades); //Llegim els bytes 0-99 del fitxer
8  bis.read(dades); //Llegim els bytes 100-199 del fitxer
9  bis.read(dades); //Llegim els bytes 200-299 del fitxer
10 bis.read(dades); //Llegim els bytes 300-399 del fitxer
11 bis.read(dades); //Llegim els bytes 400-499 del fitxer
12 bis.read(dades); //La marca es perd. Ja no es pot fer reset()
13 ...
14 bis.close();
```

També existeix una classe `BufferedOutputStream`, tot i que aquesta té un comportament absolutament diferent. Simplement serveix per optimitzar alguns dels processos d'escriptura de dades del sistema operatiu. A part d'això, no aporta cap altre funcionalitat en forma de nous mètodes.

2.4.3 Sortida amb format

La classe `PrintStream` és imprescindible dins de qualsevol aplicació que ha d'escriure cadenes de test dins d'un flux, ja que es tracta d'un modificador de fluxos de sortida que proporciona dos mètodes, sobrecarregats per poder tractar paràmetres de qualsevol tipus primitiu o objecte:

- **void print(...)**. Escriu la representació en forma de cadena de text del paràmetre d'entrada. Per exemple, si el paràmetre és un booleà a cert, escriu la cadena de text "*true*", si és el número 24, escriu la cadena de text "24", etc.
- **void println(...)**. Escriu la representació en forma de cadena de text del paràmetre d'entrada i al final fa un salt de línia.

toString

Si aquest mètode no ha estat redefinit a la classe de l'objecte a representar, s'executarà el mètode definit a la classe , que simplement mostra per pantalla la referència de l'objecte: `Test$MyClass@13e205f`.

`System.out` és un `PrintStream`. Per això per escriure línies de text per pantalla s'usa: `System.out.println(...)`.

Així, doncs, aquests mètodes transformen qualsevol cosa en una cadena de bytes d'acord amb la seva representació com a cadena de text (un objecte `String`). En cas que el paràmetre sigui un objecte, la transformació en cadena de text es realitza mitjançant la crida interna del mètode `toString()`. Atès que aquest mètode està

definit en la mateixa classeObject Object, sempre es pot garantir que és possible cridar-lo.

Tot i ser una mica estrany, en tractar-se d'un flux que transforma dades en cadenes de text, es considera orientat a dades i no a caràcter. `PrintStream` també és un cas especial en el fet que disposa de constructors addicionals en què es poden especificar directament alguns tipus de destinacions de dades.

- `PrintStream(File fitxer)`
- `PrintStream(OutputStream out)`
- `PrintStream(String nomFitxer)`

Mostrant enters

En el fragment de codi que es mostra tot seguit es veu un exemple senzill de les funcionalitats dePrintStream, mitjançant el qual és possible imprimir línies de text amb un format complex:

```
PrintStream ps =   new PrintStream(ruta);
for (int i = 0; i < 10; i++)
  ps.println( i + ". i val 5? " + (i == ));
...
ps.close();
```

El resultat és:

```
1. i val 5? false
2. i val 5? false
3. i val 5? false
4. i val 5? false
5. i val 5? true
6. i val 5? false
7. i val 5? false
8. i val 5? false
9. i val 5? false
10. i val 5? false
```

2.4.4 Compressió de dades

Compressió zip

Uns altres tipus de fluxos, una mica més complexos, que permeten tractar dades comprimides són `ZipInputStream` i `ZipOutputStream`

El Java incorpora dins la biblioteca de fluxos la possibilitat de transmetre i llegir dades comprimides de manera transparent. La manera més simple de ferho és mitjançant les classes `GzipInputStream` i `GzipOutputStream`, que usen l'algorisme de compressió GZIP. Cap de les dues classes inclou nous mètodes fora dels definits en les superclasses `InputStream` i `OutputStream`. A mesura que s'escriuen o es llegeixen dades amb els mètodes `write` o `read`, aquestes es comprimeixen automàticament sense que sigui necessari fer cap altra tasca addicional.

Al contrari que la resta de classes, aquestes pertanyen al paquet `java.util.zip`.

Compressió i descompressió de dades

Tot seguit es mostra un exemple senzill dels mecanismes de compressió de dades. Com es pot veure, tot és igual a escriure o llegir dades d'un flux qualsevol.

```
try {
 GZIPInputStream  gis = new GZIPInputStream(new FileInputStream(
     ruta));
 OutputStream out = new FileOutputStream(outFilename);
 byte[] dades = new byte[1024];
 int llegits = 0;
 while (-1 != (llegits = in.read(dades)))
  out.write(dades,0,llegits);
 gis.close();
 out.close();
} catch (IOException) { ... }
```

2.4.5 Traducció de flux orientat a caràcter a dades

Dins el conjunt de classes modificadores, també hi ha dues classes que permeten traduir un flux orientat a dades a un orientat a caràcter, de manera que es pot operar a nivell de char en lloc de byte. Es tracta de les classes `InputStreamReader` i `OutputStreamWriter`.

Totes aquestes classes modificadores, de fet, són subclasses de `InputStream` i `OutputStream`, ja que també es consideren fluxos per si mateixes.

D'InputStream a Reader

A continuació es mostra com un flux orientat a dades amb origen en un fitxer es pot processar com un flux orientat a caràcter, sempre que se sàpiga *a priori* que el fitxer conté text.

```
FileInputStream fis = new FileInputStream (ruta);
FileOutputStream fos = new FileOutputStream (novaRuta);
...
public void copiaText(InputStream in, OutputStream out) {
  try {
    Reader rd = new InputStreamReader(in);
    Writer wr = new OutputStreamWriter(out);
    char[] dades = new char[100];
    int llegits = 0;
    while (-1 != (llegits = rd.read(dades)))
      wr.write(dades,0,llegits);
    wr.close();
    rd.close();
  } catch (IOException) { ... }
}
```

2.4.6 Lectura per línies

La classe més útil entre els modificadors de fluxos orientats a caràcter és `BufferedReader`, que permet la lectura de línies completes de text mitjançant el mètode `readLine()`, que retorna directament un `String`. Ella sola s'encarrega

de llegir automàticament tots els caràcters necessaris fins a trobar un salt de línia. Aquesta classe també es considera un `Reader`, perquè és una subclasse seva.

Tot seguit es mostra com es pot usar amb un bocí de codi d'exemple, en què es mostra per pantalla el contingut d'un fitxer de text, llegint-lo línia a línia:

```
BufferedReader br = new BufferedReader (new FileReader(ruta));
String linia = null;
//Es mostra tot el contingut per pantalla
while (null != (linia = br.readLine()))
  System.out.println(linia);
br.close();
```

2.5 Operacions avançades

Els fluxos proporcionen un mecanisme genèric per al processament de grans volums d'informació de manera seqüencial. Tot i que és el cas més intuïtiu, els fluxos no solament es limiten a operar amb fitxers. En aquest apartat es descriuran un seguit de casos particulars de funcionalitats més complexes que ens ofereixen els fluxos o que només poden ser usades quan es treballa exclusivament amb fitxers. Aquestes operacions aporten una solució relativament senzilla davant problemes típics en desenvolupar certes aplicacions, o al menys molt més simples que haver de gestionar les dades byte a byte.

2.5.1 Fitxers de propietats

Carpetes

Un exemple de dades que val la pena desar entre execucions són les carpetes en què es desen certs fitxers importants amb lloc d'emmagatzemament variable (biblioteques, directoris de treball, etc.).

Un cas molt concret de fitxer que freqüentment s'utilitza en diverses aplicacions és el cas dels fitxers de propietats. En aquesta mena de fitxers s'emmagatzema informació relativa a la configuració de l'aplicació, de manera que es conservi el seu comportament entre diferents execucions.

Conceptualment, un fitxer de preferències consisteix en una successió d'elements, en què cadascun es compon d'una clau, en format cadena de text, i un valor associat, que pot ser tant una cadena de text com un tipus primitiu. Per exemple:

Normalment, cal evitar usar espais en blanc en les claus i els valors.

```
DirTreball = \home\usuari\dirTreball
HoraDarreraExecucio = 05/09/2011-17:00:03
ErrorsPendents = 4
```

Properties

Tot i que aquesta classe permet generar estructures molt més complexes, el text d'aquest apartat se centrarà a explicar com es pot generar una successió de claus i valors.

Si bé res no impedeix usar directament la classe `File` i fluxos orientats a caràcter com un `BufferedReader` per anar llegint línia a línia i processar-ne el contingut, el Java ofereix una classe que permet processar aquesta mena d'informació i llegir-la o escriure-la fàcilment en un fitxer.

La classe que gestiona directament conjunts propietats en el Java s'anomena **Properties**, dins el paquet `java.util`.

Per generar un nou conjunt de propietats partint de zero, és suficient de cridar el constructor per defecte `Properties()`. Un cop s'instancia un conjunt de propietats, és possible consultar i modificar els valors emmagatzemats mitjançant mètodes molt semblants als utilitzats per la classe `Map`, ja que un fitxer de propietats té exactament la mateixa estructura. Tot i així, només és possible operar amb cadenes de text, no objectes. Aquests mètodes són:

- **Object setProperty(String key, String value)**. Assigna a la propietat `key` el valor `value`. Si no existeix, la crea.
- **String getProperty(String key)**. Consulta el valor d'una propietat. Retorna `null`, si no existeix.

Totes les dades modificades a l'objecte `Properties` només tenen representació en la memòria. Per aconseguir-ne la persistència cal desar-les (normalment, en un fitxer). Els mètodes que permeten fer-ho són:

- **void store (OutputStream os, String comment)**. Desa un fitxer de preferències al flux de sortida `os`. En la primera línia s'afegeix el text addicional, només a efectes informatius per a qualsevol persona que obrís el fitxer amb un editor de textos, `comment`.
- **void store (Writer wr, String comment)**. Idèntic a l'anterior, però opera sobre el flux orientat a caràcter `wr`.

Generant i desant propietats

Un exemple senzill de generar i desar propietats és el següent:

```
1 Properties prop = new Properties();
2 prop.setProperty("DirTreball","/home/usuari/DirTreball");
3 prop.setProperty("HoraDarreraExecucio","05/09/2009-17:00:03");
4 prop.setProperty("ErrorsPendents","4");
5 prop.store(new FileOutputStream(ruta),"Fitxer de configuració");
```

El resultat seria el fitxer:

```
1 #Fitxer de configuració
2 #Tue Mar 24 09:45:50 JST 2009
3 HoraDarreraExecucio=05/09/2009-17\:00\:03
4 ErrorsPendents=4
5 DirTreball=/home/usuari/DirTreball
```

Excepte a la primera vegada que s'executi l'aplicació, normalment el que es farà es carregar un fitxer de propietats. Igual que existeixen uns mètodes per desar dades, hi ha els següents per carregar-les:

El mètode `propertyNames` permet obtenir una `Enumeration` amb els noms de totes les propietats disponibles.

- **void load (InputStream is)**. Carrega un fitxer de preferències des del flux d'entrada `is`.

- **void load (Reader rd)**. Carrega un fitxer de preferències des del flux d'entrada orientat a caràcter `rd`.

En qualsevol cas, el flux d'entrada sempre ha de ser un fitxer de propietats correctament formatat, d'acord amb el resultat d'una crida `store`, pel que les dades contingudes sempre són text. Aquest fitxer es pot modificar mitjançant qualsevol editor de text simple, però sempre cal anar amb compte de mantenir el format. Les línies totalment en blanc o que comencen amb el caràcter # s'ignoren en carregar un fitxer de propietats.

Normalment, amb el format de text simple proporcionat pels mètodes `store` i `load` és més que suficient per gestionar un conjunt de propietat. Tot i així, la classe `Properties` també proporciona els mètodes `storeToXML` i `loadfromXML`, que emmagatzemen les dades en format XML, un llenguatge de marques jeràrquic. Tot i així, en aquest cas, la seva funcionalitat és idèntica, i usar XML no aporta cap capacitat addicional.

L'exemple anterior en format XML seria:

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<!DOCTYPE properties SYSTEM
  "http://java.sun.com/dtd/properties.dtd">
  <properties>
    <comment>Fitxer de configuració</comment>
    <entry key="ErrorsPendents">4</entry>
    <entry key="HoraDarreraExecucio">05/09/2009-17:00:03</entry>
    <entry key="DirTreball">/home/usuari/DirTreball</entry>
  </properties>
```

2.5.2 Seriació d'objectes

Hi ha situacions en què el desenvolupador pot decidir que no vol haver de pensar quines dades concretes cal emmagatzemar dins un fitxer, o haver d'especificar quin format han de tenir i processar-les en format binari o de text. Simplement, el que vol és agafar el mapa d'objectes del Model exactament tal com està representat en la memòria i fer un abocament directe. El Java ofereix la possibilitat de fer aquesta acció mitjançant el mecanisme de seriació d'objectes.

> S'anomena **seriació d'objectes** el procés d'escriptura d'un objecte sobre una destinació de dades en forma de cadena de bits a partir de la seva representació en memòria, de manera que a partir de les dades resultants posteriorment es pugui restaurar en exactament el mateix estat.

Perquè un objecte es pugui seriar, cal que la seva classe implementi la*interface* Java `java.io.Serializable`. Aquesta és una *interface* molt especial, ja que no obliga a implementar absolutament cap mètode, només serveix per indicar que les instàncies d'una classe són seriables.

Per tant, si tenim la classe:

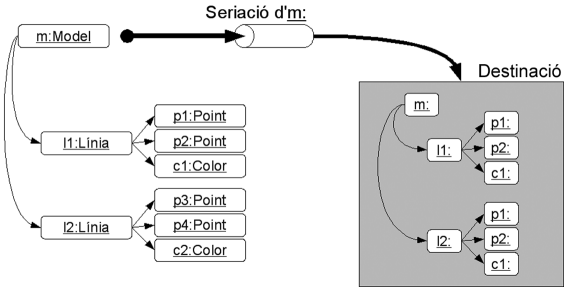
```
1 public class Model {...}
```

Per seriar-ne els objectes, l'única modificació que cal fer és:

```
1 import java.io.Serializable;
2 public class Model implements Serializable {...}
```

Una de les propietats que fa especialment potent el mecanisme de seriació d'objectes del Java és el fet que, en seriar un objecte, se segueixen totes les seves referències a altres objectes, els quals, al mateix temps, també se serien. Aquest procés es produeix iterativament en els nous objectes seriats fins que ja no es troben més referències. Per tant, és possible seriar un mapa d'objectes complet simplement indicant que cal seriar l'objecte de nivell més alt. En restaurar l'objecte seriat, amb ell és recupera el mapa d'objectes complet. Aquest fet es veu representat en la figura 2.5.

FIGURA 2.5. Seriació d'una estructura d'objectes complet a partir d'un únic objecte inicial.



Si un mapa d'objectes es compon d'instàncies de diferents classes, com serà el cas freqüentment, cal que totes implementin `Serializable`. En cas contrari, en intentar seriar una instància d'una classe que no la implementa, es produirà una excepció tipus `java.io.NotSerializableException`.

Els tipus de fluxos de dades associats a la seriació d'objectes són `java.io.ObjectOutputStream` i `java.io.ObjectInputStream`, per llegir i escriure respectivament. Ambdues classes ofereixen un ventall de mètodes per escriure de manera seqüencial tota mena de tipus de dades (en lloc de només bytes o caràcters). Per seriar objectes, els mètodes que cal usar són:

- **public void writeObject(Object o)**. Escriu un objecte en un flux de dades `ObjectOutputStream`.
- **public Object readObject()**. Llegeix un objecte d'un flux de dades `ObjectInputStream`.

A continuació es mostra un exemple de seriació d'objectes a fitxer. La classe `Model` pot ser tan complexa com calgui i referenciar als seus atributs instàncies de qualsevol altra classe. Mentre aquestes també implementin `Serializable`, l'estructura d'objectes completa s'escriurà a disc:

```
Model m = new Model();
File f = new File(ruta);
...
ObjectOutputStream oos = new ObjectOutputStream (new FileOutputStream(f));
oos.writeObject(m);
oos.close();
```

Tot seguit també es mostra l'exemple associat a la recuperació de l'objecte seriat anteriorment. En recuperar l'objecte `m:Model` serialitzat, també s'ha recuperat tot el seu mapa d'objectes subjacent:

```
File f = new File(ruta);
...
ObjectInputStream ois = new ObjectInputStream (new FileInputStream(f));
Model m = (Model)ois.readObject();
ois.close();
```

Fixeu-vos que, com que el mètode `readObject` retorna un `Object`, cal fer un *cast* per assignar la instància retornada a una referència de tipus `Model`. Això implica que el desenvolupador ha de saber exactament quin tipus d'objecte es va emmagatzemar, o en cas contrari es produeix un error en fer aquest *cast*, una `ClassCastException`. Evidentment, també s'ha de complir que totes les classes seriades, els seus fitxers .class, estiguin instal·lades en l'ordinador en què s'executa aquest codi, ja que en cas contrari es produeix una `ClassNotFoundException`. Per tant, cal anar amb compte quan hi ha un intercanvi d'objectes seriats entre diferents màquines.

Recuperant diverses vegades objectes seriats

Suposem que en el codi que recupera l'objecte seriat, el fitxer es torna a llegir i s'aboca el resultat de recuperar l'objecte sobre una variable diferent, `m2: Model`. Que passa si un mapa d'objectes es restaura diverses vegades des d'un mateix origen de dades sobre variables diferents?

La resposta es que obtenim dues copies diferents del mapa d'objectes original, cadascuna amb objectes absolutament independents els uns dels altres.

L'única excepció dins dels mecanismes de seriació per defecte del Java són els atributs estàtics (*static*), que no se serien.

Com es pot apreciar pels exemples, el mecanisme de seriació d'objectes és relativament fàcil d'usar, ja que el Java s'encarrega automàticament de tots els detalls interns sobre la representació dels objectes seriats i la seva instanciació a memòria un cop restaurats.

Seriació personalitzada

Hi ha situacions en què el desenvolupador vol poder especificar de manera més concreta com se serien els objectes. Per exemple, suposem una aplicació en què, en algun objecte, es desa informació privilegiada (com pot ser una contrasenya), que no es vol escriure en el flux de dades en seriar-lo a un fitxer, ja que llavors seria lliurement accessible per a qualsevol amb accés al fitxer.

En casos com aquest, en què es vol més control sobre el procés de seriació, el Java ofereix diferents opcions.

L'opció més senzilla, si bé també la menys potent, és definir un atribut com a **transitori** amb la paraula clau transient. Els atributs marcats d'aquesta manera són ignorats completament pel procés de seriació. En restaurar un objecte serialitzat amb atributs transitoris, aquests són inicialitzats a zero o `null`, segons el tipus. En la majoria de casos, aquesta via és més que suficient.

Atès que els atributs no seriats es restauren amb un valor segurament invàlid, és important no oblidar que és responsabilitat del desenvolupador generar el codi que els assigni un valor correcte, de manera que l'objecte estigui amb tots els atributs correctament inicialitzats abans de seguir l'execució de l'aplicació. Per exemple, en el cas de la contrasenya, un cop recuperat l'objecte serialitzat caldria preguntar-la immediatament a l'usuari.

Una opció més complexa, però que ofereix molt més poder al desenvolupador, és establir exactament de quina manera se serien realment els objectes. Per fer-ho, cal afegir un seguit de mètodes a cada classe que implementa `Serializable`:

- **private void writeObject(ObjectOutputStream out) throws IOException**. Aquest mètode és el responsable final de seriar. Si s'implementa, el codi que s'executa realment en cridar el mètode writeObject sobre un ObjectOutputStream, que és el paràmetre d'entrada out proporcionat automàticament pel Java, és aquest. Per realitzar aquesta tasca, és possible cridar sobre out tot un seguit de mètodes capaços de seriar qualsevol tipus primitiu o abocar qualsevol tipus de dades binàries. Tots aquests mètodes es troben definits en la classe `DataOutput`.

  Si és necessari, en qualsevol moment és possible cridar el mecanisme de seriació per defecte de l'objecte en curs mitjançant la crida del mètode `out.defaultWriteObject()`.

- **private void readObject(ObjectInputStream in) throws IOException, ClassNotFoundException**. Aquest és el mètode invers a `writeObject`. Es disposa com a paràmetre d'entrada el flux a partir del qual cal anar recuperant tots els camps de l'objecte d'acord amb la manera en què s'ha realitzat en seriar-lo. Novament, el paràmetre d'entrada in disposa d'un seguit de mètodes auxiliars per recuperar qualsevol tipus primitiu o llegir dades binàries, definits en la classe `DataInput`. També és possible cridar `in.defaultReadObject` per cridar el mecanisme de recuperació per defecte.

A continuació es mostra un exemple de seriació personalitzada. En aquest cas s'ha decidit que, en lloc d'usar els mecanismes proporcionats per defecte pel Java, cada atribut de la classe es codifica com a simple cadena de text separada per salts de línia. A l'hora de recuperar l'objecte, els valors s'han de restaurar en el mateix ordre en què s'han escrit i descodificar-los d'acord amb el format en què s'han emmagatzemat:

```
1 import java.io.*;
2 public class CustomSerialization implements Serializable {
3   private int valorEnter = 3;
4   private boolean valorBoolea = true;
```

```
  private String valorString = "Valor String";
  private void writeObject(ObjectOutputStream out) throws IOException {
    PrintStream ps = new PrintStream(out);
    ps.println(valorEnter);
    ps.println(valorBoolea);
    ps.println(valorString);
  }

  private void readObject(ObjectInputStream in) throws IOException,
      ClassNotFoundException {
    BufferedReader br = new BufferedReader(new InputStreamReader(in));
    String s = br.readLine();
    valorEnter = Integer.parseInt(s);
    s = br.readLine();
    valorBoolea = ("true".equals(s))?true:false;
    s = br.readLine();
    valorString = s;
  }
}
```

El contingut de les dades seriades es mostra en l'exemple següent. En negreta es remarquen les parts que s'han generat de manera personalitzada amb el codi anterior. La resta són camps necessaris perquè el Java sigui capaç de reconèixer els noms dels diferents atributs:

Exemple de dades serialitzades.

```
sr! !CustomSerialization i2 Z
LL
valorBooleaI valorEnterL valorStringtLjava/lang/String;¶xpw3 true
      Valor String
```

Res no impedeix al desenvolupador decidir no seriar algun dels atributs i, en el moment de la restauració, assignar el valor que cregui convenient o preguntar-lo a l'usuari en el mateix moment.

Com es pot tornar a veure, la *interface* `Serializable` és un cas molt especial, ja que, estranyament, els mètodes a afegir són privats, però, tot i així, el Java és capaç de cridar-los correctament per seriar l'objecte d'acord amb el seu codi. No cal donar-hi més importància.

2.5.3 Accés aleatori

Una de les característiques essencials de la gestió de dades emmagatzemades dins un fitxer mitjançant fluxos és el caràcter seqüencial en les operacions tant de lectura com d'escriptura. En tractar-se d'un mecanisme genèric, es va definir el denominador comú a qualsevol operació d'entrada sortida. Tot i així, per al cas concret d'un fitxer, es pot garantir que totes les dades són en una ubicació concreta, de manera que s'hi pot accedir de manera aleatòria.

Accés aleatori

Amb aquest nom es denomina la capacitat de llegir dades en qualsevol ubicació dins una seqüència, sense haver de processar prèviament les dades anteriors.

Per poder accedir de manera aleatòria a un **fitxer**, el Java ofereix la classe `RandomAccessFile`.

Els seus constructors són:

- `RandomAccessFile(File fitxer, String mode)`
- `RandomAccessFile(String ruta, String mode)`

Novament, el constructor està sobrecarregat per acceptar tant un objecte `File` com directament la ruta del fitxer per mitjà dels paràmetres fitxer o ruta. El paràmetre mode indica en quin mode es vol obrir el fitxer. Els diferents modes possibles són:

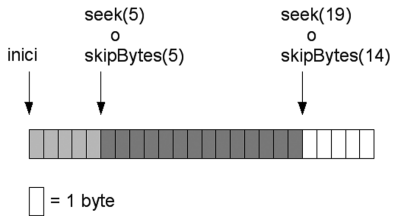
- r **:** Mode escriptura. Qualsevol intent d'escriure en el fitxer, incloent-hi el fet que no existeixi, causarà una excepció.
- rw: Mode escriptura-lectura. Si el fitxer no existeix, se'n crearà un de nou, buit.
- rws: Igual que el cas rw, però, addicionalment, es força l'actualització al sistema de fitxers cada cop que es fa una modificació en les dades del fitxer o les seves metadades. Aquest comportament és semblant a fer un *flush* cada cop que es fa una escriptura en el fitxer.
- rwd:Igual que el cas anterior, però només es força l'actualització per al cas de dades, i no metadades.

L'accés a un `RandomAccessFile` es basa en un apuntador intern que és possible desplaçar arbitràriament a qualsevol posició, partint del fet que la posició 0 correspon al primer byte del fitxer. Tots els increments en la posició d'aquest apuntador són en nombre de bytes. Per gestionar la posició d'aquest apuntador, la classe defineix amb els mètodes específics següents:

- `void seek(long pos)`. Ubica l'apuntador exactament en la posició especificada pel paràmetre pos, en bytes de manera que qualsevol accés a les dades serà sobre aquest byte. No hi ha cap restricció en el valor d'aquest paràmetre, i és possible ubicar l'apuntador molt més enllà del final real del fitxer. En aquest cas, la mida del fitxer es veurà incrementada fins a pos bytes en el moment en què es faci alguna escriptura.
- `long getFilePointer()`. Retorna la posició exacta de l'apuntador, en nombre de bytes, des de l'inici del fitxer.
- `int skipBytes(int n)`. Salta n bytes a partir de la posició actual de l'apuntador, de manera que aquest passa a valer (apuntador + n). Retorna el nombre real de bytes saltats, ja que si s'arriba al final del fitxer, el desplaçament de l'apuntador s'atura.
- `void setLength(long len)`. Assigna una nova longitud al fitxer. Si la nova longitud és menor que l'actual, el fitxer es trunca.

La figura 2.6 mostra un esquema de posicionament de l'apuntador del fitxer d'acord amb crides successives als mètodes `seek` (posicionament absolut) o `skipBytes` (posicionament relatiu respecte al darrer valor de l'apuntador).

FIGURA 2.6. Posicionament d el'apuntador a un fitxer d'accés aleatori.



Un cop ubicats en una posició concreta dins el fitxer, és possible llegir o escriure dades utilitzant tot un seguit de mètodes de lectura i escriptura definits, havent-n'hi una per cada tipus primitiu (`read/writeBoolean`, `read/writeInt`, etc.). En aquest aspecte, `RandomAccessFile` es comporta com les classes `DataInputStream` i `DataOutputStream` i totes les consideracions esmentades per a aquestes classes també s'apliquen en el cas d'accés aleatori. El nombre de bytes escrits dependrà de la mida associada al tipus primitiu a Java.

Si en llegir dades l'apuntador acaba més enllà de la mida del fitxer, es llança aquesta excepció. Gairebé sempre succeeix per una situació d'error en el codi.

Cada cop que es fa una operació de **lectura o escriptura**, l'apuntador es desplaça el mateix nombre de bytes que el nombre al qual s'ha accedit.

Accés aleatori en un fitxer d'enters i reals

En aquest exemple es mostra com es gestiona un fitxer relativament senzill en què un cert nombre de valors són de tipus enter (valorsInt), i tot seguit hi ha un altre conjunt de valors de tipus real (valorsDouble).

```
1 int[] valorsInt = ...;
2 double[] valorsDouble = ...;
```

Per generar un fitxer amb aquests valors, n'hi ha prou d'ubicar l'apuntador en la posició inicial del fitxer i anar escrivint els valors usant el mètode `writeXXX` adequat.

```
1 RandomAccessFile file = new RandomAccessFile(ruta, "rw");
2 for(int i = 0; i < valorsInt.length; i++) file.writeInt(valors[i]);
3 for(int i = 0; i < valorsDouble.length; i++) file.writeDouble(valors[i]);
4 file.close();
```

Per modificar un valor qualsevol, cal ubicar l'apuntador fins a l'inici del valor adequat. Ara bé, per a això s'ha de calcular el desplaçament correcte d'acord amb el nombre de bytes que ocupa cadascun dels valors emmagatzemats. Un `int` en el Java ocupa 4 bytes mentre que un `double` n'ocupa 8.

Aquest exemple modifica el tercer real emmagatzemat en el fitxer. Per fer-ho, cal saltar els bytes associats a tots els enters i els dos primers reals.

```
1 double nouValorDouble = ...;
2 RandomAccessFile file = new RandomAccessFile(ruta, "rw");
3 file.seek(4*valorsInt.length + 8*2);
4 file.writeDouble(nouValorDouble);
5 file.close();
```

Per llegir valors, cal ubicar l'apuntador a l'inici de cada un i anar fent crides al mètode `readXXX` associat al tipus primitiu esperat. Novament, si es volen llegir posicions no consecutives, cal anar recalculant els desplaçaments correctes dins el fitxer. En aquest tros de codi es mostren els primers quatre valors per cada cas.

```
RandomAccessFile file = new RandomAccessFile(ruta, "r");
  for(int i = 0; i < 4; i++)
    System.out.println("Int " + i + " = " + file.readInt());
  file.seek(4*valorsInt.length);
  for(int i = 0; i < 4; i++)
    System.out.println("Double " + i + " = " + file.readDouble());
file.close();
```

Tal com es desprèn dels exemples, un dels aspectes amb què cal anar amb més cura en usar l'accés aleatori és el fet que el posicionament de l'apuntador dins el fitxer es realitza comptant en nombre de bytes, però totes les escriptures i lectures es realitzen directament en tipus primitius. Això implica que el desenvolupador que està generant codi, per accedir a un fitxer ha de saber exactament la seva estructura interna i recordar la mida exacta de cada tipus primitiu de Java, per poder fer els salts a les posicions exactes en què comença cada dada emmagatzemada. En cas contrari, si es comet un error es llegiran o se sobreescriuran parcialment dades incorrectes.

Escriptures i lectures incorrectes

Si només es fan lectures i escriptures de bytes, el problema de saber la mida exacta de cada tipus primitiu de Java desapareix.

Què passa si no es calculen correctament els desplaçaments en accedir a un fitxer de manera aleatòria? Suposem que els *arrays* de valors emmagatzemats tenen deu elements, per la qual cosa el fitxer conté primer deu valors enters (4 bytes cadascun) i deu valors reals (8 bytes cadascun). La mida total del fitxer és, per tant, de 4*10 + 8*10 = 120 bytes. Si, per exemple, es fa:

```
double nouValorDouble = ...;
RandomAccessFile file = new RandomAccessFile(ruta, "rw");
file.seek(4*3);
file.writeDouble(nouValorDouble);
file.close();
```

Abans de l'escriptura, l'apuntador del fitxer en realitat es troba sobre el quart enter. Atès que `writeDouble` escriu 8 bytes (la mida d'un real), se sobreescriuran el cinquè i sisè enter amb la representació binària d'un real. És a dir, un valor totalment incorrecte.

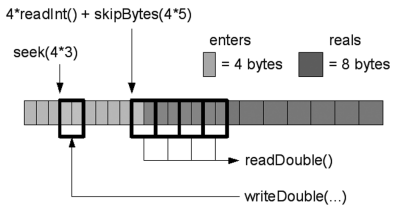
De la mateixa manera, si es fes:

```
RandomAccessFile file = new RandomAccessFile(ruta, "r");
for(int i = 0; i < 4; i++)
  System.out.println("Int " + i + " = " + file.readInt() );
file.skipBytes(4*5);
for(int i = 0; i < 4; i++)
  System.out.println("Double " + i + " = " + file.readDouble() );
file.close();
```

Al final del primer bucle l'apuntador és a l'inici del cinquè enter. Atès que cada enter ocupa 4 bytes, el mètode `skipBytes` deixa l'apuntador a l'inici del desè enter. Llavors, en fer el primer `readDouble` i llegir 8 bytes (la mida d'un real), en

realitat el que es llegirà són els 4 bytes del darrer enter i els primers 4 bytes del primer real. Per pantalla es mostrarà el valor que equival a interpretar com un real aquests 8 bytes incorrectes. Successivament, llavors el bucle llegirà tres cops els darrers 4 bytes d'un real i els primers 4 bytes del següent. Les situacions descrites en l'exemple de lectures i escriptures incorrectes es troben esquematitzades en la figura 2.7.

FIGURA 2.7. Exemple d'escriptura incorrecte en accés aleatori



2.5.4 Fitxers mapats en la memòria

Hi ha situacions en què es vol tractar amb fitxers de dades molt grans, de manera que l'aplicació, o bé es ressent en el rendiment, o directament la màquina virtual del Java retorna una excepció en forma d'una `OutofMemoryError`. Simplement, no és viable carregar totes les dades en la memòria per processar-les.

Una altra opció és usar directament fitxers d'accés aleatori, però aquesta via és ineficient, ja que l'accés directe a un disc sobre fitxers molt grans és força lent.

El paquet `java.nio` només existeix des de la versió 1.4 del Java.

La solució és usar algun mecanisme que permeti carregar només la part del fitxer que es vol tractar en la memòria, de manera que es pot operar de manera eficient sense haver de tenir totes les dades en la memòria. Un cop fetes les lectures o escriptures pertinents, només cal desar aquest bloc en un disc. Com que si el desenvolupador hagués de fer aquesta gestió seria una tasca molt pesada, el Java ja ofereix un mecanisme transparent, si bé d'una certa complexitat, dins el seu paquet avançat d'entrada/sortida, `java.nio`.

La classe **MappedByteBuffer** permet operar amb regions d'un fitxer de mida arbitrària com si aquest estigués directament emmagatzemat en la memòria.

Aquesta classe ofereix una *interface* per fer escriptures i lectures sobre parts d'un fitxer de mida arbitrària, de manera que el Java, internament, ja gestiona la càrrega en la memòria de les parts que realment s'estan utilitzant i d'anar-les escrivint automàticament en el fitxer en cas que es modifiquin. L'escriptura es fa de la manera més eficient possible pel sistema operatiu en què s'executi l'aplicació.

Els canals disponibles en el Java estan definits en el paquet `java.nio.channels`.

Per usar aquesta classe, primer de tot cal obtenir un **canal** (*channel*) a partir d'un fitxer d'accés aleatori, usant el mètode `getChannel()`. Un canal representa una connexió a qualsevol dispositiu d'entrada/sortida, de manera que es pugui llegir o

escriure-hi. Les particularitats dels canals i les operacions que permeten són una mica complexes, per això aquest text se centrarà en el seu ús per al mapatge de fitxers a memòria. N'hi ha prou de saber que la diferència principal dels canals respecte als fluxos és que no són seqüencials i permeten l'accés aleatori (sempre que tingui sentit per al dispositiu final). En el cas que s'està tractant, concretament s'obté una instància de `FileChannel`, atès que s'opera amb fitxers.

Un cop es disposa d'un canal cap al fitxer a accedir, ja és possible mapar una secció del fitxer (o tot sencer) a memòria cridant sobre el canal el mètode:

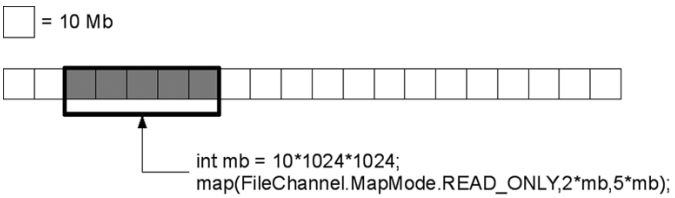
```
1 MappedByteBuffer map(FileChannel.MapMode mode, long position, long size).
```

Una de les capacitats avançades dels canals és la possibilitat de gestionar accés concurrent a un fitxer per part de diverses aplicacions.

Permet processar un fragment d'un fitxer, des de la posició `position` i amb una mida de size bytes, directament, com si estigués ubicat en la memòria. El paràmetre mode indica el mode d'accés d'acord amb un conjunt de constants definides en la classe `FileChannel.MapMode`: `READ_ONLY`, només lectura, i `READ_WRITE`, lectura i escriptura. El mode hauria de coincidir amb l'utilitzat en instanciar el fitxer d'accés aleatori`RandomAccessFile`.

Una visió esquemàtica de tot aquest procés es mostra en la figura 2.8. En aquesta figura es mapa un fragment d'intermedi 50 Mb, donat un fitxer de 200 Mb de llargària.

FIGURA 2.8. Fragment de fitxer mapat en la memòria.



Mitjançant la instància a `MappedByteBuffer` es pot operar amb aquest fragment de fitxer usant-ne els mètodes associats, heretats de la seva superclasse `ByteBuffer`. Com als `RandomAccessFile`, hi ha un apuntador intern que indica des d'on es realitzaran els accessos a les dades, el qual s'incrementa automàticament cada cop que es fa un accés.

Hi ha diversos mètodes sobrecarregats, però cadascun es correspon amb alguna de les categories següents:

1. **Mètodes get o put amb posicionament absolut**. Només operen amb un byte, però permeten establir el desplaçament exacte dins el `MappedByteBuffer` sobre el qual es farà l'operació.
   - `byte get(int posicio)`
   - `void put(int posicio, byte b)`
2. **Mètodes get en bloc**, de manera que es llegeix cert nombre de bytes consecutivament, escrits sobre un *array* de bytes.
   - `void get(byte[] destinacio).` Ł`void get(byte[] destinacio, int offset, int len)`

3. **Mètodes put en bloc**, de manera que s'escriu un cert nombre de bytes consecutivament, proporcionats mitjançant un *array* de bytes.

   - `void put(byte[] origen).` Ł`void put(byte[] origen, int offset, int len)`

4. **Mètodes get i put tant absoluts com relatius**, per accedir a valors representats amb tipus primitius. En aquest aspecte, el seu comportament és idèntic a l'explicat per la classe `RandomAccessFile`.

   - `void putInt(int valor)`
   - `void putInt(int posicio, int valor)`
   - `int getInt()`
   - `int getInt(int posicio)`
   - etc.

Per desplaçar un apuntador dins l'objecte `MappedByteBuffer`, de manera que es pugui triar a partir de quin punt s'accedirà a les dades en els mètodes relatius, s'utilitza:

- **void rewind**(). Retorna a la posició zero.
- **Buffer position(int pos**). Desplaça l'apuntador a la posició pos.

Cal tenir molt present que, com en el cas dels fluxos, els canvis efectuats sobre la porció mapada a memòria no es transmeten immediatament al fitxer físic. Si es vol forçar un *flush*, cal cridar el mètode `force()`.

Treballant amb 256 Mb

Suposem que es vol generar un fitxer de 256 Mb i treballar-hi. Mitjançant la classe MappedByteBuffer és possible accedir-hi en la seva totalitat sense que internament impliqui haver de carregar-lo sencer a memòria.

```
1  int mb = 1024*1024; // 1 Mb
2  int length = 256*mb; // 256 Mb
3
4  RandomAccessFile raf = new RandomAccessFile(ruta, "rw");
5  FileChannel ch = raf.getChannel()
6  MappedByteBuffer mbb = ch.map(FileChannel.MapMode.READ_WRITE,0,
       length);
7  for(int i = 0; i < length; i++)
8    mbb.put((byte)'a');
9
10 //Ubiquem l'apuntador a la meitat
11 mbb.position(128*mb);
12 for(int i = 0; i < length/2; i++)
13   System.out.print((char)out.get());
```

El fitxer que es copia en part a si mateix

Per acabar de veure com funciona un MappedByteBuffer i quina utilitat té poder accedir concurrentment a diferents parts d'un mateix fitxer gran, intenteu entendre el fragment de codi següent:

```
int mb = 1024*1024; // 1 Mb
int length = 128*mb; // 128 Mb

RandomAccessFile raf = new RandomAccessFile(ruta, "rw");
FileChannel ch = raf.getChannel()
MappedByteBuffer mbb1 = ch.map(FileChannel.MapMode.READ_ONLY,0,
    length/2);
MappedByteBuffer mbb2 = ch.map(FileChannel.MapMode.READ_WRITE,
    length/2, length/2);
for(int i = 0; i < length/2; i++) {
  byte b = mbb1.get();
  mbb2.put(b);
}
```

Aquest codi copia la primera meitat d'un fitxer de 128 Mb a la seva segona meitat. Realitzar aquesta tasca d'aquesta manera, mapant zones diferenciades del fitxer a memòria, és molt més eficient que usar directament un `RandomAcessFile`, ja que es disposa de dos apuntadors: un llegeix de l'origen i l'altre escriu a la destinació. En cas contrari, abans de cada lectura i escriptura caldria reposicionar l'apuntador en la seva ubicació correcta. Això implica estar movent constantment l'apuntador sobre un fitxer molt gran, amb la ineficiència resultant.