Programació orientada a objectes

CFGS.INF.M03B/0.12

Desenvolupament d'aplicacions multiplataforma
Desenvolupament d'aplicacions web

Generalitat de Catalunya
Departament d'Educació

ioc
institut obert
de catalunya

Aquesta col·lecció ha estat dissenyada i coordinada des de l'Institut Obert de Catalunya.

Coordinació de continguts
Miguel Angel Carpintero Rodríguez

Redacció de continguts
Joan Arnedo Moreno

Imatge de coberta
John Biehler

Primera edició: Febrer 2013

Dipòsit legal: DL B 12714-2016

Introducció

L'orientació a objectes és una metodologia amb la qual és possible resoldre problemes mitjançant la seva descomposició en els elements fonamentals que els componen i l'especificació de com interactuen. A la unitat formativa "Programació orientada a objectes. Fonaments", que consta de dues unitats, es presenta una introducció general a aquesta metodologia, adoptant la perspectiva tant del dissenyador de programari com del desenvolupador final.

Aquesta introducció la fareu bàsicament a la unitat "Introducció a la programació orientada a objectes. Classes" on a més a més es descriu de quina manera s'estructura una aplicació orientada a objectes i quina és la relació entre els components que es desenvolupen, les classes, i els elements que hi ha a memòria quan s'executen, els objectes.

A la unitat "Utilització avançada de classes" us centrareu en l'ús d'estratègies per poder generar programari que sigui modular, reaprofitable i fàcil d'adaptar davant canvis imprevistos en el futur, tant en aspectes de disseny com d'implementació. Veureu els tres mecanismes més importants dins l'orientació a objectes: l'encapsulació/ocultació d'informació, l'herència i el polimorfisme.

A la unitat formativa "POO. Llibreries de classes fonamentals" trobareu també dues unitats. A la unitat "Classes fonamentals" estudiareu una petita part de les llibreries de classes de Java, centrant-vos en el que es podrien considerar els dos aspectes imprescindibles de cara a dur a terme aplicacions en Java: per una banda, veureu com dur a terme la gestió d'errors mitjançant un conjunt de classes especials anomenades "Excepcions" i d'altra banda, es presentaran les classes que representen les anomenades "Col.leccions" del Java. Aquestes permeten emmagatzemar conjunts arbitraris d'objectes, aportant una alternativa molt més còmoda i, en alguns aspectes, eficient que la utilització d'*arrays*.

A la unitat "Interfícies gràfiques d'usuari. Fluxos i fitxers" tractareu els conceptes necessaris per poder assolir un cert domini de les llibreries de Java que permeten desenvolupar aplicacions amb funcionalitats d'entrada/sortida considerades bàsiques en un sistema modern. En la part on veureu "Interfícies gràfiques d'usuari" es presenta a grans trets la llibreria gràfica de Java, anomenada Swing. Posteriorment en la part de "Fluxos i fitxers" s'expliquen les llibreries d'entrada/sortida usades pel Java per gestionar fluxos.

Per tal d'assolir la persistència de dades, de manera que sigui possible conservar certa informació entre diferents execucions de l'aplicació, o compartir dades entre diferents aplicacions, les aplicacions modernes normalment s'inclinen per usar les bases de dades. Aquest serà el contingut de la unitat formativa "POO. Introducció a la persistència en BD", que consta d'una única unitat "POO i gestors de bases de dades".

Abans de poder establir com fer millor ús d'una base de dades, però, cal tenir ben clar de quina manera s'estructurarà tota la informació dins el vostre programa.

Quins objectes hi ha i quines dades contenen. Per assolir aquesta fita, una eina molt útil és el llenguatge UML. Veureu una petita introducció. A continuació us presentarem els aspectes bàsics per poder assolir la persistència de les dades mitjançant el mecanisme més popular per desplegar aplicacions a gran escala: una base de dades relacional, per acabar introduint un nou tipus de bases de dades, relativament recent, on la informació emmagatzemada si que s'estructura d'una manera semblant a com es fa amb els objectes a memòria dins una aplicació orientada a objectes: les bases de dades orientades a objectes.

Resultats d'aprenentatge

En finalitzar aquest mòdul l'alumne/a:

Programació orientada a objectes. Fonaments

1. Escriu i prova programes senzills, reconeixent i aplicant els fonaments de la programació orientada a objectes
2. Desenvolupa programes organitzats en classes analitzant i aplicant els principis de la programació orientada a objectes
3. Desenvolupa programes aplicant característiques avançades dels llenguatges orientats a objectes i de l'entorn de programació

POO. Llibreries de classes fonamentals

1. Escriu programes que manipulin informació seleccionant i utilitzant els tipus avançats de dades facilitats pel llenguatge
2. Gestiona els errors que poden aparèixer en els programes, utilitzant el control d'excepcions facilitat pel llenguatge.
3. Desenvolupa interfícies gràfiques d'usuari simples, utilitzant les llibreries de classes adequades.
4. Realitza operacions bàsiques d'entrada/sortida de informació, sobre consola i fitxers, utilitzant les llibreries de classes adequades.

POO. Introducció a la persistència en BD

1. Gestiona informació emmagatzemada en bases de dades relacionals mantenint la integritat i consistència de les dades.
2. Gestiona informació emmagatzemada en bases de dades objecte- relacionals mantenint la integritat i consistència de les dades.
3. Utilitza bases de dades orientades a objectes, analitzant les seves característiques i aplicant tècniques per mantenir la persistència de la informació

Continguts

Programació orientada a objectes. Fonaments

Unitat 1

Introducció a la programació orientada a objectes. Classes

1. Fonaments de la programació orientada a objectes
2. Declaració de classes

Unitat 2

Utilització avançada de classes

1. Creació d'aplicacions escalables

POO. Llibreries de classes fonamentals

Unitat 3

Classes fonamentals

1. Classes fonamentals

Unitat 4

Interfícies gràfiques d'usuari. Fluxos i fitxers.

1. Interfícies gràfiques d'usuari
2. Fluxos i fitxers

POO. Introducció a la persistència en BD

Unitat 5

POO i gestors de base de dades

1. Introducció al diagrama estàtic UML
2. Aplicacions amb BD no orientades a objectes
3. Aplicacions amb BD orientades a objectes

Introducció a la programació orientada a objectes. Classes

Joan Arnedo Moreno

Índex

Introducció

Sovint, un dels problemes que us trobareu com a desenvolupadors és decidir com traduir el problema que heu de resoldre en les estructures de dades i la manera com cal que aquestes interactuïn entre si o siguin processades per l'ordinador. Per a problemes relativament simples és molt senzill associar cada dada en algun dels tipus que l'ordinador sap interpretar directament, però per a problemes complexos aquesta tasca es pot complicar. Afortunadament, per tal de facilitar aquesta tasca existeixen diferents metodologies que intenten simplificar aquest procés. Una d'aquestes metodologies és l'anomenada orientació a objectes, la qual es basa en la resolució de problemes mitjançant la seva descomposició en els elements fonamentals que els componen i l'especificació de com interactuen usant com a marc de referència el llenguatge natural, i com funcionarien les coses si el problema, en lloc de resoldre's dins un ordinador, es volgués resoldre en el món real.

Aquesta unitat didàctica exposa els conceptes fonamentals de l'orientació a objectes i proporciona una motivació de per què és considera una metodologia útil i àmpliament usada actualment. Per fer-ho, es presenta una introducció general adoptant la perspectiva tant del dissenyador de programari com del desenvolupador final. Tot i això, només s'usa la perspectiva del segon rol en situacions molt concretes, quan fer-ho aporta més claredat als conceptes exposats. Majoritàriament es posa un èmfasi especial en el primer rol: el del responsable de dissenyar una aplicació informàtica abans que s'implementi. Actua de la mateixa manera que un dissenyador de maquinària, que fa el plànol de la màquina abans de començar a muntar-la, o que un arquitecte, que dibuixa el plànol d'un gratacels abans de ni tan sols començar a fer-ne els fonaments o d'apilar maons. És important seguir l'estudi de l'orientació a objectes des del punt de vista del dissenyador de programari per poder aplicar els coneixements adquirits sense lligar-nos a cap llenguatge de programació concret, de manera que siguin aplicables a qualsevol.

L'apartat "Fonaments de la programació orientada a objectes " serveix com a punt de partida, s'introdueix quins són els motius que han dut a l'aparició de l'orientació a objectes i es mostra com funciona aquesta aproximació amb vista a resoldre problemes, en general, i a generar aplicacions informàtiques específiques. Un cop establerta la motivació, s'aprofundeix en la comprensió del procés per traslladar qualsevol problema a resoldre a una aproximació orientada a objectes. Es descriuen els passos necessaris per aconseguir-ho, quines pautes bàsiques cal seguir i quines implicacions té cada decisió a l'hora de generar el disseny de l'aplicació. Finalment, es descriu de quina manera s'estructura una aplicació orientada a objectes i quina és la relació entre els components que es desenvolupen, les classes, i els elements que hi ha a memòria quan s'executen, els objectes.

Per tal de dur a la pràctica tots els conceptes teòrics, a l'apartat "Declaració de classes" s'explica com dur a terme el codi font Java de la unitat fonamental de tot programa orientat a objectes: la classe. Mitjançant el codi de les classes, és possible definir amb precisió quin és el comportament i les dades que contenen els diferents objectes que interactuen en una aplicació orientada a objectes.

Al llarg de la unitat didàctica se seguirà tot un conjunt d'exemples que permeten fer més fàcil la comprensió dels conceptes exposats. D'altra banda, per treballar-ne els continguts, és convenient anar fent les activitats i els exercicis d'autoavaluació, a més de consultar la bibliografia bàsica proposada.

Resultats d'aprenentatge

En finalitzar aquesta unitat l'alumne/a:

1. Escriu i prova programes senzills, reconeixent i aplicant els fonaments de la programació orientada a objectes.

- Instancia objectes a partir de classes predefinides.
- Utilitza mètodes i propietats dels objectes. .
- Escriu crides a mètodes estàtics.
- Utilitza paràmetres a la crida a mètodes.
- Incorpora i utilitzat llibreries d'objectes.
- Utilitza constructors.
- Distingeix dades estàtiques de dades dinàmiques.
- Reconeix els mecanismes de destrucció i/o finalització d'objectes.
- Reconeix els mecanismes d'alliberament de memòria.
- Utilitza l'entorn integrat de desenvolupament en la creació i compilació de programes simples.

2. Desenvolupa programes organitzats en classes analitzant i aplicant els principis de la programació orientada a objectes.

- Reconeix la sintaxi, estructura i components típics d'una classe.
- Defineix classes.
- Defineix propietats i mètodes.
- Crea constructors.
- Crea destructors i/o mètodes de finalització.
- Desenvolupa programes que instancien i utilitzen objectes de les classes creades anteriorment.
- Utilitza mecanismes per controlar la visibilitat de les classes i dels seus membres.
- Defineix i utilitza classes heretades.
- Crea i utilitza mètodes estàtics.
- Crea i utilitza conjunts i llibreries de classes.

1. Fonaments de la programació orientada a objectes

Ens trobem en el punt de partida per veure com s'ha d'enfocar qualsevol problema, si bé amb un èmfasi especial en el desenvolupament d'aplicacions, des de la perspectiva de l'orientació a objectes. Se sol dir que per entendre com funciona l'orientació a objectes és necessari fer un "canvi de xip", ja que representa un salt cap a un nivell d'abstracció superior. La millor manera d'assolir aquest canvi és entenent els motius pels quals va sorgir i les estratègies bàsiques a seguir per aplicar de manera senzilla l'orientació a objectes. Un cop introduïts aquests motius i estratègies, ja és possible començar a pensar com és el món vist des d'aquesta perspectiva.

1.1 Un món orientat a objectes

A mesura que els sistemes s'han fet més complexos, ha estat necessari crear nous mecanismes que permetin usar un nivell cada cop més alt d'abstracció en el procés de desenvolupament del programari. Amb aquesta abstracció, el que s'intenta és, per una banda, dissenyar i desenvolupar programes sense haver d'estar lligats al maquinari o el sistema operatiu final de la màquina i, per l'altra, que siguin fàcils de mantenir i entendre per qualsevol desenvolupador. Aquest fet és imprescindible a mesura que programar s'ha convertit en un treball en equip. Una manera d'assolir-ho és que la tasca de programar sigui cada cop més propera al procés del pensament i del llenguatge humà.

Per començar des del cas més extrem, a l'hora de crear un programa es pot pensar en el codi màquina, les cadenes de 0 i 1 que s'executen directament dins del maquinari. Cada cadena codifica una operació molt simple, estretament vinculada al maquinari: gestionar memòria, fer una operació matemàtica bàsica en una unitat aritmètica, etc. Evidentment, avui en dia, a pràcticament ningú no se li acudiria fer un programa directament d'aquesta manera.

Immediatament, un pas abans, trobem el llenguatge assemblador, si bé aquest no deixa de ser un conjunt de mnemotècnics que reemplacen els 0 i els 1 del codi màquina amb sentències més intel·ligibles per als humans. Tot i que l'assemblador encara s'utilitza, i és molt important en alguns entorns, ja es pot veure que només un expert podria fer un programa de certa complexitat (per exemple, un navegador web) íntegrament amb aquest llenguatge. En tot cas, fer-lo implicaria invertir molt de temps i resultaria en una dificultat enorme per depurar-lo, ampliar-lo en el futur o perquè algú altre el pogués entendre.

Fent un salt una mica més llarg en el procés d'abstracció, i ja entrant dins del dia a dia d'un programador actual, trobem els llenguatges estructurats i la programació

modular. És el cas, per exemple, dels llenguatges C, Pascal o Basic, juntament amb les seves biblioteques, on hi ha sentències que ens permeten fer des d'operacions matemàtiques simples fins a imprimir directament cadenes de text per pantalla. Ja no cal conèixer tots els detalls del maquinari i és possible fer tasques complexes de manera senzilla. Tot i així, un programador que utilitzi aquestes eines encara ha de conèixer alguns aspectes lligats a la màquina o el sistema operatiu: com funciona la memòria d'un ordinador, per definir estructures de dades, o els mecanismes d'entrada i sortida, per accedir als fitxers, el teclat o la pantalla.

En aquests materials s'arriba fins al salt immediat següent: l'orientació a objectes. És aquí on es comença a fer una veritable aproximació entre la manera com un ésser humà pot estructurar un problema i com codificarlo en forma de programa, de manera que ambdós processos siguin, conceptualment, tan semblants com sigui possible. Un aspecte molt important en què cal fer èmfasi, ja des del començament, és que l'orientació a objectes com a tal és una metodologia, o estratègia, amb vista al desenvolupament del programari. No es tracta simplement d'una família de llenguatges de programació. Els anomenats *llenguatges de programació orientats a objectes* són els que suporten aquesta metodologia i ens permeten plasmar el disseny generat en forma de programa.

L'orientació a objectes no és un tipus de llenguatge de programació. És una metodologia de treball per crear programes.

Un dels aspectes més importants de l'orientació a objectes és l'anàlisi dels problemes que volem resoldre mitjançant aplicacions informàtiques de la mateixa manera que per als problemes que hi ha al món real. La conseqüència directa d'aquest fet és la possibilitat d'aplicar mecanismes formals d'enginyeria al programari sense la necessitat d'implementar primer la solució. Això, d'una banda, permet treballar de la mateixa manera que altres disciplines afins a l'enginyeria ho fan dins el seu àmbit (mecànica, electrònica, arquitectura, etc.). D'altra banda, permet dissenyar l'aplicació de manera totalment independent del llenguatge de programació que s'utilitzarà. De fet, el dissenyador ni tan sols no ha de saber programar necessàriament (tot i que saber-ne ajuda, és clar).

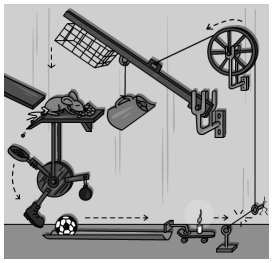
1.1.1 Gènesi i evolució de l'orientació a objectes

En els primers passos de l'aprenentatge d'una matèria és habitual donar una visió històrica de la seva evolució al llarg dels anys. Si bé normalment aquest fet és justificat per l'obtenció d'un cert grau de cultura general, en aquest cas, aquesta visió aporta una bona perspectiva del motiu que va dur a l'aparició de la metodologia de l'orientació a objectes, quins objectius persegueix i com enfoca el procés de desenvolupament d'una aplicació. Per fer-ho, però, n'hi ha prou de centrar-se només en els llenguatges més significatius, no cal una revisió exhaustiva. Aquest repàs històric també servirà per introduir ja, però encara sense aprofundir-hi gaire, alguns dels conceptes clau de l'orientació a objectes.

Els inicis de l'orientació a objectes es remunten a l'any 1960, amb Ole-Johan Dahl i Kristen Nygaard, que treballaven en el Centre de Computació de Noruega en la creació de simulacions de maquinària complexa. Un dels problemes més grans en la seva tasca era la traducció de l'esquema de la maquinària al llenguatge informàtic, ja que implicava un salt conceptual molt gran amb les eines existents: per modelar la màquina calia ser un expert en mecànica, electricitat, etc., mentre que per generar un programa calia ser expert en ordinadors.

Per tal de minimitzar aquest salt, van decidir utilitzar una nova estratègia: dividir els programes en mòduls independents, cadascun dels quals representaria, únicament i exclusivament, un símil exacte de cadascun dels tipus de peça de la maquinària a simular. En cada mòdul s'especificaria quines eren les seves propietats generals (per exemple, l'amplada, el pes, etc.) i el seu comportament (per exemple, pot ser pitjada, genera un cert corrent elèctric, etc.). Aquests mòduls actuarien com un motlle en el món real, i a partir d'ells es generarien els elements, les peces, que formarien realment el programa en execució. L'execució del programa, pròpiament, vindria donada per la interacció entre els diferents elements, de la mateixa manera que les peces que componen una màquina en el món real ho farien per fer-la funcionar en la seva totalitat.

Aquesta estratègia tenia un avantatge addicional: un cop desenvolupat un mòdul que definia un tipus de peça concret, aquest es podia tornar a aprofitar per simular qualsevol altra màquina que fes ús del mateix tipus de pe- ça, fet que era molt habitual.



En un programa orientat a objectes, els elements interactuen per resoldre una tasca.

Per aplicar aquesta filosofia a l'hora de generar un programa van necessitar desenvolupar un nou llenguatge que acomodés totes aquestes propietats. Així van néixer els primers llenguatges orientats a objectes: el **Simula I**(1962- 1965) i el **Simula 67**(1967). En aquests llenguatges ja es va establir part de la nomenclatura més important de l'orientació a objectes. Els mòduls que defineixen els tipus de peça s'anomenarien **classes**, i per referir-se a les peces concretes que s'executarien dins un programa s'empraria el terme **objectes**.

> Un **programa** s'entén com una simulació d'un escenari del món real, en què un conjunt d'elements, els objectes, interactuen entre ells per dur a terme la tasca que es vol resoldre.

Tot i que els llenguatges Simula es poden considerar els originadors del concepte de programació orientada a objectes, va ser l'empresa Xerox, amb el seu llenguatge **SmallTalk**, la que va començar a utilitzar aquest nom al principi de la dècada de 1970. Alan Kay, de la Universitat de Utah, va conèixer l'obra de Dahl i Nygaard i va aplicar aquesta nova visió per afrontar un problema de programació molt concret: la creació d'un ordinador personal especialment dissenyat per suportar aplicacions gràfiques. En aquest cas, va considerar que una interfície d'usuari també es podia interpretar com un conjunt d'elements comuns o peces amb unes propietats clarament definides (botons, finestres, menús, etc.) que porten a terme una tasca en interaccionar entre ells o amb l'usuari. Kay va vendre la seva idea a Xerox, que va desenvolupar el llenguatge SmallTalk per crear aquest ordinador totalment gràfic, batejat com a Dynabook.

El llenguatge SmallTalk va aportar millores noves molt importants a la idea inicial. Per una banda, permetia la possibilitat que els objectes tinguessin un comportament dinàmic: ser creats, destruïts o que les seves propietats poguessin canviar al llarg de l'execució del programa. Aquesta evolució es pot considerar lògica en un salt des de la simulació d'una màquina, en què les peces sempre són les mateixes, a una interfície gràfica, en què els elements són totalment dinàmics: les finestres s'obren i tanquen o canvien de mida, els botons s'habiliten, inhabiliten o canvien les icones que els representen, etc. Va ser justament aquesta evolució la que va catapultar l'orientació a objectes com a metodologia ideal per desenvolupar interfícies gràfiques.

La popularitat del C++ s'ha mantingut fins avui en dia i encara és molt utilitzat.

Una altra aportació molt important, i que actualment es considera bàsica en qualsevol llenguatge orientat a objectes, va ser la introducció del concepte d'**herència**: poder definir diferents tipus d'objecte, diferents classes, només especificant les diferències que hi ha entre ells.

Arribats a aquest punt, en què es poden considerar establertes les bases de l'orientació a objectes, aquesta metodologia va començar a guanyar impuls. Aquest impuls es va veure especialment reflectit a partir de la dècada de 1980, quan un seguit de llenguatges no orientats a objectes molt populars com **BASIC, Pascal o Fortran**, o bé van començar a incorporar aspectes de l'orientació a objectes, o bé, a partir d'ells, es va generar una versió orientada a objectes. El màxim exponent de l'època, tant per la popularitat com per la complexitat a l'hora de fer programes, va ser el llenguatge C++, creat a partir del llenguatge C per Bjorn Stroustrup. Entre les noves aportacions de C++ als llenguatges orientats a objectes es pot comptar l'**herència múltiple** , la capacitat d'un objecte per aplicar herència a partir de més d'una classe.

Compile once, Run everywhere

Malauradament, alguns dels programadors més experts d'aplicacions solen canviar la frase a "compile once, debug everywhere" (compila un cop, depura el codi a tot arreu).

En els anys 1990 l'orientació a objectes va arribar al seu moment de màxima popularitat amb l'aparició d'un nou llenguatge molt inspirat en C/C++, però que intentava disminuir-ne la complexitat a l'hora de programar: el**llenguatge Java**. Desenvolupat per SUN Microsystems i publicat en la seva versió 1.0 l'any 1995, una de les innovacions més importants que aportava era l'execució sobre una **màquina virtual**: els programes, en lloc d'executar-se directament sobre un maquinari o sistema operatiu específic, s'executen sobre una programa especial que crea una capa d'abstracció. D'aquesta manera, un programa generat per Java es pot executar sobre qualsevol plataforma, fet que en maximitza la portabilitat, però a costa d'una eficiència menor. Un dels seus eslògans va ser la frase *"compile once, run everywhere"* (compila un cop, executa a tot arreu).

Java 7

L'any 2012, la darrera versió publicada de Java és la 7, actualització 5, amb la particularitat que es tracta principalment de programari lliure en la seva major part.

Aquesta possibilitat va permetre enfocar el llenguatge Java a la programació d'aplicacions a Internet, en què hom es troba en un entorn de sistemes heterogenis. La incorporació de la màquina virtual de Java a tots els navegadors més populars i la possibilitat d'executar programes en Java des d'aquests (els anomenats *applets*), lligada a l'explosió d'Internet, van ser uns dels factors principals de la seva popularitat.

Els motius de la popularitat de Java, tan lligats a la idea d'una màquina virtual i un llenguatge enfocat a desenvolupar aplicacions a Internet, no van passar gens desapercebuts dins el mercat i, l'any 2002, l'empresa Microsoft va contraatacar

publicant la versió 1.0 de l'entorn .NET, amb la seva pròpia proposta de màquina virtual. Si bé l'especificació de la plataforma es va desenvolupar de manera genèrica, la implementació publicada es va lligar exclusivament a PC amb sistemes operatius Windows. Entre els llenguatges per desenvolupar en aquest entorn hi ha el C# (C Sharp), creat exclusivament per al seu ús en aquest entorn. Queda obert per al debat fins a quin punt el C# es pot considerar més fortament inspirat en Java que no pas el seu llenguatge pare, el C++. Aquest es pot considerar un dels darrers llenguatges orientats a objectes que s'han creat i que encara és de certa rellevància.

Com diu la frase: "No hi ha elogi més gran que la imitació".

1.1.2 Bases de l'orientació a objectes

Durant el procés de creació del llenguatge SmallTalk, Alan Kay va definir les que es consideren les bases de l'orientació a objectes, les quals serveixen per establir com s'estructura la resolució d'un problema mitjançant l'orientació a objectes i de quina manera interactuen els diferents components per assolir una tasca concreta. Aquestes bases, compartides pels diferents llenguatges orientats a objectes, són les següents:

- Tot és un objecte, amb una identitat pròpia.
- Un programa és un conjunt d'objectes que interactuen entre ells.
- Un objecte pot estar format per altres objectes més simples.
- Cada objecte pertany a un tipus concret: una classe.
- Objectes del mateix tipus tenen un comportament idèntic.

La descripció més detallada del significat de cadascuna d'aquestes bases servirà com a fil argumental per explicar amb més claredat en què consisteix la metodologia de l'orientació a objectes. Amb vista a fer més entenedores les explicacions, en totes s'usa un exemple comú basat en un escenari en què sigui fàcil identificar els elements que el componen, però que a la vegada tingui sentit plasmar-lo en forma d'una aplicació informàtica: el joc del Monopoly.

El joc del Monopoly: descripció

En el joc del Monopoly, els jugadors gestionen béns i immobles amb l'objectiu final d'arruïnar la resta d'adversaris. Cada jugador disposa d'uns diners inicials, en forma de bitllets, i és representat per una fitxa al tauler. Aquesta fitxa avança d'acord amb el resultat de la tirada de dos daus, per caselles amb noms de carrers i d'un color concret, que representen propietats. Cada cop que es cau en una casella, el jugador pot optar per comprar-la pagant el preu que marca la casella. En fer-ho, rep un títol de propietat. Quan un jugador cau en una casella que és propietat d'un altre jugador, ha de pagar al jugador propietari la quantitat de diners especificada en el títol de propietat. Dins aquesta dinàmica, hi ha un jugador especial anomenat *banca* que exerceix d'àrbitre del joc (no té fitxa) i gestiona els bitllets i les propietats que encara no pertanyen a cap jugador.

Quan un jugador és el propietari de totes les caselles d'un mateix color al tauler, cada cop que hi cau té l'opció d'edificar fins a quatre cases. Quan ha construït quatre cases, pot

edificar-hi un hotel. Per fer-ho, ha de pagar una certa quantitat de diners, però a partir de llavors, el preu que han de pagar els adversaris que caiguin en aquesta casella també és més alt. Les edificacions es representen amb peces que se situen sobre la casella.

Si bé aquesta és la descripció bàsica, cal dir que en realitat el joc és més complex, ja que al llarg del joc també hi ha un seguit de circumstàncies especials: robar cartes que donen premis o càstigs, una casella de presó en què la fitxa queda atrapada fins a treure doble número, etc. De totes maneres, per seguir l'exemple no és necessari conèixer tots aquests detalls, és suficient amb el que s'ha explicat.

Els jocs de taula són una bona elecció per practicar com es pot aplicar l'orientació a objectes.

És molt important remarcar que l'explicació es mantindrà en els aspectes conceptuals: quina és l'estratègia que usa l'orientació a objectes per estructurar un problema.

1) Tot és un objecte, amb una identitat pròpia. Aplicar l'orientació a objectes a un programa és equivalent a intentar crear la simulació d'un escenari que podríem tenir en el món real, però dins de l'ordinador. Aquesta simulació s'estructura en un conjunt d'elements, cadascun del quals té unes propietats i un comportament concret que intenten imitar les de l'element del món real que representen. Aquests elements dins la simulació s'anomenen **objectes**. Així, doncs, en el programa en execució, tot element serà sempre un objecte.

Quan parlem de les **propietats d'un objecte** ens referim a les qualitats que es considera important quantificar i que defineixen l'aspecte o l'estat de l'objecte. Així, doncs, donat un botó, podem decidir definir quin és el seu color, la seva mida, etc.

Les propietats d'un objecte s'anomenen formalment els seus **atributs**.

Per ara, els atributs dins un objectes els expressarem amb aquesta representació.

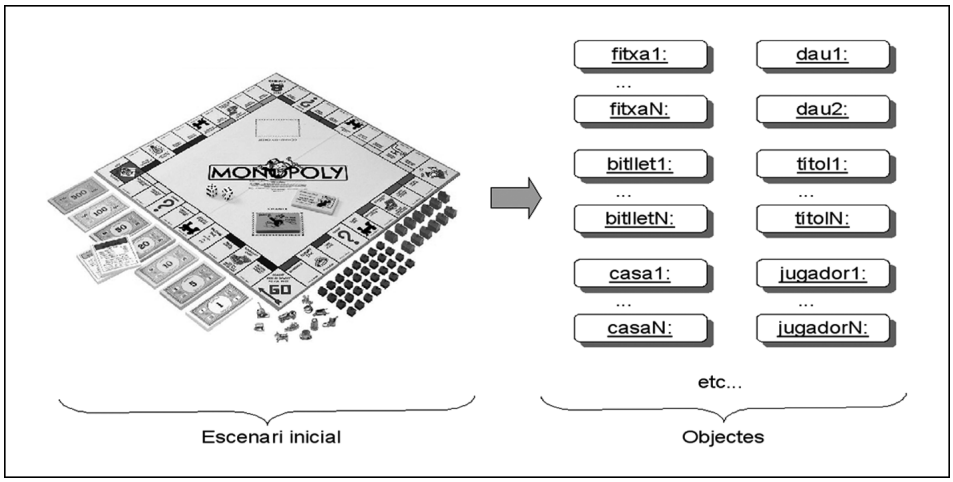
Un altre fet important és que cada objecte dins un programa és únic, tot i que n'hi pot haver més d'un amb propietats idèntiques, i clarament identificable dins aquesta simulació, de la mateixa manera que dues peces dins una màquina (per exemple, dos botons) poden ser idèntiques, però són clarament dos elements diferenciats situats en ubicacions físiques diferents i amb funcions diferents.

El joc del Monopoly: descomposició en objectes

Dins del joc del Monopoly, cada objecte correspondria a un element que podem veure al llarg d'una partida: el tauler, els daus, cada jugador, les fitxes que es mouen pel tauler, els títols de propietat, les targetes, les cases i els hotels, els bitllets, etc. Alguns d'aquests objectes són individuals (només hi ha un tauler) i d'altres n'hi ha diversos, amb atributs idèntics (els diversos bitllets de cent són tots iguals) o diferents (cada títol de propietat, que correspon a un carrer i té un valor diferent).

En la figura 1.1 es pot veure part d'una possible descomposició del joc del Monopoly en alguns dels objectes que el componen.

FIGURA 1.1. Tot es pot descompondre en objectes



En la translació representada en la figura 1.1, des del món real a un conjunt d'objectes, cal remarcar dues coses. Per una banda, cada element individual és representat per un objecte. Així, doncs, si tenim tretze fitxes d'hotel, també hi haurà tretze objectes hotel. Això està expressat per les diferents representacions tipus **objecte1:**, ..., **objecteN**: (per exemple, **fitxa1:**, ..., **fitxaN:** en la figura). Per altra banda, entre els objectes hi haurà qualsevol element que formi part i interactuï dins d'una partida del joc. Per tant, tot i que no apareix a la imatge, també cal tenir en compte elements com els jugadors o la banca.

Normalment, és l'usuari qui inicia cada cadena d'interaccions que du a la realització d'una tasca.

2) Un programa és un conjunt d'objectes que interactuen. De la mateixa manera que les màquines de Dahl i Nygaard es posaven en funcionament mitjançant la interacció de les diferents peces que les formaven, l'execució d'un programa vindrà donada pel conjunt d'interaccions entre els diferents objectes que el componen. Per exemple, podem interactuar amb un botó pitjant-lo. Aquest, a la vegada, interactuarà amb altres objectes (en el món real, potser amb una molla o enviant un senyal elèctric) per transmetre que cal executar una ordre donada.

El que defineix el comportament de cada objecte és una llista amb el conjunt de les interaccions que pot rebre, cada una sempre d'acord amb una tasca que pot fer o un canvi d'estat (una bombeta es pot apagar i encendre, un botó pot ser pitjat, una finestra tancada, etc.).

Cada interacció que pot rebre un objecte s'anomena **operació**.

Quan un objecte A vol interactuar amb un objecte B, diem que A **crida una operació** de B. Quina operació es crida depèn del tipus d'interacció que vol desencadenar A. Així, doncs, un programa en execució es compon d'un conjunt d'objectes que criden operacions entre ells.

El joc del Monopoly

Al llarg d'una partida del Monopoly hi ha un conjunt d'interaccions possibles entre els objectes, d'acord amb les regles del joc. Així, doncs, entre moltes altres coses, un jugador pot fer les interaccions amb altres objectes del joc descrites en la taula 1.1. Cada interacció equival a cridar una operació sobre l'objecte que la rep.

TAULA 1.1. Algunes interaccions que poden ser iniciades per un objecte jugador

Interacció sobre un altre objecte	**crida d'operació...**
Pagar un deute a un altre jugador	...*pagar* sobre un altre objecte jugador
Tirar els daus	...*tirar* sobre els dos objectes dau
Comprar propietat a la banca	...*comprar* sobre l'objecte banca

També és possible que no sigui un altre objecte, sinó l'usuari de l'aplicació, qui cridi una operació sobre un objecte. Aquest aspecte cal tenir-lo en compte quan es defineixen les operacions que pot rebre un objecte.

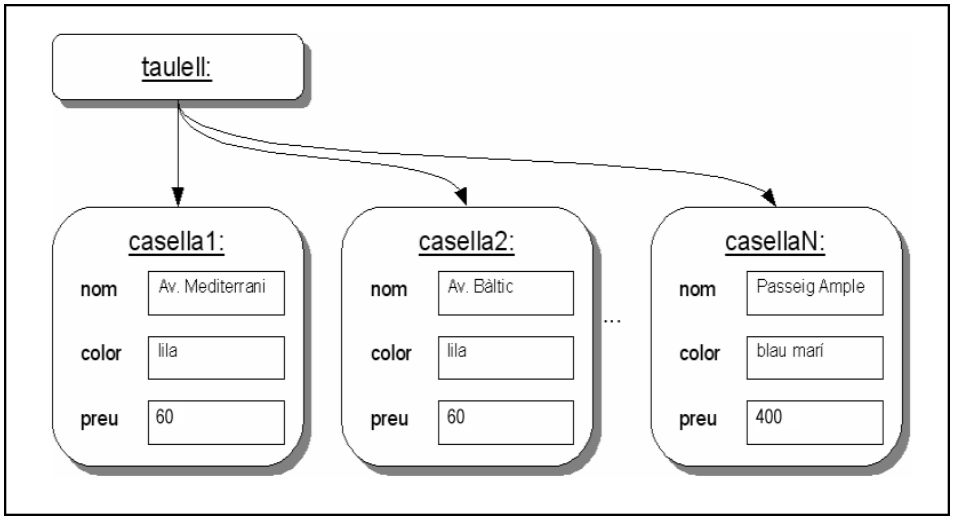
3) Un objecte es pot compondre d'altres objectes més simples. Quan es vol resoldre un problema, la manera més senzilla de tractar amb la complexitat és mitjançant la descomposició en problemes més simples. Per exemple, el motor d'un cotxe es descompon en elements més senzills i molt més fàcils d'analitzar, els quals, a la vegada, es poden descompondre en un altre conjunt d'elements encara més simples. Intentar copsar el funcionament d'un motor de cotxe sense aquesta descomposició sol ser molt més complicat.

Aquesta tècnica també es pot usar dins l'orientació a objectes, de manera que es poden crear objectes que en realitat es componen d'altres de més simples. L'objectiu és el mateix, és a dir, poder generar elements complexos a partir d'altres de més simples. Però també hi ha una altra motivació: la possibilitat d'interactuar directament tant amb l'objecte complex com amb qualsevol dels seus subelements.

El joc del Monopoly: objectes compostos

Al Monopoly, un objecte fàcilment identificable és el tauler. Tot i així, també es pot considerar que aquest en realitat està format per la unió d'altres subelements més senzills: les caselles. De fet, com es pot veure en la figura 1.2, en aquest escenari resulta especialment útil mirar les caselles com a objectes, ja que cada una té uns atributs que ens interessa diferenciar (color, preu, etc.).

FIGURA 1.2. Un objecte es pot compondre d'altres objectes



4) Cada objecte pertany a un tipus concret: una classe. A mesura que es defineixen els diferents objectes que componen el programa en execució, sovint

es troba que hi ha objectes que es poden considerar del mateix tipus: tenen exactament les mateixes propietats, tot i que el valor de cada una pot ser diferent per a cada objecte concret, i el mateix comportament. Llavors es diu que aquests objectes pertanyen a la mateixa classe.

Una **classe** és l'especificació formal de les propietats (els atributs) i el comportament esperat (la llista d'operacions) d'un conjunt d'objectes del mateix tipus, i que actua com una plantilla per generar cadascun d'ells.

Alguns objectes del programa compartiran la mateixa plantilla i d'altres en tindran una només per a ells. Pel que fa a la nomenclatura formal, es diu que un objecte és una **instància** d'una classe i que un objecte és **instanciat** quan es crea dins l'aplicació. És tot just en aquest moment quan s'usa la classe per generar l'objecte, tot determinant quin és el seu conjunt d'atributs i assignant un valor concret per a cada un. A efectes pràctics, quant a nomenclatura, es pot considerar que *instància* i *objecte* són termes equivalents.

Un cop **instanciat un objecte**, aquest sempre pertany a la mateixa classe. No es pot canviar el seu tipus dinàmicament.

Els objectes només existeixen mentre l'aplicació està en marxa.

Fins ara sempre ens havíem referit als objectes com a elements que componen un programa en execució (èmfasi en la paraula "execució"). Però, donat un programa desenvolupat mitjançant l'orientació a objectes, allò que el programador realment generarà, el codi font, serà, per una banda, les classes de cada objecte necessari dins el seu programa i, per l'altra, el codi que instancia i organitza tots els objectes.

Nomenclatura

Normalment, quan parlem d'objectes o instàncies, s'escriu amb la lletra inicial en minúscula. Quan parlem de classes ho fem amb la inicial en majúscula.

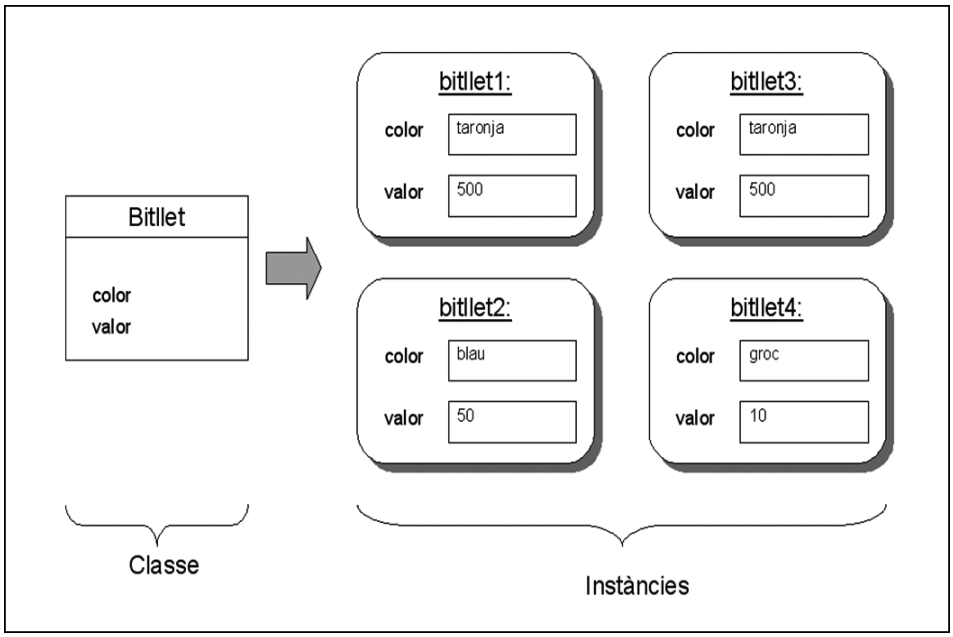
El joc del Monopoly: classes d'objectes

Fins ara s'han identificat els objectes que es diferencien clarament en una partida de Monopoly. Tot i així, ja es pot veure que hi ha objectes que comparteixen unes propietats i un comportament, encara que el seu valor concret sigui diferent en cada cas. Per exemple, tots els jugadors són el mateix tipus d'objecte (són el mateix, tot i que puguin tenir un valor diferent per a l'atribut nom). De la mateixa manera, podem detectar aquesta relació en els bitllets, els títols de propietat o les caselles del tauler, entre d'altres.

La conclusió d'aquest fet és que, en el codi font del programa, el desenvolupador ha de generar una classe per a cadascun d'aquests tipus d'objectes: la classe `Jugador`, la classe `Bitllet`, la classe `Títol`, la classe `Casella`, etc. Només hi haurà una classe per a cada tipus d'objecte, i dintre seu es definirà quins són els atributs i quines són les operacions que es poden cridar sobre els objectes d'aquest tipus.

Així, doncs, podem tenir quatre jugadors en una partida (en executar el programa), però el desenvolupador només haurà definit una única classe Jugador. En començar la partida, el codi del programa s'encarregarà d'instanciar quatre objectes jugador, a partir de la classe Jugador, cadascun amb els seus valors concrets per als seus atributs (un nom diferent per a cadascun). De la mateixa manera, a partir d'una única classe Bitllet generem tots els objectes bitllet que hi ha quan el programa està en execució, com es pot veure en la figura 1.3. Val la pena remarcar novament el fet que podem tenir objectes amb valors idèntics per als seus atributs (**bitllet1:** i **bitllet2:**), però tot i així, continuen essent entitats diferenciades.

FIGURA 1.3. Tots els objectes són una instància d'una classe



Sempre que calgui referir-se a una classe en un text o esquema, s'usarà el seu nom. En contrast, quan vulguem referir-nos a una instància d'aquesta classe, un objecte, usarem la nomenclatura:

nomObjecte:NomClasse

Nom compost

Si s'usa un nom compost, s'utilitza majúscula en cada inicial de paraula, si bé, se sol evitar usar articles o preposicions en el nom. Per exemple, `TítolPropietat`.

Per exemple: **bitllet1:Bitllet**, **jugador4:Jugador**, etc. Tot i així, si no es vol concretar la classe, perquè ja és evident pel context, no cal. Quant a la nomenclatura, en el cas dels objectes, la primera inicial sempre sol ser en minúscula. En canvi, per a les classes sempre se sol usar majúscula. Això ens permet diferenciar fàcilment quan s'està fent referència a una classe o a un objecte concret.

Si no es vol donar un identificador específic a un objecte en referir-nos-hi, es pot usar simplement la nomenclatura següent per referir-se a un objecte qualsevol d'una classe determinada:

:NomClasse

Atès qualsevol esquema o descripció, cada identificador d'objecte és únic (tot és un objecte, amb identitat pròpia). Si usem el mateix identificador diverses vegades, en cada una es considerarà que ens referim exactament al mateix objecte.

5) Els objectes del mateix tipus tenen un comportament idèntic. Finalment, arribem a la darrera base de l'orientació a objectes, tot i que aquesta es pot considerar una extensió de l'anterior. De la mateixa manera que una classe defineix els atributs d'objectes del mateix tipus, també n'especifica el comportament: la seva llista d'operacions. Per tant, el conjunt d'interaccions possibles amb objectes de la mateixa classe sempre és el mateix. Per cada interacció que un objecte de la classe A pot rebre (sense que importi el possible objecte origen), caldrà definir una operació associada a aquesta en especificar la classe A.

Una **operació** és una funció o transformació que es pot aplicar a tots els objectes d'una classe.

1.2 Resolució de problemes usant orientació a objectes

L'orientació a objectes aporta una nova perspectiva a la visualització i la resolució de problemes. Per aplicar-la correctament és necessari seguir alguns passos per tal de plantejar el problema i anar fent la descomposició de l'escenari. Per seguir aquests passos, val la pena tenir una idea de què compon realment un objecte i una classe, com es poden especificar classes i quins aspectes cal tenir en compte a l'hora de fer-ho. Un programa orientat a objectes es composarà, al igual que una màquina, de la unió de cadascuna de les seves peces (les classes), que interactuant, faran que el tot funcioni. Una eina útil per visualitzar com, al final del procés, tot un conjunt de classes interactuen per tal de formar una aplicació són els mapes d'objectes.

1.2.1 Esquema general d'aplicació de l'orientació a objectes

Reaprofitament

Dins l'orientació a objectes és molt útil que les classes definides per a un programa concret també puguin ser usades per altres programes que es puguin fer en el futur.

Un cop s'han establert les bases de l'orientació a objectes, és possible aproximar-nos de manera general a la resolució d'un problema mitjançant aquesta metodologia. Seguint un conjunt de passos ordenats, és possible establir de quina manera cal organitzar els seus components i els seus mecanismes de col·laboració. Aquests passos són els següents:

1. Plantejar l'escenari descriptivament, amb llenguatge humà. Com més detallada sigui la descripció, més fàcil serà la feina.

2. Localitzar, dins la descripció de l'escenari, els elements que es consideren més importants: els que realment interactuen amb vista a resoldre el problema. Aquests seran els objectes del programa. Normalment, solen ser substantius dins la descripció.

3. Considerar quins elements són d'una certa complexitat. Redefinir-los com a agrupacions d'objectes més simples. Una bona estratègia és partir del fet que tot l'escenari en si és un objecte complex (igual que una màquina també és un objecte complex) i anar-lo descomponent en parts més petites.

4. Agrupar els diferents objectes segons el tipus: quins objectes veiem que tenen propietats o comportaments idèntics. Cada tipus d'objecte serà una classe que caldrà especificar.

5. Identificar i enumerar les característiques dels objectes de cada classe: quines són les seves propietats (els atributs) i el seu comportament (les

operacions que ofereixen). N'hi ha prou amb una llista general, escrita en llenguatge humà però suficientment entenedora.

6. Establir les relacions que hi ha entre els objectes de les diferents classes a partir del paper que interpreten en el problema general. Els objectes no es generen en un buit, sinó que estan relacionats entre si, de la mateixa manera que les peces d'una màquina o els elements d'un edifici no floten en l'aire, sinó que estan connectat per formar un tot. De la mateixa manera, un cop identificats els objectes que conformen el problema a resoldre (el programa que es vol fer en aquest cas), cal identificar com es relacionen entre ells. Normalment, aquesta mena d'enllaços es poden identificar com "aquest objecte en conté d'aquests altres" o "aquest objecte en gestiona aquests altres". A mode d'ajut, es pot generar un **mapa d'objectes**.

7. Per cada classe, especificar formalment els seus atributs i operacions, extrets a partir de la llista de propietats dels seus objectes dels punts 5 i 6. Normalment, especificar-ne els atributs és un procés més immediat que l'especificació de les operacions.

Per resoldre un problema complex, dividiu i vencereu.

Per a un problema de certa complexitat, és molt difícil identificar a la primera totes les classes, atributs i operacions. Normalment, caldrà fer diverses iteracions. Usar una aproximació incremental per a tot el procés no és una mala idea, ja que per a problemes complexos és impossible copsar tots els detalls a la vegada. De fet, moltes vegades no serà possible adonar-se que calen certes coses fins al moment de la implementació. Això no invalida la idea que, abans de començar a implementar el programa, hi ha d'haver una fase prèvia de descomposició del problema i de definició formal dels components a desenvolupar.

El codi font d'un programa fet amb un llenguatge orientat a objectes es compon principalment de les classes definides.

Un cop definida la descomposició del problema, i arribada l'hora de la implementació, una part important del nostre programa és el conjunt de classes que haurem especificat al pas 7. L'altra part és l'encarregada de posar tot el procés en marxa: crear els diferents objectes en la memòria de l'ordinador, estructurar-los seguint com a model els mapes d'objectes de l'apartat 6 i iniciar la cadena d'interaccions entre ells que conformarà el programa en execució.

Un aspecte de vital importància en aquest procés és que cal evitar pensar en qualsevol interfície d'usuari concreta. Si bé es pot assumir que hi ha mecanismes mitjançant els quals l'usuari podrà cridar les operacions de certs objectes o veure'n l'estat, per cap concepte cal establir quin és el mecanisme específic que s'usarà. El motiu principal per fer-ho és que, novament, la manera com es faran aquestes accions està molt vinculada al llenguatge de programació concret que s'ha utilitzat. En conseqüència, en obviar la interfície a l'hora d'especificar les classes, s'evita lligar el disseny a un llenguatge, de manera que es pot aprofitar per a diferents implementacions.

La descomposició inicial d'un problema mitjançant l'orientació a objectes no ha d'explicitar mai la interfície d'usuari a emprar. Només es defineix la lògica interna del sistema i com s'estructura la informació a processar. Això és el que s'anomena el **model** de l'aplicació.

Posteriorment, un cop descompost el problema i ja triat el llenguatge de programació i quins mecanismes s'usarà per interactuar amb l'aplicació, es pot fer un disseny específic a part per a la interfície d'usuari.

1.2.2 Exemples d'aproximacions orientades a objectes

La millor manera de donar una idea més aclaridora de com, partint del problema, es pot arribar als elements bàsics que accepten interaccions per resoldre'l és amb exemples específics que presentin la descomposició en objectes i classes. Com que només es vol donar una idea, el problema que es planteja en cada cas només es descriu de manera general, sense entrar a descriure en detall totes les funcionalitats. En tot cas, els problemes exposats no són merament didàctics, sinó que tenen sentit si es consideren com una aplicació informàtica a desenvolupar (si bé sense ser gaire complexos).

És possible que al seguir els exemples, en algun moment, us trobeu que alguna decisió presa vosaltres la faríeu diferent. Potser a partir de la descripció del problema vosaltres identifiqueu altres conceptes com possibles classes o atributs. O, senzillament, el text amb el que descriureu el problema és totalment diferent als que es proposa aquí. Doncs bé, precisament una particularitat que val la pena destacar sempre que apliquem l'orientació a objectes és la importància de tenir present que cada problema pot tenir diferents solucions, i totes poden ser totalment correctes. Tot depèn de la subdivisió en objectes que plantegi el dissenyador, segons la seva manera d'enfocar el problema proposat. Per tant, la solució exposada en aquest apartat és una de les moltes que hi pot haver. De totes maneres, sempre hi ha elements molt evidents que molt probablement seran comuns, o molt semblants, a la majoria de solucions.

A fi de comptes, si s'aplica aquest principi a altres disciplines, donats dos arquitectes als que se'ls demana fer un edifici amb certes característiques, quina és la probabilitat de que els dos facin exactament la mateixa proposta de projecte? Però segur que al menys fonaments, parets i finestres n'hi haurà en els dos casos.

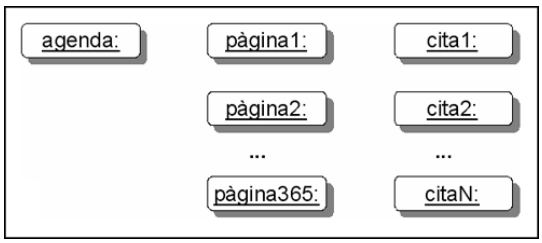
1) Una agenda. El primer exemple és molt senzill, amb molts pocs elements i funcionalitats i amb un paral·lelisme clar amb el món real per facilitar-ne la comprensió. Es vol dissenyar una agenda que permeti consultar les dates d'un calendari per a un any concret i apuntar cites a unes hores concretes. En aquest cas, és possible fer un cert paral·lelisme amb el món real, ja que el concepte d'agenda hi existeix. Es pot pensar en l'agenda com un llibre en què es van passant pàgines endavant o endarrere, cadascuna de les quals correspon a un dia. En cada pàgina es poden escriure cites establertes per a unes hores d'inici i de finalització determinades. Aquesta descripció en llenguatge humà seria la que s'ha descrit en el pas 1 de l'esquema d'aplicació de l'orientació a objectes.

Una bona manera de descompondre un problema en objectes és partir de l'element més general i, a partir d'aquí, anar extraient els elements més senzills. Així, doncs, en aquest cas es pot partir d'un objecte `agenda:` i deduir els elements que el

Pel que fa al disseny, per referir-se a un objecte s'usa el seu nom seguit de dos punts.

componen: les pàgines. Sobre aquesta base, els objectes poden ser els que es mostren en la figura 1.4.

FIGURA 1.4. Objectes d'una agenda



Hi ha objectes per als quals només podem fitar-ne el nombre per a un instant concret del programa. El seu nombre pot anar variant.

Hi ha 365 objectes `pàgina:`, ja que s'ha decidit que l'agenda és per a un any (per a aquest exercici s'obviaran els anys de traspàs). Cal remarcar que el nombre d'objectes cita no es pot prefixar per endavant, ja que serà dinàmic i pot ser diferent per a cada instant de l'execució de l'aplicació. Així, en iniciar l'aplicació no n'hi haurà cap i a mesura que avanci se n'hi afegiran, és a dir, s'aniran instanciant objectes `cita:`.

Un cop identificats els objectes ja es poden establir les classes que compondran el programa. En aquest cas, és relativament fàcil veure que són tres: `Agenda`, `Pàgina` i `Cita`. Alguns atributs que es poden considerar per a aquestes classes són:

- `Agenda`: any.
- `Pàgina`: dia, mes, si és dia festiu.
- `Cita`: hora inici, hora finalització, motiu.

Amb vista a fer una primera aproximació a les operacions de cada classe, cal tenir molt clar què es vol resoldre i, per tant, què es pot fer amb cada objecte de cada classe. Algunes interaccions lògiques en aquest problema i, per tant, operacions a definir, poden ser:

- `Agenda`: passar pàgina endavant, passar pàgina endarrere.
- `Pàgina`: escriure cita, esborrar cita.
- `Cita`: escriure contingut.

La particularitat més importat d'aquest exemple, i el motiu pel qual val la pena haver-lo presentat, és que no hi ha interaccions directament entre objectes. Totes les interaccions seran donades per l'usuari.

2) Un reproductor multimèdia. En aquest exemple es presenta la descomposició d'un sistema de reproducció multimèdia (música, vídeos, etc.). La motivació pot ser crear una aplicació senzilla per a l'ordinador o generar el sistema de control d'un reproductor portàtil (un dispositiu físic). Aquest exemple és més complex que l'agenda i, per tant, es descriurà tot el procés amb molt més detall.

Dispositius digitals

Els dispositius digitals actuals estan controlats per processadors amb un programari encastat associat. Avui en dia tot és un petit ordinador.

L'objectiu principal d'aquest exemple és mostrar que, si bé és molt útil fer un símil amb el món real, mai no s'ha de perdre de vista que, en darrera instància, hi haurà un programa d'ordinador. Per tant, en aquest exemple ja es deixarà de banda fer un paral·lelisme exacte entre el món real i els objectes i ens centrarem en els elements que realment aporten alguna cosa al funcionament del sistema.

La conseqüència directa és que les peces que componen el programa poden ser tan abstractes com es vulgui: no s'ha de fer un símil peça a peça, com ara objecte `cargol:`, objecte `tapa:` o objecte `cableAltaveus:`, etc. A més a més, tampoc no tenim les limitacions físiques del món real, per la qual cosa tampoc no caldrà un objecte `pila:` o `bateria:`, tot i existir en el món real.

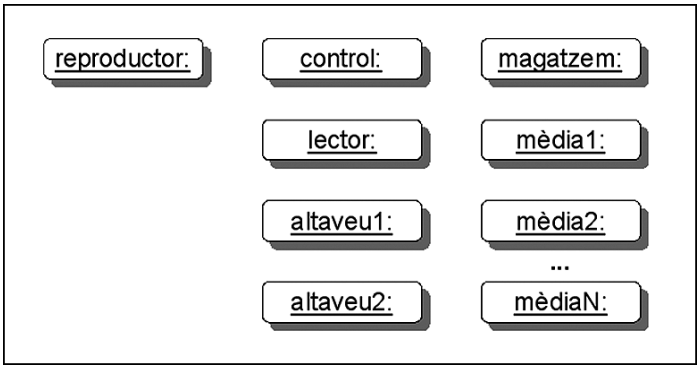
S'ha esmentat que una bona estratègia per dividir els problemes mitjançant l'orientació a objectes és partir de la base que tot és un únic objecte, i anar cercant subelements. Per tant, un dels objectes serà el `reproductor:`.

Ara cal cercar els subelements del `reproductor:` que fan que, en interaccionar, el sistema funcioni. Segons el pas 1 de l'esquema general d'aplicació de l'orientació a objectes, cal fer-se una idea clara de quin és el procés de reproducció d'una peça musical en llenguatge natural. Segons com es defineixi, els objectes que s'identificaran poden ser molt diferents. Es decideix fer-ho de la manera següent:

En el reproductor hi ha emmagatzemats fitxers multimèdia, el format dels quals no és important. Quan donem l'ordre de reproducció, el lector del sistema s'encarrega d'interpretar els **arxius multimèdia** i envia el resultat a l'**altaveu**, de manera que es poden escoltar. Les ordres es donen per mitjà d'un **tauler de control** (engegar, aturar, volum, etc.).

Seguint el pas 2 de l'esquema general, en aquesta descripció s'han identificat, subratllant-los, els elements que clarament són importants en el procés de reproducció multimèdia. Aquests apareixen llistats en la figura 1.5. Normalment, els substantius són bons llocs on començar la cerca.

FIGURA 1.5. Objectes del reproductor



En aquest cas, hi haurà molts objectes `mèdia:`, però de la resta d'objectes només n'hi haurà un o dos. Un cas especial són els objectes `altaveu:`. Si es vol l'opció de poder sentir en mono o estèreo, en caldran dos. Si no, amb un n'hi ha prou. Això ho decidirà qui dissenyi el sistema. Establir quins subelements componen el `reproductor:` satisfà el pas 3.

Arribats al pas 4 de l'esquema general, cal identificar les classes. Partint de la llista d'objectes es poden identificar les classes `Reproductor`, `Altaveu`, `PannellControl`, `Lector`, `Magatzem` i `Mèdia`. Immediatament ja es pot seguir amb els passos següents, per caracteritzar les propietats i el comportament dels objectes, definir els atributs i les operacions, i completar així el pas 5.

Alguns dels atributs possibles són els següents:

- `Mèdia`: dades, nom, artista, durada en segons, ubicació de les dades.
- `Control`: volum, marxa/pausa.
- `Lector`: mèdia en curs.
- `Magatzem`: peces de mèdia.

En aquest exemple són especialment importants les operacions, ja que es tracta d'un cas en què realment cal la col·laboració dels objectes per fer funcionar tot el sistema. Algunes d'aquestes operacions són:

- `Magatzem`: mèdia següent, afegir mèdia, esborrar mèdia.
- `Control`: apujar volum, abaixar volum, engegar, aturar, parar.
- `Lector`: reproduir mèdia, aturar, parar.
- `Altaveu`: generar so, establir volum.

Cal destacar que hi haurà classes amb molts atributs i poques operacions (`Mèdia`) i d'altres amb pocs atributs però més operacions (`Lector`).

En aquest exemple també val la pena veure que hi ha alguns objectes molt vinculats a l'entrada de dades, com l'objecte `panellControl:`, les operacions del qual es pot considerar que venen cridades directament per l'usuari. Altres objectes, en canvi, serviran com a mecanisme de sortida (`altaveu:`). Com es veu, això és fora de l'abast de la descomposició del problema, per la qual cosa no cal preocupar-se'n. Dependrà exclusivament del llenguatge de programació que s'utilitzi per implementar-ho tot plegat.

3) Una aplicació de gestió. Un exemple molt utilitzat en la descomposició de problemes mitjançant l'orientació a objectes és la creació d'aplicacions de gestió de dades: facturació, matriculació, control d'estocs, etc. El motiu principal és que són fàcils de visualitzar, ja que normalment els elements que es volen gestionar són evidents en la descripció del problema, no cal fer gaires interpretacions i les seves propietats (els atributs) pràcticament ja venen enumerats. El client que vol l'aplicació sol tenir molt clar quines són exactament les dades que vol emmagatzemar en el sistema. Un altre aspecte que en facilita la comprensió és el fet que, en tractar-se bàsicament de sistemes de manipulació de dades, la majoria d'operacions estaran vinculades a l'accés a aquestes dades (altes, baixes, modificacions, etc.).

En aquest apartat, se suposa una aplicació per gestionar el transport d'encàrrecs fins a les cases dels clients d'una franquícia. El primer pas serà explicar el problema en llenguatge humà i identificar els elements importants en què es pot descompondre:

Una empresa vol crear una aplicació que gestioni el transport d'**encàrrecs** d'una **sucursal** d'una franquícia. Cada sucursal té un conjunt de **transportistes** assignats, el nombre dels quals pot variar segons la grandària de la sucursal. Cada dia hi ha un transportista que no treballa, però es considera que està en reserva. Cada un disposa del seu propi vehicle, identificat per un número de llicència.

Quan un **client** vol fer un encàrrec, se n'enregistren les dades personals i aquest especifica les condicions de lliurament: dia i hora, adreça, etc. En l'encàrrec fa constar la llista de **productes** que vol que li serveixin. Tan bon punt es genera un encàrrec, automàticament ja s'assigna a algun transportista perquè el serveixi. Mai no hi ha encàrrecs sense transportista assignat.

Els clients també tenen l'opció de recomanar a amics seus perquè s'hi apuntin com a clients. Aquest fet es té en compte amb vista a algunes promocions o descomptes especials.

En aquest exemple ja s'obviarà el pas d'identificació dels diferents objectes i es passarà a la llista de classes, en ser un procés força immediat. De totes maneres, sí que val la pena mirar amb detall un cas molt concret, ja que segons la interpretació del dissenyador, el resultat és molt diferent: què és un producte en la descripció?

Per una banda, es pot considerar que un objecte `producte:` és estrictament un producte físic. Si en estoc hi ha cent unitats d'un producte, en el programa en execució hi haurà cent objectes `producte1:`, ...,`producte100:`, de la mateixa manera que si hi ha trenta clients donats d'alta, hi haurà trenta objectes `client1:`, ..., `client30:`.

Per altra banda, es pot interpretar que, quan es parla d'un producte, en realitat es refereix a un tipus de producte. A la sucursal hi ha un ventall de tipus de productes disponibles. Per tant, independentment de l'estoc, per a cada tipus de producte només hi ha un objecte `tipusProducte:` instanciat. En l'exemple s'usarà aquesta interpretació.

A continuació s'enumeren les classes identificades, amb alguns dels seus atributs que podrien ser més evidents. Alguns ja han estat llistats en la descripció del problema i d'altres no. En tot cas, serien fàcils d'identificar preguntant directament al client que vol l'aplicació. Es pot considerar la llista com un exemple, aplicant el sentit comú.

- `Encàrrec`: dia, mes, hora.
- `Sucursal`: nom, adreça postal, telèfon de contacte, adreça de correu electrònic.
- `Transportista`: nom, telèfon mòbil, número de llicència.
- `Client`: nom, adreça postal, telèfon de contacte, adreça de correu electrònic.

- `TipusProducte`: codi identificador, preu, estoc, si ja és a la venda.

Com ja s'ha esmentat, per a cada problema hi pot haver diverses solucions. Les classes que caldrà especificar poden variar d'acord amb les interpretacions del dissenyador. A continuació es mostren algunes interpretacions que podrien fer canviar la llista de classes:

- Es pot considerar que les dates de lliurament pròpiament són objectes (englobant hora, dia, mes i any) i que, per tant, hi ha una classe `Data`. Aquesta aproximació seria equivalent a descompondre un objecte `encàrrec:` en altres de més senzills, un dels quals és la data de lliurament. No seria incorrecte.

- També es pot interpretar que els vehicles són elements del problema, ja que s'esmenten explícitament en la descripció. En aquest cas, el número de llicència correspondria al vehicle, i s'hi poden afegir nous atributs (model, quilometratge, etc.). Aquesta aproximació tampoc no és incorrecta.

A l'hora d'identificar algunes de les operacions possibles, es pot veure que majoritàriament corresponen a la creació i la gestió de les dades emmagatzemades en el sistema:

- `Encàrrec`: modificar data, afegir producte, esborrar producte

- `Sucursal`: fer descansar transportista, alta de client, baixa de client

- `Transportista`: assignar encàrrec, esborrar encàrrec

- `Client`: modificar dades personals

- `TipusProducte`: modificar preu, modificar estoc

En aquesta llista d'operacions possibles es podrien trobar a faltar algunes operacions per modificar dades. Per què no és possible canviar les dades d'un producte, excepte el preu, però si les d'un client? La resposta és que només s'ha d'oferir aquesta mena d'operacions en els casos en què realment té sentit que canviïn unes dades. Un client pot canviar d'adreça, però un tipus de producte només canviarà de preu o l'estoc. Si canviés de nom, ja es podria considerar que és un producte diferent i, per tant, el que realment caldria fer és generar un nou objecte dins el sistema per a aquest nou tipus de producte; és a dir, instanciar un objecte `tipusProducte:` amb un valor concret per al seu atribut nom. De totes maneres, novament, és una decisió de disseny que pot variar segons qui resol el problema.

Tot i ser un problema relativament directe, la lliçó que es pot extreure d'aquest exemple és que en realitat molts aspectes varien segons les interpretacions del dissenyador, però aquestes decisions s'han de meditar suficientment, ja que afecten l'aplicació final.

1.2.3 Mapes d'objectes

Una eina útil per reflexionar sobre com els diferents objectes s'estructuren dins una aplicació és fer un esquema que representi alguns dels estats possibles dins l'aplicació al llarg de la seva execució, d'acord als objectes que hi participen.

En un **mapa d'objectes** es mostren tots els objectes instanciats i els enllaços que hi ha entre ells en un moment determinat de l'execució, de manera coherent amb el que s'espera de l'aplicació.

Cal dir que els mapes d'objectes només són una eina de suport, i no s'utilitzen com a mecanisme formal per representar el disseny d'una aplicació.

El primer que cal tenir present de cara a estructurar els objectes d'una aplicació és que, perquè dos objectes puguin interactuar entre ells durant l'execució d'un programa, han d'estar **enllaçats** (igual que dues peces dins una màquina han d'estar vinculades entre si d'alguna manera per poder interactuar). En cas contrari, la crida d'operacions no és possible. Quan, per mitjà d'un enllaç, un objecte `objecteA:` crida una operació sobre un objecte `objecteB:`, les transformacions fetes per l'operació únicament afectaran l'objecte `objecteB:`. No n'afectaran cap de la resta d'instàncies que hi hagi en aquell moment que pertanyin a la mateixa classe que l'`objecteB:`.

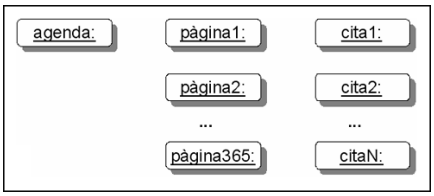
El mapa d'objectes indica de quina manera poden estar enllaçats els objectes identificats perquè l'aplicació funcioni i es consideri correcta (o, si més no, tingui sentit conceptualment). Ara bé, cal tenir molt present que, com que al llarg de l'execució d'una aplicació el nombre d'objectes de cada tipus pot anar variant, un mapa d'objectes és només un esquema dels objectes tal com podrien estar enllaçats en un instant concret de l'aplicació, i no una representació de com ho estan per sempre. En una agenda, en un moment donat, hi poden haver més o menys cites, o una data concreta pot tenir moltes cites o cap, i l'endemà pot variar totalment. Tot i això, aquest esquema és suficient per fer-se una idea de com s'estructuren dins la memòria de l'ordinador tots els objectes existents en un moment donat.

La millor manera de veure-ho és amb exemples.

1) Una agenda. Un objecte `agenda:` ha d'estar enllaçat amb els objectes `pàgina:` per poder gestionar-los. A més a més, també ha de saber explícitament quina pàgina està oberta, que seria la pàgina que es pot llegir en aquest moment. A part, cada pàgina és qui s'encarrega d'emmagatzemar les cites que conté. Un possible mapa d'objectes per a un cas concret durant l'execució de l'aplicació podria ser el que es veu a la figura 1.6:

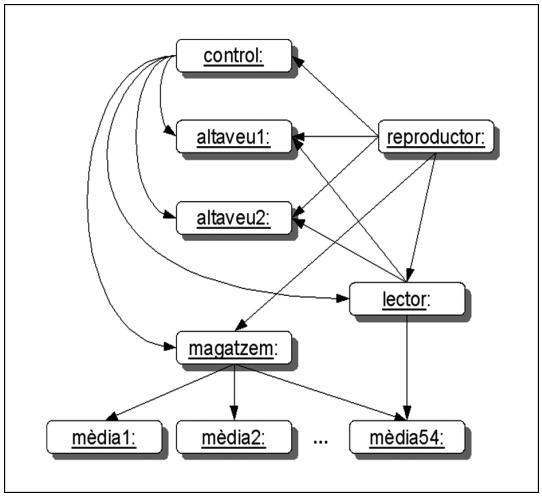
En aquest mapa, es representa un usuari que té un total de 128 cites apuntades a l'agenda. No té cap cita per a l'1 de gener (la primera pàgina de l'agenda), i en té dues per al dia 2 i el 3. El 31 de desembre té una altra cita.

FIGURA 1.6. Mapa d'objectes de l'agenda



2)Un reproductor multimèdia. Suposem que es decideix que el reproductor té dos altaveus, de manera que es pot controlar el mode mono o l'estèreo. Un possible mapa d'objectes vàlid seria el de la figura 1.7.

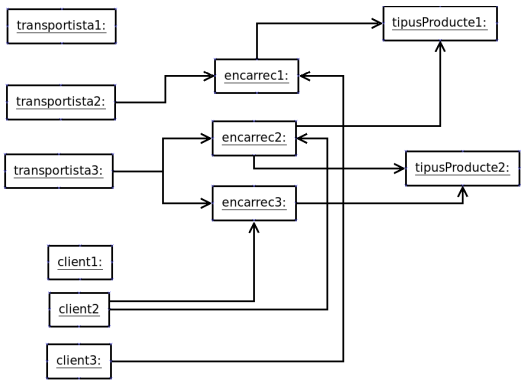
FIGURA 1.7. Mapa d'objectes del reproductor



L'objecte `reproductor` és el que controla tots els elements bàsics. En el que es refereix als objectes `media:`, els únics components que en fan alguna cosa són el magatzem i el lector. La resta no té sentit que els processin. Del mapa, també s'extreu que en aquest moment n'hi ha cinquanta-quatre i s'està reproduint la darrera.

3)Una aplicació de gestió. Un possible mapa d'objectes vàlid pot ser el que es mostra a la figura 8. Aquest ja té un cert grau de complexitat pel que fa als enllaços possibles, ja que hi ha moltes associacions i cardinalitats *.

FIGURA 1.8



Del mapa, se'n dedueix que s'ha volgut representar el cas on hi ha tres transportistes disponibles a la sucursal. El que no té assignat cap encàrrec és el de reserva. Un transportista té assignats dos encàrrecs, mentre que l'altre només en té assignat un. A part, la sucursal gestiona N clients i tipus de productes. Respecte els clients, es pot veure que dos de diferents, en els seus encàrrecs, demanen el mateix tipus de producte (`tipusProducte1:`). Curiosament, el client 2 demana el mateix producte en dos encàrrecs diferents (`tipusProducte2:`). El client 1 no té cap encàrrec pendent ara mateix i és qui va fer una recomanació al client 2.

1.3 Especificació completa de les classes

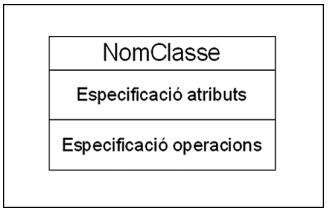
Com sempre, abans de saltar sobre el teclat cal fer una tasca prèvia de reflexió. Per aquest motiu, un cop es té una idea més o menys clara de quins objectes formaran part del vostre programa i com s'estructuren, és el moment de moure el focus a l'especificació formal de les classes de manera completa, amb tots els seus atributs i operacions. Per dur a terme aquesta especificació formal, normalment s'usa el llenguatge UML.

L'**UML** és un llenguatge estàndard que permet especificar amb notació gràfica programari orientat a objectes.

Mitjançant UML es poden representar molts aspectes diferents d'un programa orientat a objectes, però per ara ens conformarem amb l'especificació d'una classe.

En UML, una classe es representa en format complet mitjançant una caixa dividida horitzontalment en tres parts. La part superior compleix exactament la mateixa funció i té el mateix format que en el format simplificat, i s'estableix el nom de la classe. En la part del mig es defineixen els atributs que tindran les seves instàncies. Finalment, en la part inferior, es defineixen les operacions que es poden cridar sobre qualsevol de les seves instàncies. L'aspecte és el que es mostra en l'exemple de la figura 1.9.

FIGURA 1.9. Representació d'una classe en UML



En realitat, un atribut no és més que una variable.

1.3.1 Especificació d'atributs

Els atributs ens permeten especificar les propietats o l'estat dels objectes d'una classe. Abans d'especificar com es defineix formalment un atribut dins la declaració d'una classe, és un bon moment per veure exactament com es representa realment un objecte dins l'ordinador quan s'executa el programa.

> Dins un programa en execució, un **objecte** es pot considerar que no és més que un bloc de memòria, dins el qual es troben emmagatzemats tots els seus atributs.

En definir una classe, els atributs s'especifiquen segons la sintaxi següent:

```
1 visibilitat nomAtribut: tipus [= valor inicial]
```

Nomenclatura

Per a atributs s'usen paraules concatenades, en què la primera inicial està amb minúscula i la resta amb majúscula. Per exemple: `elMeuAtribut`.

El camp de valor inicial es correspon al valor que pren l'atribut en el moment d'instanciar un objecte d'aquesta classe. Concretar-lo en l'especificació dels atributs és opcional. Com veieu, el format es semblant a la declaració d'una variable qualsevol en Java.

Visibilitat dels atributs

Una característica específica de l'orientació a objectes és que per a cada atribut cal definir el que s'anomena *la seva visibilitat*. Aquesta és una propietat dels atributs que no existeix en la definició de tuples en altres llenguatges.

> La **visibilitat** d'un atribut indica si aquest és accessible directament des d'altres classes.

Hi ha diferents tipus de visibilitat, si bé es destacaran els dos més utilitzats: la visibilitat pública i la privada.

Normalment, els atributs es defineixen amb visibilitat privada.

L'UML no indica explícitament quin és el significat real de cada tipus de visibilitat, i deixa aquesta tasca a cada llenguatge de programació. El motiu és que aquest terme es refereix a l'accessibilitat a un objecte en l'àmbit del codi. Tot i així, es descriurà quina sol ser la seva interpretació en la majoria de llenguatges de programació orientats a objectes. Cada tipus de visibilitat s'identifica a la definició de l'atribut amb un símbol especial.

- Un **atribut públic** s'identifica amb el símbol "+". En aquest cas, si una instància `a:` està enllaçada amb una instància `b:`, `a:` pot accedir lliurement als valors emmagatzemats en els atributs de `b:`.
- Un **atribut privat** s'identifica amb el símbol "-". No es pot accedir a aquest atribut des d'altres objectes, independentment del fet que existeixi un enllaç o no. A efectes pràctics, és com si no existís fora de l'especificació de la

classe i, en conseqüència, només es pot utilitzar en les operacions dins de la mateixa classe en què s'ha definit.

En qualsevol cas, sigui quina en sigui la visibilitat, un objecte sempre té accés als seus propis atributs dins del seu codi.

Tipus dels atributs

El significat del camp de tipus de l'atribut no varia gaire respecte a la definició d'una variable normal i corrent en qualsevol llenguatge de programació: una manera d'especificar què representa la informació que conté. Pel que fa al disseny no hi ha un conjunt de tipus estàndard, hi ha la possibilitat de definir els que faci falta i tinguin un sentit dins el context del problema a resoldre.

De totes maneres, els tipus que normalment s'usen són els que mostra la taula 1.2.

Taula 1.2. Tipus bàsics dels atributs

Tipus	Significat	Exemple
Enter	Un nombre sense decimals	1, 56, 128, 15487
Real	Un nombre amb decimals	1,34, 3,2415, 267,14, 41,0
Caràcter	Una lletra	A, a, b, g, -, ?, ç
Booleà	Cert/fals	Cert, fals
Byte	Un byte	0x30, 0xA2, 0xFF
Matriu de... (...[ ])	Un conjunt d'elements...	[1, 2, 3], [a, b, c, f, g], [1,2, 3,0]

En el darrer cas, els tipus múltiples es poden especificar de dues maneres diferents, segons la interpretació que es vol expressar:

- enter[5] indica que hi ha exactament cinc enters.
- enter[0..5] indica que hi pot haver entre zero i cinc enters.

La definició d'atributs en UML obvia totalment els aspectes vinculats al llenguatge de programació o a l'arquitectura en què es desenvoluparà el programari. Per tant, no s'han d'usar mai tipus la principal característica dels quals sigui un aspecte de baix nivell, com ara la precisió o el nombre de bits de la seva representació, tal com passa en alguns tipus de dades en diversos llenguatges de programació.

A part dels tipus bàsics, quan s'empra l'orientació a objectes també és possible establir que un atribut és un objecte. Atès que el tipus d'un objecte ve donat per la classe, per fer-ho, en el camp tipus s'ha de posar el nom de la classe que descriu el tipus d'objecte que es vol usar com a atribut. Això permet usar atributs que contenen elements més complexos.

Partint d'aquest fet, quan s'especifiquen atributs es pot considerar que ja hi ha predefinides un conjunt de classes de propòsit general, que pràcticament tots els llenguatges suporten d'una manera o d'una altra:

- **La classe** `String`, que serveix per especificar tipus de dades que corresponen a cadenes de caràcters, així s'evita haver d'operar amb caràcters.

- **La classe** `List`, usada per especificar seqüències d'elements sense cap fita predeterminada. Aquesta classe pertany a una família especial de classes anomenades **classes parametritzades**. Aquesta denominació prové del fet que, quan es defineix, cal especificar un paràmetre addicional que indica el tipus d'elements amb què opera. Un cop definit aquest tipus, ja no pot canviar. Pel que fa a la notació, això es fa de la manera següent:

```
1 List<nomTipus>
```

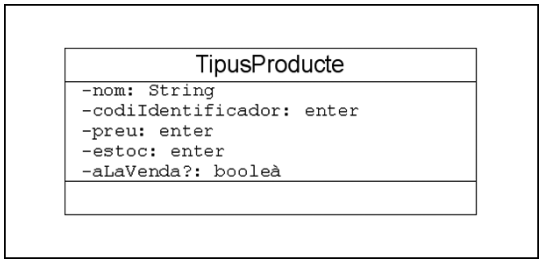
En el camp "nomTipus" s'indica el tipus d'element que conté la llista, per exemple:List<enter>, List<Cita>, List<String> etc. Tot i que no tots els llenguatges orientats a objectes suporten directament classes parametritzades, sempre hi ha alguna manera de simular aquest comportament. Per tant, assumir que existeixen en l'etapa de disseny no és problemàtic.

No oblideu que el "-" abans del nom indica que l'atribut té visibilitat privada.

Atès que en ambdós casos es tracta de classes, a l'hora de la implementació, abans de poder-hi operar cal instanciar-les i només és possible interactuar amb les instàncies mitjançant la crida d'operacions.

Així, doncs, els atributs d'una classe anomenada `TipusProducte` es poden definir tal com mostra la figura 1.10.

FIGURA 1.10. Especificació d'atributs de la classe TipusProducte



Partint de l'opció de poder especificar els atributs com a tipus bàsics o com a objectes, cal decantar-se entre dues aproximacions diferents: la pura i l'híbrida.

En una aproximació **pura**, tot element dins un programa és sempre un objecte, incloent-hi els atributs de cada objecte. Aquesta via és simplement una aplicació estricta de les bases de l'orientació a objectes, que indiquen que tot és un objecte i que un objecte es pot compondre d'altres objectes més simples. En aquesta aproximació, es pot considerar que ja hi ha també predefinides un conjunt de classes equivalents als tipus de dades enumerats en la taula 1.2. Per tant, es considera que les classes `Enter`, `Real`, `Caràcter`, etc. existeixen. No cal preocupar-se de com s'emmagatzemen realment els valors que representen; simplement, són capaços de fer la seva feina.

En canvi, en una aproximació **híbrida**, es considera acceptable especificar atributs usant tant objectes com tipus simples indistintament. D'aquesta manera, es té el millor dels dos mons. Normalment, s'usen tipus simples per als atributs més

senzills, els que són directament valors, i objectes per als que representen elements més complexos (com és el cas d'una cadena de text, per exemple).

Atès que en una aproximació pura es considera que ja hi ha classes predefinides per a qualsevol tipus simple, l'elecció de quina aproximació usar és purament estilística: depèn de quant estricte vol ser el dissenyador respecte a l'aplicació de les bases de l'orientació a objectes. De totes maneres, cal tenir present que sí hi ha algunes implicacions en l'elecció. La més important de totes és que només es pot interactuar amb un objecte per mitjà de la crida d'operacions. Per tant, en una aproximació pura qualsevol atribut només és manipulable d'aquesta manera.

Donat que Java usa una aproximació híbrida, nosaltres treballarem sempre d'aquesta manera.

Enllaços entre objectes

A partir dels mapes d'objectes s'han detectat enllaços entre objectes. De fet, aquests són molt importants, ja que un objecte només pot cridar l'operació d'un altre objecte si existeix aquest enllaç. Per tant, també cal poder indicar aquests enllaços al especificar la classe.

> Cada **enllaç** indica, implícitament, un atribut a la classe de l'objecte origen a la classe de l'objecte destinació.

Depenent de si un objecte ha de gestionar un o molts enllaços, es pot usar un únic atribut o una llista (`List`). Per exemple, una agenda gestiona moltes pàgines, pel que es pot especificar l'atribut:

```
1 –pagines: List<Pagina>
```

D'altra banda, si ens interessa controlar la pàgina actual, que només és una, es pot fer:

```
1 –paginaActual: Pagina
```

Atributs de classe

A part dels atributs que defineixen les propietats de cada instància d'una classe, hi ha un tipus especial d'atributs, anomenats **atributs** de classe. A l'hora de definir-los, es diferencien subratllant-los, si bé la sintaxi és idèntica als atributs genèrics:

```
1 __visibilitat nomAtributClasse: tipus [= valor inicial]__
```

Per exemple:

```
1 __+pi: real__
```

Els atributs de classe són especialment útils per definir constants.

La particularitat d'aquesta mena d'atributs és que descriuen una propietat de la classe, no dels seus objectes, i el seu valor és únic dins el programa. No hi ha

una variable separada dins de cada instància, com passa amb la resta d'atributs. A efectes pràctics, es pot considerar que un atribut de classe actua com una variable global, compartida per totes les instàncies.

1.3.2 Especificació d'operacions

Les operacions especifiquen el comportament dels objectes d'una classe. Igual que en el cas dels atributs, és un bon moment per concretar com es representa realment aquest comportament dins un programa real, de manera que sigui més entenedora l'explicació de com s'especifiquen quan es dissenya una classe.

Les operacions definides en cada classe s'implementen mitjançant la definició de **mètodes** en el codi font de les classes. Cada mètode conté el conjunt d'instruccions del llenguatge de programació necessàries per efectuar la tasca associada. Quan en un programa en execució un objecte crida una operació, s'executa el codi del mètode associat. Així, doncs, en aquest aspecte, un mètode no és diferent d'una funció o acció dins de qualsevol programa no orientat a objectes. L'orientació a objectes serveix per establir de quina manera es distribueix el codi dins el programa: dins de les classes que defineixen els diferents tipus d'objectes del programa.

> Dins un programa en execució, cada operació es materialitza en un **mètode**, el conjunt de codi que fa la tasca corresponent.

Val la pena mencionar que, tot i que moltes vegades s'usen els termes *operació* i *mètode* indistintament, formalment descriuen coses diferents. Mentre que el terme *operació* s'utilitza exclusivament per parlar de disseny, el terme *mètode* està vinculat únicament a la implementació, al codi font de l'aplicació. Aquesta diferenciació ve donada pel fet que dins de diverses classes es pot especificar exactament la mateixa operació, però la implementació pot ser diferent per a cada classe.

La convenció de nomenclatura d'una operació és idèntica a la dels atributs.

Dins la definició d'una classe, les operacions disponibles s'especifiquen de la manera següent:

```
__visibilitat nomOperació (llistaParàmetres): tipusRetorn__
```

El camp "llistaParàmetres" té el format següent:

```
nomParàmetre1: tipus, ... , nomParàmetreN: tipus
```

Les operacions normalment es defineixen amb visibilitat pública.

Totes les explicacions donades per als tipus o la visibilitat en el cas dels atributs també són aplicables al cas de les operacions.

En el cas de la visibilitat, tot i que ja s'ha dit que l'UML no concreta cap significat específic, indicar a una classe A que una operació és pública normalment significa que qualsevol altre objecte `b:` que tingui un enllaç amb alguna instància d'`a:` la

pot cridar. Marcar-la com a privada significa que no la pot cridar cap altre objecte, independentment de la presència d'enllaços. En aquest darrer cas, l'operació només es pot cridar des del mateix objecte, de manera que es pot considerar una operació auxiliar.

Un cop decidit si es vol usar una aproximació pura o híbrida, cal mantenir aquesta elecció en la definició dels tipus dels paràmetres i del tipus de retorn de l'operació: només objectes o tipus primitius i objectes indistintament. Per indicar que una operació fa un conjunt de tasques sense retornar cap valor concret, no cal posar res en el camp de valor de retorn.

Alguns exemples d'especificacions d'operacions poden ser:

```
+afegirMèdia (m: Mèdia)
+ajustarVolum (v: enter)
+pausa/reanuda ()
+mèdiaSegüent(): Mèdia
```

Recordeu que el "+" abans del nom indica que les operacions tenen visibilitat pública.

Addicionalment, hi ha un conjunt d'operacions que no sempre cal especificar, ja que se suposen en dissenyar una classe: les operacions accessores.

S'acostumen a considerar **operacions o mètodes** (si ja es parla d'implementació) **accessors** els que donen accés de lectura o escriptura als atributs d'una classe.

Les implementacions d'aquestes operacions també se solen anomenar familiarment mètodes *setter* i *getter*.

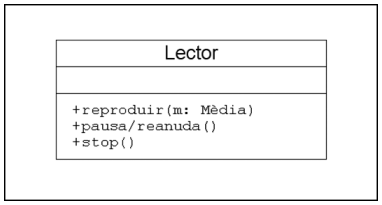
La nomenclatura estàndard per a l'accessor d'escriptura (per modificar el valor de l'atribut) i de lectura (per consultar-lo) és respectivament:

- setNomAtribut (valor: tipus).
- getNomAtribut(): tipus.

Per tant, en l'especificació d'una classe no cal explicitar tots els accessors entre les seves operacions. Per a cada atribut especificat ja es dóna per entès que sempre hi ha les operacions set i get associades, a menys que es digui el contrari.

La figura 1.11 mostra una representació gràfica completa per a la una classe anomenada `Lector`.

FIGURA 1.11. Especificació d'operacions de la classe Lector



Accessibilitat de dades dins d'operacions

Una operació s'acaba convertint en codi quan ha arribat el moment d'implementar-la. Per tant, a fi d'establir l'especificació d'una operació, és molt útil saber quines

Tenir una variable en què hi ha un objecte és equivalent a tenir un enllaç a aquest objecte.

dades és capaç de manipular. Donada una operació cridada sobre un objecte a:ClasseA, es pot considerar que el mètode que s'executarà té accés directe a les dades següents:

- Els valors emmagatzemats en els atributs de l'objecte a:ClasseA.
- Els valors dels paràmetres de l'operació.

En cas que qualsevol d'aquests valors sigui un objecte b:ClasseB, l'operació també pot:

- Accedir als valors dels atributs amb visibilitat pública de b:ClasseB.
- Cridar operacions amb visibilitat pública sobre l'objecte b:ClasseB.

Un cop es crida una nova operació sobre b:ClasseB, és aplicable exactament el mateix respecte a aquesta operació i l'objecte. Això és el que permet establir crides d'operacions que permeten fer una tasca concreta segons les ordres inicials de l'usuari.

Operacions de classe

Igual que en el cas dels atributs, hi ha el concepte d'operació de classe, especificada subratllant la definició.

```
__visibilitat nomOperació (llistaParàmetres): tipusRetorn__
```

De manera similar als atributs de classe, les operacions de classe no estan vinculades a objectes i, per tant, no es poden cridar sobre ells, ja que no tenen àmbit sobre els atributs de cap objecte concret, són generals. En canvi, una operació de classe sí que pot manipular atributs de classe. Aquest tipus d'operació se sol usar per a tasques de propòsit general a les quals s'ha d'accedir directament des de qualsevol objecte dins el programa (de la classe que sigui) o per manipular fàcilment atributs de classe. Per exemple:

```
__+comptarObjectesInstanciats (): enter__
```

Aquest és un cas en què pot tenir sentit disposar d'una funció fàcilment accessible des de qualsevol classe.

Abusar d'operacions de classe converteix un programa orientat a objectes en un de clàssic. Si bé aquestes operacions poden ser útils, cal pensar molt bé si una operació realment només ha de ser de classe o no.

Ubicació d'operacions

Un dels moments que pot resultar dificultós dins de tot el procés de disseny és decidir quines operacions cal especificar en cada moment. El motiu principal és

que, per fer-ho, cal tenir una visió general de tot el diagrama estàtic i com han d'interactuar els objectes per resoldre les diferents tasques que es vol que faci el programa. Això contrasta amb l'especificació dels atributs, en què, en la majoria de casos, només cal tenir present la classe que els conté i res més.

Un dels principis que cal seguir en ubicar operacions és el de la **cohesió:** obtenir al final del disseny classes amb una certa coherència i que aportin una idea molt clara de quin és el seu paper dins el problema que s'està descomponent. Quan es dissenyen diferents classes, el que no es vol crear és un conjunt de "calaixos de sastre" on s'amunteguin atributs i operacions sense cap mena de lògica. Les diferents classes del disseny han de coexistir en harmonia i cadascuna ha de tenir un objectiu molt clar. Així, s'espera que una torradora torri llesques de pa o que un caixer automàtic permeti consultar un saldo o treure diners, però no s'espera que cap d'aquests dos dispositius pugui inflar globus. Des del punt de vista purament tècnic, res no impedeix que puguin arribar a fer aquesta tasca aplicant-hi certes modificacions, però no és l'objectiu per al qual s'han creat ni el seu comportament lògic.

> Per mantenir la **cohesió** en un disseny orientat a objectes, cada classe ha de representar un únic element, perfectament definit, dins la descomposició del problema.

Un aspecte vinculat a garantir la cohesió de les classes és el d'**assignació de responsabilitats**: a partir de cada classe s'instancien objectes que actuen com a peces dins la simulació i cadascuna d'aquestes peces té una tasca molt concreta. Només cal assignar els atributs i les operacions mínims imprescindibles per fer aquesta tasca exclusivament. Un altre aspecte important per assolir una correcta cohesió és assignar les operacions a la classe correcta.

> Cal ubicar les **operacions** que operen amb una informació determinada en la mateixa classe en què es troba aquesta informació.

Exemple: el mètode afegirCita

En una aplicació d'una agenda, composada per les classes `Agenda`, `Pàgina` i `cita`, es vol especificar l'operació següent:

```
+afegirCita(c: Cita)
```

Aquesta és l'encarregada d'escriure una nova cita en una pàgina de l'agenda. En quina classe s'hauria d'ubicar? D'acord amb el principi de cohesió, ha d'estar ubicada en la classe que gestiona o conté directament les cites. En aquest cas, seria la classe "Pàgina".

Donant per suposat que s'aplicarà el principi de cohesió, una estratègia per ubicar operacions és la següent:

1. Localitzar les classes les instàncies de les quals interactuaran (rebran ordres o intercanviaran informació) directament amb l'usuari. Normalment, si s'ha seguit l'estratègia d'especificar una classe que representa "el tot", de manera

que el problema es va descomponent a partir d'ella, aquesta sol ser la classe a escollir.

2. Elaborar una llista d'interaccions possibles amb l'usuari: què és el que realment vol fer l'aplicació. Novament, en cap moment s'ha de pensar en una interfície d'usuari concreta. Cal pensar en el *què* però no en el *com*.
3. Cada element de la llista és una operació que cal especificar a les classes identificades al pas 1.
4. Per a cada operació, pensar de quina manera cal que els objectes del programa interactuïn per dur-la a terme, i anar especificant noves operacions a la resta de classes.

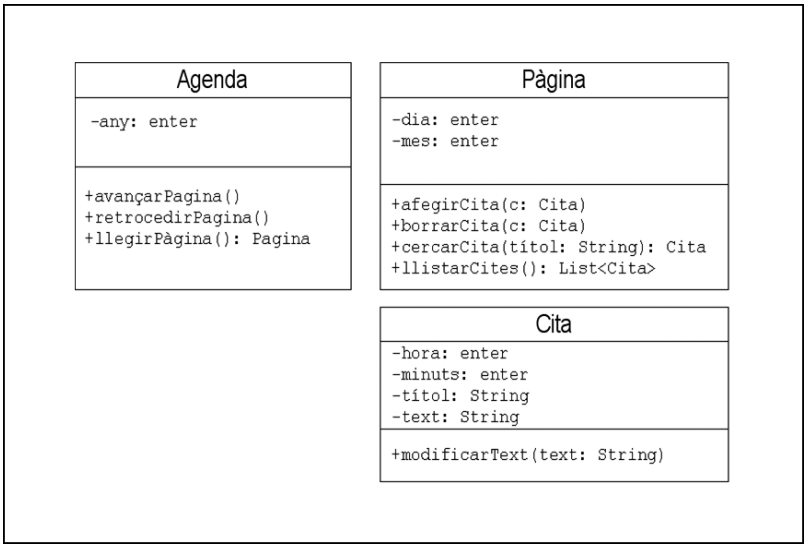
Com es pot veure, en el procés d'especificació d'operacions, el punt de partida no és cada classe individual sinó cada operació. Primer es pensen les operacions i després es mira on s'ubiquen.

1.3.3 Exemples d'especificacions d'atributs i operacions

Si bé identificar els atributs sol ser una tasca relativament senzilla, per identificar les operacions cal pensar què es vol obtenir amb l'aplicació i quina mena d'operacions han d'anar cridant els diferents objectes per arribar a fer cada tasca. En l'estudi d'aquests exemples val especialment la pena reflexionar sobre els principis d'ubicació d'operacions.

1) L'agenda. La figura 1.12 presenta l'especificació total de les classes d'una aplicació que serveix d'agenda, amb tots els atributs i operacions.

FIGURA 1.12. Especificació completa a les classes de l'agenda



En aquest exemple senzill és interessant estudiar com els objectes interactuen per dur a terme una tasca. D'acord amb aquesta especificació, el protocol per escriure una cita, per exemple, és el següent:

- A partir de l'objecte `agenda:Agenda`, es passaria de pàgina fins arribar a la que correspon a la data escollida, usant les operacions `avançarPàgina` i `retrocedirPàgina`. Aquesta acció la faria l'usuari: és ell qui cridarà aquesta operació. Com ho faci ja depèn de la interfície i del llenguatge emprat. En l'etapa de disseny, aquest fet no importa, n'hi ha prou de saber que ja hi haurà algun mecanisme.

- Cada cop que es passa de pàgina, es pot veure quina és la pàgina actual amb l'operació `llegirPàgina`. Un cop s'obté l'objecte de la pàgina actual, se'n pot inspeccionar el contingut mitjançant les operacions accessores de la classe Pàgina (no especificades explícitament, però existents).

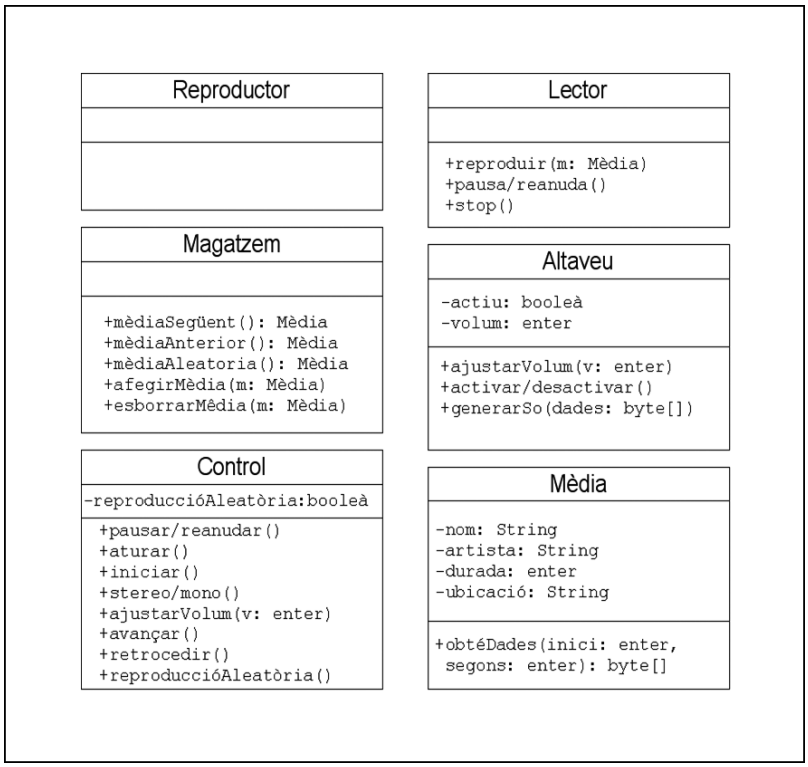
- Per a cada pàgina, es poden visualitzar totes les cites existents amb l'operació `llistarCites`.

- Si aquesta és la pàgina en què es vol afegir una cita, cal instanciar un objecte `novaCita:Cita`, inicialitzant tots els seus atributs al valor que correspongui. Novament, com s'instancia un objecte ja és un detall d'implementació, que dependrà del llenguatge escollit; en l'etapa de disseny no cal entrar en aquests detalls.

- Finalment, cal escriure la cita cridant sobre l'objecte de la pàgina actual l'operació `afegirCita(novaCita)`. Evidentment, aquesta operació ha de controlar que no hi hagi encavalcaments d'hora entre les cites escrites en la pàgina.

2) Un reproductor multimèdia. La figura 1.13 presenta l'especificació completa de les classes de l'aplicació del reproductor multimèdia.

Figura 1.13. Especificació completa a les classes del reproductor multimèdia



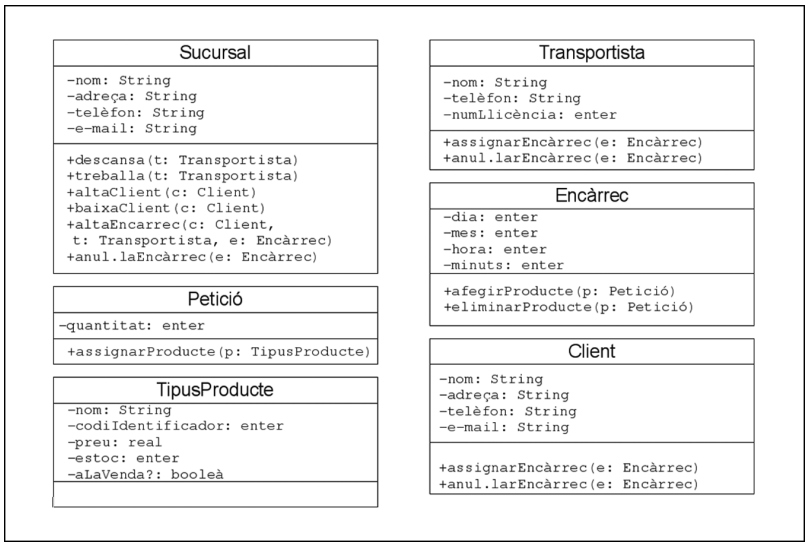
En aquesta especificació es pot apreciar que hi ha operacions repetides. Per entendre els motius d'aquesta circumstància, el més senzill és fer un símil amb un aparell reproductor del món real. D'una banda, l'usuari de l'aparell no interactua mai directament amb tots els components interns, només ho fa amb el tauler de control. Això, en aquest cas, és en contrast amb l'agenda, en què un usuari sí que pot interactuar directament amb les pàgines individuals. Per tant, aquest objecte que representa el tauler de control és el punt d'entrada de les ordres de l'usuari.

D'altra banda, quan l'usuari dóna una ordre (en el món real, per exemple, prem el botó d'aturada), l'ordre es propaga del tauler de control als components interns (per exemple, un senyal elèctric o una molla fa aturar el lector). Per tant, tot i que el tauler de control i el lector poden rebre una ordre d'aturar, l'acció que faran serà diferent. Mentre que el primer passa l'ordre i informa l'usuari del resultat, el segon atura el processament de la música en marxa i deixa d'enviar senyals als altaveus.

En aquesta especificació, tot aquest símil es tradueix en forma d'objectes i crides d'operacions. Per tant, en aquesta especificació la instància de la classe Control fa de gestor de totes les peticions de l'usuari cap a la resta de components del reproductor.

3) L'aplicació de gestió. L'especificació completa de les classes duna aplicació de gestió d'encàrrecs es plasma en la figura 1.14.

FIGURA 1.14. Especificació completa a les classes de l'aplicació de gestió



1.4 Manipulació d'objectes

L'execució d'un programa orientat a objectes es basa totalment en la gestió i manipulació dels objectes que el componen en cada moment. Per tant, el primer pas per veure com funciona tot plegat és saber què es pot fer amb un objecte. Per poder manipular-lo, primer cal crear-lo d'alguna manera dins el programa. Un cop creat, es poden invocar operacions sobre ell. Però per tal de poder cridar

operacions, és imprescindible un enllaç a l'objecte. En el cas d'objectes que manipulin altres objectes, el que manipula ha de disposar d'un enllaç al manipulat. En aquest apartat veureu com es fa tot això en el codi font Java. De moment, però, ens conformarem a treballar amb objectes de classes que ja proporciona el Java, en lloc de fer-ho directament amb classes definides per vosaltres.

Fent un petit resum, per poder **treballar amb objectes**, ens cal saber el següent: com crear-los, com accedir-hi (o referenciar-los), com inicialitzar-los, com manipular-los i com eliminar-los.

1.4.1 Com es creen els objectes?

La creació d'un objecte la realitza sempre una operació especial de la classe, anomenada *constructor*, que es distingeix perquè té el mateix nom que la classe (incloent majúscules/minúscules). Els constructors poden incorporar paràmetres i això permet que hi pugui haver diferents constructors, que es distingeixen pel nombre i/o els tipus dels seus paràmetres.

Per crear un objecte d'una classe cal, doncs, consultar prèviament la documentació de la classe per conèixer quins són els constructor proporcionats. Així, per exemple, si volem crear un objecte de la classe `Date` proporcionada per Java, en consultarem la documentació (figura 1.15).

Figura 1.15. Documentació proporcionada pel Java referent als constructors de la classe Date

Constructor Summary

`Date(int year, int month, int day)`
Deprecated. *instead use the constructor* `Date(long date)`

`Date(long date)`
Constructs a `Date` object using the given milliseconds time value.

En la figura veiem que la classe `Date` incorpora dos constructors, un dels quals ens diu que és obsolet (*deprecated*). És molt possible que no tinguem prou informació amb el prototipus, i en aquesta situació procedirem a desplegar la informació específica de cada constructor (figura 1.16).

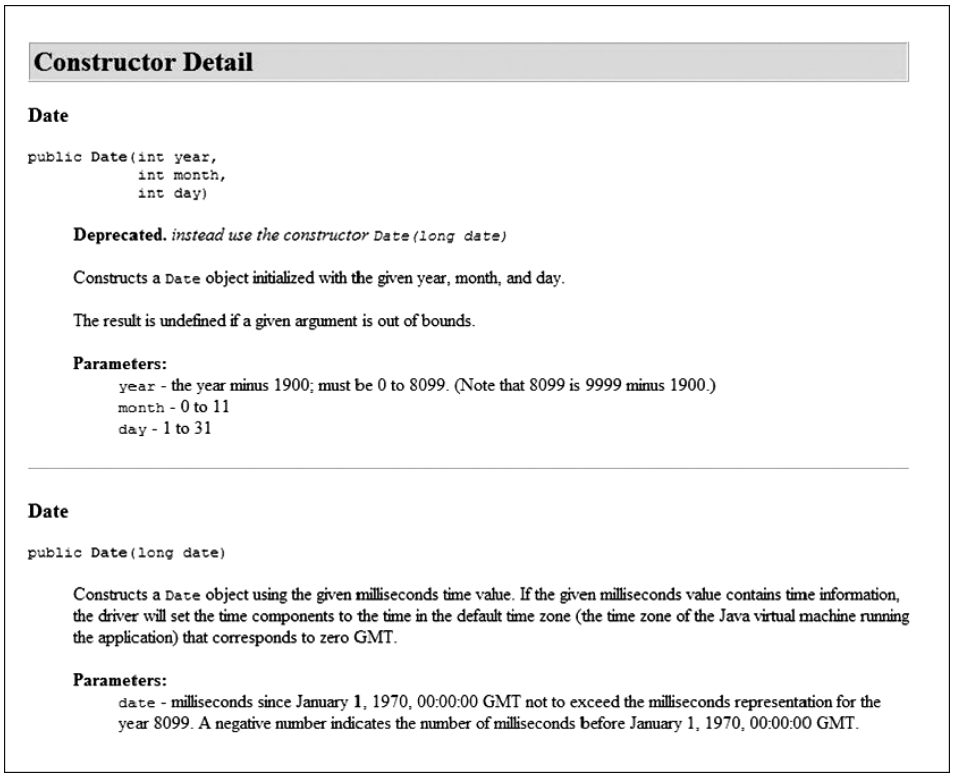
Instanciació

Atès que els objectes són instàncies de la classe, en lloc de dir que l'operador crea un objecte, també es diu que l'operador instancia la classe, ja que executa la creació d'una instància de la classe.

Sobre el constructor obsolet, cal dir que el llenguatge Java té una forta evolució i, de vegades, decideix crear noves classes amb noves funcionalitats, per substituir les existents, ja que es necessitaven més prestacions que les proporcionades fins el moment. Per qüestions de compatibilitat del programari ja existent amb les noves versions de Java, quan això succeïx no s'elimina la versió antiga, però s'avisa del fet que és obsoleta (*deprecated*), i potser arribarà una versió de Java en què se'n decideixi la desaparició.

Una vegada ja hem decidit quin constructor utilitzarem, ja podem crear l'objecte. Tota classe té, com a mínim, un constructor i no totes les classes tenen més d'un constructor. En el llenguatge Java, tot objecte es crea obligatòriament amb l'operador `new` acompanyat de la crida al constructor que correspongui.

FIGURA 1.16. Documentació detallada dels constructors de la classe Date



Constructor Detail

Date

```
public Date(int year,
            int month,
            int day)
```

Deprecated. *instead use the constructor* `Date(long date)`

Constructs a `Date` object initialized with the given year, month, and day.

The result is undefined if a given argument is out of bounds.

Parameters:

`year` - the year minus 1900; must be 0 to 8099. (Note that 8099 is 9999 minus 1900.)
`month` - 0 to 11
`day` - 1 to 31

Date

```
public Date(long date)
```

Constructs a `Date` object using the given milliseconds time value. If the given milliseconds value contains time information, the driver will set the time components to the time in the default time zone (the time zone of the Java virtual machine running the application) that corresponds to zero GMT.

Parameters:

`date` - milliseconds since January 1, 1970, 00:00:00 GMT not to exceed the milliseconds representation for the year 8099. A negative number indicates the number of milliseconds before January 1, 1970, 00:00:00 GMT.

Així, en Java, podem crear objectes `Date` fent:

```
1 new Date (109,0,1);// Objecte amb 1–1–2009 a les 00:00:00
2 new Date (0);// Objecte amb 1–1–1970 a les 00:00:00
```

L'operador `new` crea un **objecte** assignant la memòria necessària de manera automàtica.

1.4.2 Com es fa referència als objectes?

Si sabem com crear un objecte amb l'operador `new` acompanyat d'un constructor de la classe, necessitem saber com s'accedeix a l'objecte una vegada creat. Necessitem algun mecanisme per referir-nos-hi i això s'aconsegueix declarant una variable per fer referència a objectes de la classe concreta i assignant a aquesta variable el resultat de l'execució de l'operador `new`, el qual retorna una referència (adreça de memòria) a l'objecte creat. Aquest concepte és semblant a el que es fa amb un tipus primitiu, per exemple un enter. Per una banda tenim un literal, amb el valor amb el que es vol treballar, i volem emmagatzemar-lo en algun lloc per poder-nos-hi referir dins el codi, i manipular-lo.

Per abús de llenguatge, enlloc de dir "declarar una variable per fer referència a objectes de la classe X" es diu "declarar un objecte de la classe X". Val a dir que aquesta segona manera de parlar necessita menys paraules i els programadors en POO saben què s'hi amaga al darrera... Però s'ha de vigilar perquè programadors

que s'inicien en POO poden pensar que "declarar un objecte" porta implícita la "creació de l'objecte", i això seria un gran error.

La sintaxi per declarar una variable de nom "obj" per fer referència a objectes de la classe `X` és:

```
X obj;
```

La sintaxi és idèntica a la declaració d'altres variables: primer el tipus (en aquest cas, el nom de la classe) i tot seguit un identificador.

Alerta, però, ja que en aquesta variable no hi ha cap objecte! Per aconseguir que "obj" faci referència a un objecte, hem d'assignar-hi el resultat de l'execució de l'operador `new` o assignar-hi el contingut d'una altra variable que estigui fent referència a un objecte de la classe. L'assignació de valor a una variable de referència es pot efectuar en el mateix moment en què s'efectua la declaració o amb posterioritat, tal com es veu en els exemples següents:

```
// Declaració de variable de referència no inicialitzada
X obj1;

// Creació d'objecte al que es podrà accedir via la variable de
// referència obj1
obj1 = new X(...);

// Declaració de variable de referència i creació d'objecte al
// que es podrà accedir via la variable de referència obj2
X obj2 = new X(...);

// Declaració de variable de referència no inicialitzada
X obj3;

// La variable obj3 fa referència al mateix objecte que fa
// referència la variable obj1
obj3 = obj1;

// Declaració de variable de referència que fa referència al
// mateix objecte que fa referència la variable obj2
X obj4 = obj2;
```

Ara bé, cal tenir present que en crear un objecte amb l'operador `new` no sempre és necessari explicitar una variable per recollir la referència a l'objecte creat. De moment, deixarem això com un cas especial.

Si tornem a la classe `Date`, per crear un nou objecte d'aquest tipus i desar-lo a una variable, es faria, per exemple:

```
Date d = new Date();
```

1.4.3 Com s'inicialitzen els objectes?

En l'orientació a objectes, la inicialització dels objectes és una tasca que s'efectua durant el procés de construcció dels objectes. És a dir, si el dissenyador de la classe ha considerat oportú que els objectes, en la seva creació, inicialitzin les

seves dades (algunes o totes) amb uns valors determinats, haurà hagut de plasmar aquestes inicialitzacions en el(s) constructor(s).

Cal dir que, en certs llenguatges, la construcció dels objectes és responsabilitat exclusiva del constructor cridat i és molt lícit dir "la inicialització dels objectes és una tasca que efectua el constructor". En Java, però, la construcció d'un objecte pot tenir quatre fases de les quals el constructor cridat només actua en la darrera i la inicialització es pot dur a terme en les quatre fases, motiu pel qual és més lícit dir que "la inicialització dels objectes és una tasca que s'efectua durant el procés de construcció dels objectes".

Com que els constructors admeten el pas de paràmetres, el dissenyador de la classe pot proporcionar, als programadors usuaris de la classe, constructors que incorporin paràmetres, de manera que els valors indicats en la crida del constructor puguin ser utilitzats per inicialitzar les dades de l'objecte creat.

Vegem diverses construccions d'objectes de la classe Date que permeten diferents maneres d'inicialitzar els objectes creats:

```
1 Date d1 = new Date (109,0,1); //Objecte inicialitzat amb data 1-1-2009 a les
      00:00:00
2 Date d2 = new Date (0);   //Objecte inicialitzat amb data 1-1-1970 a les
      00:00:00
3 Date d3 = new Date ();   //Objecte inicialitzat amb la data i l'hora del
      sistema
```

1.4.4 Com es manipulen els objectes?

La manipulació dels objectes d'una classe s'ha de fer per mitjà de les operacions que proporciona la pròpia classe, amb una sintaxi molt simple:

```
1 <variableQueFaReferènciaObjecte>.<nomMètode>(<paràmetres>);
```

Així, per canviar el dia, el mes o l'any d'objectes Date, el llenguatge Java ens proporciona `setDate()`, `setMonth()` i `setYear()` i podrem cridar-los sobre qualsevol objecte `Date`:

```
1 Date d = new Date (109,0,1); // Nou objecte amb valor 1-1-2009
2 d.setYear (100);      // Canviem valor a 1-1-2000
3 d.setMonth (5);      // Canviem valor a 1-6-2000
4 d.setDate (40);      // Canviem valor a 10-7-2000
```

La manera lògica de manipular els objectes d'una classe és utilitzar les operacions que la classe proporciona i, en la majoria de casos, aquesta serà l'única possibilitat, ja que els dissenyadors de les classes acostumen a obligar a la seva utilització i no permeten l'accés directe a les dades contingudes en els objectes.

1.4.5 Com es destrueixen els objectes?

Els objectes, en el moment de la seva creació, ocupen un espai de memòria i, per tant, cal ser conscients que cal destruir els objectes quan ja no es necessitin. En la majoria de llenguatges de programació orientats a objectes (C++ entre ells) és responsabilitat del programador tenir sempre present les dades dinàmiques generades per tal d'eliminar-les de la memòria quan ja no siguin necessàries. Escriure el codi per fer aquest tipus de gestió de la memòria és avorrit i provoca molts errors (oblits, adreces perdudes de dades dinàmiques...).

En Java tots els objectes són dinàmics. Per tant, caldria portar un control exhaustiu de tots els objectes creats i anar-los destruint explícitament quan ja no fossin necessaris. Afortunadament, Java ens estalvia aquesta feina, de manera que ens permet crear tants objectes com es vulgui (únicament limitats per la pròpia capacitat de memòria del sistema), els quals mai han de ser destruïts, ja que és l'entorn d'execució de Java el que elimina els objectes quan determina que no s'utilitzaran més.

El **garbage collector** (recuperador de memòria) és un procés automàtic de la màquina virtual Java que periòdicament s'encarrega de recollir els objectes que ja no es necessiten i els destrueix tot alliberant la memòria que ocupaven.

El mecanisme que segueix el recuperador de memòria per detectar els objectes que ja no s'utilitzaran més és molt senzill: escaneja tots els objectes i totes les variables de referències a objectes que hi ha en la memòria de manera que els objectes pels quals no hi ha cap variable de referència que hi apunti són objectes que ja no s'utilitzaran més i, per tant, són recol·lectats per ser destruïts.

Les referències a objectes es perden en els casos següents:

- Quan la variable que conté la referència deixa d'existir perquè el flux d'execució del programa abandona definitivament l'àmbit en què havia estat creada.
- Quan la variable que conté la referència passa a contenir la referència en un altre objecte o passa a valer null.

L'execució del recuperador de memòria és automàtica, però un programa pot demanar al recuperador de memòria que s'executi immediatament mitjançant una crida al mètode `System.gc()`. Ara bé, l'execució d'aquesta crida no garanteix que la recol·lecció s'efectuï; dependrà de l'estat d'execució de la màquina virtual.

Abans que es reculli un objecte, el recuperador de memòria li dóna la possibilitat d'executar unes darreres voluntats, les quals han d'estar recollides en una operació de nom `finalize()` dins la classe a què pertany l'objecte. Aquesta possibilitat pot ser necessària en diverses situacions:

- Quan calgui alliberar recursos del sistema gestionats per l'objecte que és a punt de desaparèixer (arxius oberts, connexions amb bases de dades...).
- Quan calgui alliberar referències a altres objectes per fer-los candidats a ser tractats pel recuperador de memòria.

2. Declaració de classes

L'element fonamental de tot programa orientat a objectes és l'objecte. Aquests objectes es generen a partir d'un fitxer de codi font, una classe, on es defineixen les seves propietats i el seu comportament (atributs i mètodes). El llenguatge Java proporciona un conjunt de classes ja creades que es poden usar directament, però gairebé sempre és necessari generar classes noves, d'acord a les necessitats de cada programa concret. Per tant, la clau per poder generar el codi font d'un programa orientat a objectes està en el fet de saber com generar codi per declarar classes correctament.

2.1 Pas a codi de classes

La codificació d'una classe segueix la sintaxi següent, en què s'aprecien dues parts ben diferenciades.

```
1 DeclaracióDeLaClasse {
2   CosDeLaClasse
3 }
```

Cadascuna de les dues parts (declaració i cos) pot ser més o menys complexa i, com acostuma a succeir en l'aprenentatge de qualsevol llenguatge, començarem per les formes més simples per avançar posteriorment cap a formes més complexes.

En principi, totes les classes que hem dissenyat han tingut, com a declaració, la sintaxi següent:

```
1 public class <NomClasse>
```

Aquesta declaració es pot veure ampliada amb altres modificadors (a més del public) a l'esquerra de la paraula `class` i amb uns modificadors a la dreta de `NomClasse`. Per crear les primeres classes, però, no els necessitem.

El modificador davant el nom d'una classe possibilita que la classe sigui accessible des d'altres classes.

El cos de la classe és una seqüència de tres tipus de components:

- Els relatius a les **dades** que contindran els objectes de la classe (els atributs).
- Els relatius a blocs de codi sense nom, coneguts com a **iniciadors**.
- Els relatius als **mètodes** que la classe proporciona per gestionar les dades que emmagatzema.

En principi aquests tres tipus de components es poden incloure dins la definició de la classe en qualsevol ordre, però hi ha el conveni de començar amb les dades, continuar amb els iniciadors i finalitzar amb els mètodes.

Així, doncs:

```
public class <NomClasse> {
  <seqüènciaDeclaracionsDeDades>;
  <seqüènciaIniciadors>;
  <seqüènciaDefinicionsDeMètodes>
}
```

Un fitxer de codi Java pot incorporar diverses classes, però normalment el millor és que només es declari una a cada fitxer. El nom del fitxer ha de ser exactament igual al de la classe (incloses majúscules/minúscules)

2.1.1 Declaració de les dades

La seqüència de declaracions de dades consisteix en declaracions de variables de tipus primitius i/o de referències a objectes d'altres classes, seguint la sintaxi següent:

```
[<modificadors>] <nomTipus> <nomDada> [=<valorInicial>];
```

En aquesta sintaxi veiem que la declaració de la dada pot estar precedida d'uns modificadors. Normalment, sempre s'usarà el modificador `private`, excepte en casos especials, com la declaració de constants. En aquest cas, s'usa `public static final`.

Veiem també que la declaració d'una dada pot estar acompanyada d'una inicialització explícita (`=<valorInicial>`).

Java inicialitza implícitament les dades dels objectes durant el procés de creació, però en canvi no inicialitza les variables declarades en mètodes.

En el moment en què crea cada dada, Java efectua una inicialització implícita de totes les dades amb valor zero pels tipus enters, reals i caràcter, amb valor false per al tipus lògic i amb valor `null` per a les variables de referència. Posteriorment s'executen les inicialitzacions explícites que hagi pogut indicar el programador en la declaració de la dada.

2.1.2 Iniciadors

Els iniciadors són blocs de codi (sentències entre claus) que s'executen cada vegada que es crea un objecte de la classe. Es defineixen seguint la sintaxi següent:

```
{
  <conjunt_de_sentències>;
}
```

Quin sentit té l'existència d'iniciadors si ja disposem dels constructors per indicar el codi a executar en la creació d'objectes? La resposta és que de vegades podem tenir blocs de codi a executar en el procés de creació d'un objecte de la classe, sigui quin sigui el constructor (n'hi poden haver diversos) emprat en la creació, i

la utilització d'iniciadors ens permet no haver de repetir el mateix codi dins els diversos constructors.

A més, els iniciadors també són indicats per ser utilitzats en el disseny de classes anònimes, les quals, en no tenir nom, no poden tenir mètodes constructors.

En cas d'existir diversos iniciadors s'executen en l'ordre en què es trobin dins la classe.

2.1.3 Definició de les operacions

La seqüència de definicions d'operacions consisteix en la definició (prototipus i contingut) dels diversos mètodes amb la sintaxi de Java. La manera més simple de definir un mètode en Java segueix la sintaxi següent:

```
[<modificadors>] <tipusRetorn> <nomMètode> (<llistaArguments>) {
  <declaracióVariablesLocals>
  <cosDelMètode>
}
```

En aquesta sintaxi veiem que la declaració del mètode pot anar precedida d'uns modificadors, tot i que el més habitual (però no sempre) serà `public`. Per crear els primers mètodes, però, no els necessitem. Per indicar que un mètode no retorna cap resultat, s'utilitza el tipus `void`.

Respecte a la llista d'arguments, cal comentar que el pas de paràmetres en Java sempre és usant el mecanisme anomenat *per valor*, o sigui, es garanteix que tot paràmetre utilitzat en una crida a un mètode manté el valor inicial en finalitzar l'execució del mètode, però, si el paràmetre és una variable que fa referència a un objecte, l'objecte sí pot ser modificat (no substituït) dins el mètode. Al acabar la crida, aquesta modificació es manté.

Ja estem en condicions de dissenyar la primera classe i fer un petit programa que comprovi el funcionament dels diferents mètodes desenvolupats.

Primera versió d'una classe per gestionar persones

Suposem que es vol dissenyar una classe per gestionar persones, per a les quals interessa gestionar-ne el dni, el nom i l'edat.

Prenem les primeres decisions de disseny i decidim que dni i nom han de ser objectes `String` i que edat ha de ser un `short`. Respecte als mètodes, en un principi se'ns acut desenvolupar els mètodes corresponents a les operacions accessores i, potser, un mètode visualitzar() per mostrar tot el contingut d'una persona.

Mètodes accessors

Totes les classes acostumen a proporcionar uns mètodes de lectura (*get*) i escriptura (*set*) sobre els atributs de la classe, de manera que poden ser manipulats: són les anomenades *operacions accessores*.

Una possible solució és:

```java
//Fitxer Persona.java
public class Persona {
   String dni;
   String nom;
   short edat;
   // Retorna: 0 si s'ha pogut canviar el dni
   //          1 si el nou dni no és correcte – No s'efectua el canvi
   int setDni(String nouDni) {
      // Aquí hi podria haver una rutina de verificació del dni
      // i actuar en conseqüència. Com que no la incorporem, retornem sempre 0
      dni = nouDni;
      return 0;
   }

   void setNom(String nouNom) {
      nom = nouNom;
   }

   // Retorna: 0 si s'ha pogut canviar l'edat
   //          1 : Error per passar una edat negativa
   //          2 : Error per passar una edat "enorme"
   int setEdat(int novaEdat) {
      if (novaEdat<0) return 1;
      if (novaEdat>Short.MAX_VALUE) return 2;
      edat = (short)novaEdat;
      return 0;
   }

   String getDni() { return dni; }
   String getNom() { return nom; }
   short getEdat() { return edat; }

   void visualitzar() {
      System.out.println("Dni...........:" + dni);
      System.out.println("Nom...........:" + nom);
      System.out.println("Edat..........:" + edat);
   }

   public static void main(String args[]) {
      Persona p1 = new Persona();
      Persona p2 = new Persona();
      p1.setDni("00000000");
      p1.setNom("Pepe Gotera");
      p1.setEdat(33);
      System.out.println("Visualització de persona p1:");
      p1.visualitzar();
      System.out.println("El dni de p1 és " + p1.getDni());
      System.out.println("El nom de p1 és " + p1.getNom());
      System.out.println("L'edat de p1 és " + p1.getEdat());
      System.out.println("Visualització de persona p2:");
      p2.visualitzar();
   }
}
```

L'execució d'aquest programa dóna el resultat següent:

```
Visualització de persona p1:
Dni...........:00000000
Nom...........:Pepe Gotera
Edat..........:33
El dni de p1 és 00000000
El nom de p1 és Pepe Gotera
L'edat de p1 és 33
Visualització de persona p2:
Dni...........:null
Nom...........:null
Edat..........:0
```

Classes embolcall

El llenguatge Java proporciona per a cadascun dels vuit tipus primitius una classe corresponent, amb el mateix nom que el tipus però iniciades amb majúscula, anomenades *classes embolcall* (*wrapper*, en anglès), que proporcionen dades i mètodes per a la gestió dels tipus de dades corresponents.

L'execució del programa sembla adequada. Veiem que les dades de la persona "p2", no inicialitzades explícitament, han estat inicialitzades - tal i com hem dit més amunt - implícitament amb valor zero les numèriques i valor `null` les referències. Aprofitem aquest exemple per presentar una problemàtica que ens podem trobar en el disseny de moltes classes, relativa al fet que la classe conté dades de tipus `byte` o `short` i, en canvi, els arguments dels mètodes que recullen valors per emplenar aquestes dades es defineixen de tipus `int`. Per què ho fem? Què hem de tenir en compte?:

- La declaració dels arguments dels mètodes de tipus `int` està fonamentada en el tipus de dada associat als literals que s'acostumaran a utilitzar. Així, com que és molt possible cridar el mètode `setEdat()` passant un literal enter (com en l'exemple), és lògic declarar l'argument d'aquest mètode de tipus `int`, ja que els literals enters són d'aquest tipus (o `long` si s'afegeix la lletra "L" al final del literal). Si haguéssim declarat l'argument de tipus `short` (adequat al tipus de la dada a què s'assignarà en l'interior del mètode), la crida al mètode s'hauria de fer passant un valor de tipus `short` o explicitant conversions com `setEdat( (short) 33)` i això no és desitjable.
- El fet de declarar els arguments dels mètodes amb els tipus de dada més usuals tenint en compte els literals amb els quals podem cridar el mètode ens porta al fet que a l'interior del mètode haguem de fer comprovacions relatives a si el valor que arriba és adequat en termes de grandària (rang). En l'exemple, el mètode `setEdat()` rep per paràmetre un valor `int` i en el seu interior, abans d'emplenar la dada edat declarada de tipus `short`, ens interessa comprovar si el valor és assumible per a una dada de tipus `short`. Per fer aquests tipus de comprovacions, el llenguatge Java ens proporciona mecanismes per saber quins són els rangs de valors permesos pels diferents tipus de dades. En l'exemple, utilitzem el valor `MAX_VALUE` de la classe `Short` per comprovar si el valor enter de l'argument "novaEdat" del mètode `setEdat()` és massa gran per la dada edat.

2.1.4 Modificadors dins d'una classe

A l'hora de definir atributs o mètodes dins una classe, és possible indicar un **modificador d'accés**. Vegem amb més detall quins són a la sintaxi del Java.

```
1 [<modificadorAccés>] [<altresModificadors>] <tipusDada> <nomDada>;
2
3 [<modificadorAccés>] [<altresModificadors>] <tipusRetorn> <nomMètode> (<
      llistaArgs>)
4 {...}
```

El modificador d'accés pot prendre quatre valors:

- **Public**, que dóna accés a tothom.
- **Private**, que prohibeix l'accés a tothom menys pels mètodes de la pròpia classe.

Paquets

Les classes es poden organitzar en paquets i aquesta possibilitat s'acostuma a utilitzar quan tenim un conjunt de classes relacionades entre elles. Totes les classes no incloses explícitament en cap paquet i que estan situades en un mateix directori es consideren d'un mateix paquet.

- **Protected**, que es comporta com a `public` per a les classes derivades de la classe i com a `private` per a la resta de classes.
- **Sense modificador**, que es comporta com a `public` per a les classes del mateix paquet i com a `private` per a la resta de classes.

Donada la classe `Persona`, si desenvolupem un programa que instanciï objectes de la classe, tenim accés directe a les dades dni, nom i edat? Considerem el programa següent en què es creen objectes de la classe `Persona`.

```
1  //Fitxer CridaPersona.java
2  public class CridaPersona {
3     public static void main(String args[]) {
4        Persona p = new Persona();
5        p.dni = "--$%#@--";
6        p.nom = "";
7        p.edat = -23;
8        System.out.println("Visualització de la persona p:");
9        p.visualitzar();
10    }
11 }
```

En aquest cas estem en un programa extern a la classe Persona i es veu com accedim directament a les dades dni, nom i edat de la persona creada, i podem fer autèntiques animalades. El compilador no es queixa (cal haver compilat també l'arxiu Persona.java en el mateix directori) i l'execució dóna el resultat:

```
1  Visualització de la persona p:
2  Dni...........:--$%#@--
3  Nom...........:
4  Edat..........:-23
```

Acabem de veure, doncs, que la versió actual de la classe `Persona` permet el lliure accés als valors dels seus atributs, ja que en la definició d'aquestes dades no s'ha posat al davant el modificador adequat per evitar-ho. Les classes `CridaPersona` i `Persona`, en estar situades en el mateix directori, s'han considerat del mateix paquet i, per tant, en no haver-hi cap modificador d'accés en la definició de les dades dni, nom i edat, la classe `CridaPersona` hi ha tingut accés total. A més, en no haver-hi cap modificador d'accés en la definició dels mètodes, aquests no poden ser cridats per classes de paquets diferents del paquet al qual pertany la classe `Persona`.

Normalment, al crear classes el més correcte és que els atributs no tinguin accés directe. Els motius són:

- Protegir les dades de modificacions impròpies.
- Facilitar el manteniment de la classe, ja que si per algun motiu es creu que cal efectuar alguna reestructuració de dades o de funcionament intern, es podran efectuar els canvis pertinents sense afectar les aplicacions desenvolupades (sempre que no es modifiquin els prototipus dels mètodes existents).

Sembla lògic, doncs, fer evolucionar la versió actual de la classe `Persona` cap a una classe que tingui les dades declarades com a privades i els mètodes com

a públics. Fixem-nos que el mètode `main` per comprovar el funcionament d'una classe sempre ha estat declarat public.

Versió de la classe Persona amb modificadors d'accés adequats

A continuació presentem una versió evolucionada de la classe Persona que inclou els modificadors d'accés adequats: dades a `private` i mètodes a `public`.

```
//Fitxer Persona.java

public class Persona {
   private String dni;
   private String nom;
   private short edat;

   // Retorna: 0 si s'ha pogut canviar el dni
   //          1 si el nou dni no és correcte – No s'efectua el canvi
   public int setDni(String nouDni) {
      // Aquí hi podria haver una rutina de verificació del dni
      // i actuar en conseqüència. Com que no la incorporem, retornem sempre 0
      dni = nouDni;
      return 0;
   }

   public void setNom(String nouNom) {
      nom = nouNom;
   }

   // Retorna: 0 si s'ha pogut canviar l'edat
   //          1 : Error per passar una edat negativa
   //          2 : Error per passar una edat "enorme"
   public int setEdat(int novaEdat) {
      if (novaEdat<0) return 1;
      if (novaEdat>Short.MAX_VALUE) return 2;
      edat = (short)novaEdat;
      return 0;
   }

   public String getDni() { return dni; }

   public String getNom() { return nom; }

   public short getEdat() { return edat; }

   public void visualitzar() {
      System.out.println("Dni...........:" + dni);
      System.out.println("Nom...........:" + nom);
      System.out.println("Edat..........:" + edat);
   }

  public static void main(String args[]) {
     Persona p1 = new Persona();
     Persona p2 = new Persona();
     p1.setDni("00000000");
     p1.setNom("Pepe Gotera");
     p1.setEdat(33);
     System.out.println("Visualització de persona p1:");
     p1.visualitzar();
     System.out.println("El dni de p1 és " + p1.getDni());
     System.out.println("El nom de p1 és " + p1.getNom());
     System.out.println("L'edat de p1 és " + p1.getEdat());
     System.out.println("Visualització de persona p2:");
     p2.visualitzar();
  }
}
```

Amb aquesta versió de la classe `Persona` compilada, vegem què succeeix quan intentem compilar la classe `CridaPersona` que crea una persona i intenta accedir directament a les dades:

```
CridaPersona.java:11: dni has private access in Persona
   p.dni = "--$%#@--";
   ^
CridaPersona.java:12: nom has private access in Persona
   p.nom = "";
   ^
CridaPersona.java:13: edat has private access in Persona
   p.edat = -23;
   ^
3 errors
```

Mètodes privats

Pot tenir sentit un mètode `private`? La resposta és afirmativa, ja que en el disseny d'una classe pot interessar desenvolupar un mètode intern per ser cridat en el disseny d'altres mètodes de la classe i no es vol donar a conèixer a la comunitat de programadors que utilitzaran la classe.

Fixem-nos que el compilador ja detecta que no hi ha accés a les dades. Hem aconseguit el nostre objectiu: protegir l'accés directe a les dades. Ara potser no sigui massa evident encara, però els avantatges d'assolir aquest objectiu s'aniran fent més evidents a mesura que es vagi avançant en l'aprenentatge de la creació de programes orientats a objectes.

2.1.5 Sobrecàrrega de mètodes

De vegades, en els programes, cal dissenyar diverses versions de mètodes que tenen un mateix significat i/o objectiu però que s'apliquen en diferents tipus i/o nombre de dades. Així, si necessitàvem disposar d'una funció que sabés sumar dos enters i d'una funció que sabés sumar dos reals, podriem fer simplement dos mètodes diferents anomenats `sumaEnters` i `sumaReals`. Els dos tenen el mateix objectiu i significat, tot i que la gestió interna pot ser força diferent, i des d'un punt de vista lògic, com que les dues permeten calcular una suma,

Java permet declarar mètodes repetits amb el mateix nom. Això no és possible a tots els llenguatges de programació. Per exemple:

```
public int suma (int n1, int n2) { ... }
public double suma (double r1, double r2) { ... }
```

El terme anglès per a la sobrecàrrega, molt emprat en informàtica, és overloading.

La **sobrecàrrega** de mètodes és la funcionalitat que permet tenir mètodes diferents amb un mateix nom.

Normalment la sobrecàrrega d'un nom de mètode s'utilitza en aquells que tenen un mateix objectiu, però és lícit utilitzar-la en mètodes que no tinguin res a veure. Això no acostuma a succeir si el dissenyador assigna a els mètodes noms que tinguin a veure amb el seu objectiu.

Hi ha dues regles per poder aplicar la sobrecàrrega de mètodes:

- La llista d'arguments ha de ser suficientment diferent per permetre una determinació inequívoca del mètode que es crida.

- Els tipus de dades que retornen poden ser diferents o iguals i no n'hi ha prou de tenir els tipus de retorn diferents per distingir el mètode que es crida.

El compilador només pot distingir el mètode que es crida a partir del nombre i tipus dels paràmetres indicats en la crida.

Exemples de mètodes sobrecarregats els podem trobar en moltes classes proporcionades pel llenguatge Java. Així, per exemple, la coneguda classe `String` té molts mètodes sobrecarregats, com ara `format()`, `getBytes()`, `indexOf()`, etc.

2.2 Inicialització d'objectes

La construcció d'un objecte s'efectua amb la utilització de l'operador `new` acompanyada d'un constructor de la classe. Si bé per classes ja existents dins les llibreries de Java aquests constructors ja existeixen, pel cas de classes noves generades dins un programa orientat a objectes, caldrà declarar-hi aquests constructors entre els seus mètodes disponibles.

2.2.1 Procés d'inicialització d'un objecte al Java

Els passos que segueix la màquina virtual davant l'execució de l'operador `new` són:

1. Reserva memòria per desar el nou objecte i totes les seves dades són inicialitzades amb valor zero pels tipus enters, reals i caràcter, amb valor `false` pel tipus booleà, i amb valor `null` per les variables on es desen objectes.
2. S'executen les inicialitzacions explícites. Les dades membres d'una classe es poden inicialitzar explícitament tot assignant expressions en la declaració dels membres.
3. S'executen els iniciadors (blocs de codi sense nom) que hi ha dins la classe seguint l'ordre d'aparició dins d'aquesta.
4. S'executa el constructor indicat en la construcció de l'objecte amb l'operador `new`.

Exemple d'inicialització explícita de dades membres en una classe

```
1  //Fitxer InicialitzacioExplicita.java
2  import java.util.Date;
3
4  public class InicialitzacioExplicita {
5     private int x = 20;
```

```
6     private int y;
7     private Date d = new Date (100,0,1);
8     private String s;
9
10    public static void main(String args[]) {
11       InicialitzacioExplicita obj = new InicialitzacioExplicita();
12       System.out.println("x = " + obj.x);
13       System.out.println("y = " + obj.y);
14       System.out.println("d = " + obj.d);
15       System.out.println("s = " + obj.s);
16    }
17 }
```

En aquesta classe veiem que conté quatre dades membre ("x", "y", "d" i "s") de les quals n'hi ha dues que són inicialitzades explícitament en el moment de la declaració corresponent. Posteriorment, en crear un objecte "obj" de la classe, podem comprovar que les diferents dades d'aquest objecte tenen el valor esperat ("x" i "d" són inicialitzades amb els valors indicats en la declaració i "y" i "s" són inicialitzades amb els valors zero i `null` que assigna Java).

L'execució d'aquest programa és:

```
1 x = 20
2 y = 0
3 d = Sat Jan 01 00:00:00 CET 2000
4 s = null
```

2.2.2 Declaració de constructors

El mecanisme d'inicialització explícita és una manera senzilla d'inicialitzar els camps d'un objecte. No obstant això, de vegades es necessita executar un mètode constructor en concret per implementar la inicialització, ja que pot ser necessari fer el següent:

- Recollir valors (pas de paràmetres en el moment de construcció) de manera que es puguin tenir en compte en la construcció de l'objecte.
- Gestionar errors que puguin aparèixer en la fase d'inicialització.
- Aplicar processos, més o menys complicats, en els quals poden intervenir tot tipus de sentències (condicionals i repetitives).

Tot això és possible gràcies a l'existència dels mètodes constructors, un dels quals sempre es crida en crear un objecte amb l'operador `new`. Per tant, a les classes creades per vosaltres pot ser necessari declarar constructors. En el disseny d'una classe es poden dissenyar mètodes constructors, però si no se'n dissenya cap, el llenguatge proveeix automàticament d'un constructor sense paràmetres.

Els mètodes constructors d'una classe han de seguir les normes següents:

- El nom del mètode és idèntic al nom de la classe.
- No se'ls pot definir cap tipus de retorn (ni tant solsvoid, no es posa absolutament res. Es deixa en blanc).
- Poden estar sobrecarregats, és a dir, podem definir diversos constructors amb el mateix nom i diferents arguments. En cridar l'operador `new`, la llista de paràmetres determina quin constructor s'utilitza.
- Si es defineix algun constructor (amb paràmetres o no), el llenguatge Java deixa de proporcionar el constructor sense paràmetres automàtic i, per tant, per poder crear objectes cridant un constructor sense paràmetres, caldrà definir-lo explícitament.

Exemple de constructors adequats per a la classe Persona

A continuació presentem un parell de constructors adequats per a la classe Persona:

```
1  public Persona () {}
2
3  public Persona (String sDni, String sNom, int nEdat) {
4    dni = sDni;
5    nom = sNom;
6    if (nEdat>=0 && nEdat<=Short.MAX_VALUE)
7     edat = (short)nEdat;
8  }
```

Gràcies als dos constructors podem crear objectes com mostra el mètode següent `main()`:

```
1  public static void main(String args[]) {
2    Persona p1 = new Persona("00000000","Pepe Gotera",33);
3    Persona p2 = new Persona();
4    System.out.println("Visualització de persona p1:");
5    p1.visualitzar();
6    System.out.println("Visualització de persona p2:");
7    p2.visualitzar();
8  }
```

El constructor que permet passar per paràmetres el dni, el nom i l'edat de l'objecte `Persona` a construir s'ha utilitzat per crear l'objecte a què fa referència la variable "p1".

El constructor sense paràmetres permet la creació de l'objecte `Persona` a què fa referència la variable "p2". Si no haguéssim definit el constructor sense paràmetres, la creació d'aquest objecte no hauria estat possible.

L'execució del mètode `main()` presentat facilita la sortida:

```
1  Visualització de persona p1:
2  Dni...........:00000000
3  Nom...........:Pepe Gotera
4  Edat..........:33
5  Visualització de persona p2:
```

```
Dni..........:null
Nom..........:null
Edat.........:0
```

2.2.3 La paraula reservada "this"

En Java existeix una paraula reservada especialment útil per tractar la manipulació d'atributs i la seva inicialització. Es tracta de `this`, que té dues finalitats principals:

- Dins els mètodes no constructors, per fer referència a l'objecte actual sobre el qual s'està executant el mètode. Així, quan dins un mètode d'una classe es vol accedir a una dada de l'objecte actual, podem utilitzar la paraula reservada `this`, escrivint `this.nomDada`, i si es vol cridar un altre mètode sobre l'objecte actual, podem escriure `this.nomMètode(...)`. En aquests casos, la utilització de la paraula `this` és redundant, ja que dins un mètode, per referir-nos a una dada de l'objecte actual, podem escriure directament `nomDada`, i per cridar un altre mètode sobre l'objecte actual podem escriure directament `nomMètode(...)`. De vegades, però, la paraula reservada `this` no és redundant, com en el cas en què es vol cridar un mètode en una classe i cal passar l'objecte actual com a argument: `nomMètode(this)`.
- Dins els mètodes constructors, com a nom de mètode per cridar un altre constructor de la pròpia classe. De vegades pot passar que un mètode constructor hagi d'executar el mateix codi que un altre mètode constructor ja dissenyat. En aquesta situació seria interessant poder cridar el constructor existent, amb els paràmetres adequats, sense haver de copiar el codi del constructor ja dissenyat, i això ens ho facilita la paraula reservada `this` utilitzada com a nom de mètode: `this(<llistaParàmetres>)`. La paraula reservada `this` com a mètode per cridar un constructor en el disseny d'un altre constructor només es pot utilitzar en la primera sentència del nou constructor. En finalitzar la crida d'un altre constructor mitjançant `this`, es continua amb l'execució de les instruccions que hi hagi després de la crida `this(...)`.

Exemple d'utilització de la paraula reservada "this" en mètodes de la classe Persona

En primer lloc veiem que ens pot interessar tenir un constructor per crear una persona a partir d'una persona ja existent, és a dir, el constructor `Persona (Persona p)`.

Però, d'altra banda, ja tenim un constructor (anomenem-lo `xxx`) que ens sap construir una persona a partir d'un dni, un nom i una edat passats per paràmetre. Per tant, per construir una persona a partir d'una persona p donada, ens interessa cridar el constructor xxx passant-li com a paràmetres el dni, el nom i l'edat de

la persona p. Això ens ho facilita la paraula reservada `this` com a crida d'un constructor existent:

```
public Persona (Persona p) {
  this (p.dni, p.nom, p.edat);
}
```

En segon lloc, suposem que volem tenir un mètode, anomenat clonar, que aplicat sobre un objecte `Persona` en creï un clon, és a dir, una altra persona idèntica, i retorni la referència a la nova persona. Per aconseguir-ho hem de dissenyar el mètode que en seu interior cridi un dels constructors de la classe. Si optem per utilitzar el constructor `Persona (Persona p)` necessitem la paraula reservada `this` per fer referència a l'objecte actual:

```
public Persona clonar () {
  return new Persona (this);
}
```

El mètode `main()` següent permet comprovar el funcionament de tots dos mètodes:

```
public static void main(String args[]) {
  Persona p1 = new Persona("00000000","Pepe Gotera",33);
  Persona p2 = new Persona(p1);
  Persona p3 = p1.clonar();
  System.out.println("Visualització de persona p2:");
  p2.visualitzar();
  System.out.println("Visualització de persona p3:");
  p3.visualitzar();
}
```

Veiem que els dos mètodes proporcionen el mateix resultat (creació d'una nova persona com a còpia d'una persona existent) i, per tant, el mètode clonar és irrellevant si ja tenim el constructor, però ens ha servit per veure una aplicació de la paraula reservada `this` per fer referència a l'objecte actual sobre el qual s'executa un mètode.

L'execució del mètode `main()` presentat facilita la sortida:

```
Visualització de persona p2:
Dni...........:00000000
Nom...........:Pepe Gotera
Edat..........:33
Visualització de persona p3:
Dni...........:00000000
Nom...........:Pepe Gotera
Edat..........:33
```

2.3 Elements estàtics d'una classe

Les dades membre estàtic, com que són comunes per a tots els objectes de la classe, també s'anomenen variables classe.

Alguns elements d'una classe es poden declarar com "estàtics". Per fer-ho, el llenguatge Java proporciona la paraula reservada `static`, amb tres finalitats:

1) Com a modificador en la declaració de dades membres d'una classe, per aconseguir que la dada afectada sigui comuna a tots els objectes de la classe. Per aconseguir aquest efecte, la dada corresponent es declara amb el modificador `static`, seguint la sintaxi següent:

```
static [<altresModificadors>] <tipusDada> <nomDada> [=<valorInicial>];
```

Les dades `static` es creen en efectuar la càrrega de la classe, quan encara no hi ha cap instància (objecte) de la classe. Atès que una dada `static` és comuna per a tots els objectes de la classe, s'hi accedeix de manera diferent de la utilitzada per les dades no `static`:

- Per accedir-hi des de fora de la classe (possible segons el modificador d'accés que l'acompanyi), no es necessita cap objecte de la classe i s'utilitza la sintaxi `NomClasse.nomDada`. Recordeu que perquè això funcioni, igualment, la dada s'ha de declarar com pública.
- Per accedir-hi des de la pròpia classe, no cal indicar cap nom d'objecte (`nomObjecte.nomDada`), sinó directament el seu nom.

En qualsevol cas, el llenguatge Java permet accedir a una dada `static` mitjançant el nom d'un objecte de la classe, però no és una nomenclatura coherent.

2) Com a modificador en la declaració de mètodes d'una classe, per aconseguir que el mètode afectat es pugui executar sense necessitat de ser cridat sobre cap objecte concret de la classe.

Si feu una ullada a la documentació del llenguatge Java, en la majoria de les classes us adonareu de l'existència de mètodes que tenen una sintaxi similar a la següent:

```
... static <valorRetorn> <nomMètode> (<llistaArguments>)
```

Com a exemple, dins la classe `String`, podeu veure el mètode:

```
public static String valueOf(char[]data)
```

L'explicació que l'acompanya ens diu que aquest mètode, a partir d'una taula de caràcters, ens proporciona un nou objecte `String` que conté la seqüència de valors de la taula de caràcters. Per tant, és clar que l'execució d'aquest mètode no necessita cap objecte `String` i, per tant, és lògic que sigui declarat `static`. Davant aquest raonament, pot aparèixer la pregunta de per què, si no necessita de cap objecte String, és declarat com un mètode de la classe `String`? La resposta rau en el fet que en el llenguatge Java tot mètode s'ha d'implementar en alguna classe i, ja que aquest mètode permet aconseguir un objecte `String`, sembla lògic que resideixi dins la classe `String`.

Un altre cas potser més habitual i evident és el mètode `main` que s'usa en les classes principals. Per poder invocar un mètode cal fer-ho sobre un objecte. Però com és possible cridar `main`, si en iniciar l'execució del programa encara no existeix cap objecte? Els objectes es creen precisament dins el `main`! Aquest problema

seria un peix que es mossega la cua. La resposta està a fer-lo `static`, de manera que és possible fer-ne la crida sense la necessitat que hi hagi cap objecte existent prèviament.

Dels mètodes `static` cal saber:

- Es criden utilitzant la sintaxi `NomClasse.nomMètode()`. El llenguatge Java permet cridar-los pel nom d'un objecte de la classe, però no és lògic.
- En el seu codi no es pot utilitzar la paraula reservada `this`, ja que l'execució no s'efectua sobre cap objecte en concret de la classe.
- En el seu codi només es pot accedir als seus propis arguments i a les dades `static` de la classe.
- No es poden sobreescriure (sobrecarregar-los en classes derivades) per fer-los no `static` en les classes derivades.

3) Com a modificador d'iniciadors (blocs de codi sense nom), per aconseguir un iniciador que s'executi únicament quan es carrega la classe. La càrrega d'una classe es produeix en la primera crida d'un mètode de la classe, que pot ser el constructor involucrat en la creació d'un objecte o un mètode estàtic de la classe. La declaració d'una variable per fer referència a objectes de la classe no provoca la càrrega de la classe.

La sintaxi a emprar és:

```
static {...}
```

2.3.1 Exemple d'utilització de la paraula reservada "static" en les diverses possibilitats

La classe següent ens mostra una situació en què la declaració d'una dada `static` és necessària, ja que es vol portar un comptador del nombre d'objectes creats de manera que a cada nou objecte es pugui assignar un número de sèrie a partir del nombre d'objectes creats fins al moment.

Així mateix sembla oportú proporcionar un mètode, anomenat `nombreObjectesCreats()` per donar informació, com el seu nom indica, referent al nombre d'objectes creats de la classe en un moment donat.

Per acabar, s'ha inclòs un parell d'iniciadors per comprovar el funcionament dels iniciadors `static` i no `static`.

```
//Fitxer ExempleUsosStatic.java

public class ExempleUsosStatic {
   private static int comptador = 0;
   private int numeroSerie;

```

```
7     static { System.out.println ("Iniciador \"static\" que s'executa en carregar
            la classe"); }
8
9     { System.out.println ("Iniciador que s'executa en la creació de cada objecte
          "); }
10
11    public ExempleUsosStatic () {
12       comptador++;
13       numeroSerie = comptador;
14       System.out.println ("S'acaba de crear l'objecte número " + numeroSerie);
15    }
16
17    public static int nombreObjectesCreats () {
18       return comptador;
19    }
20
21    public static void main(String args[]) {
22       ExempleUsosStatic d1 = new ExempleUsosStatic();
23       ExempleUsosStatic d2;
24       d2 = new ExempleUsosStatic();
25       System.out.println("Número de sèrie de d1 = " + d1.numeroSerie);
26       System.out.println("Número de sèrie de d2 = " + d2.numeroSerie);
27       System.out.println("Objectes creats: " + nombreObjectesCreats());
28    }
29 }
```

L'execució del programa dóna el resultat:

```
1  Iniciador "static" que s'executa en carregar la classe
2  Iniciador que s'executa en la creació de cada objecte
3  S'acaba de crear l'objecte número 1
4  Iniciador que s'executa en la creació de cada objecte
5  S'acaba de crear l'objecte número 2
6  Número de sèrie de d1 = 1
7  Número de sèrie de d2 = 2
8  Objectes creats: 2
```

2.3.2 Exemple per comprovar quan es produeix la càrrega d'una classe

El programa següent demostra en quin moment es carrega una classe i, per tant, s'executen els iniciadors `static` que pugui tenir definits. Per executar aquest programa cal tenir en el mateix directori el fitxer compilat de la classe `ExempleUsosStatic`.

```
1  //Fitxer: CarregaClasse.java
2
3  public class CarregaClasse {
4     public static void main (String args[]) {
5        System.out.println ("Punt 1. Abans de declarar la variable obj");
6        ExempleUsosStatic obj;
7        System.out.println ("Punt 2. Després de declarar la variable obj");
8        System.out.println ("        i abans d'invocar el mètode static");
9        System.out.println ("Anem a invocar el mètode static: " +
10                           ExempleUsosStatic.nombreObjectesCreats());
11    }
12 }
```

L'execució d'aquest programa mostra com l'execució de l'iniciador `static` de la classe `ExempleUsosStatic` es produeix just abans de la sentència que inclou la

crida del mètode `static`, malgrat que abans s'hagi declarat una variable per fer referència a objectes de la classe `ExempleUsosStatic`.

```
Punt1.Abans de declarar la variable obj
Punt2.Després de declarar la variable obj
   i abans d'invocar el mètode static
Iniciador "static" que s'executa en carregar la classe
Anem a invocar el mètode static: 0
```

2.4 Llibreries de classes

Normalment, a l'hora de generar diferents classes, serà desitjable organitzar-les de manera que se'n pugui facilitar la gestió i saber quines estan relacionades entre si, per exemple, formant part d'un mateix programa. El llenguatge Java proporciona un mecanisme, anomenat *package*, per poder agrupar classes.

Abans d'entrar a veure en profunditat el funcionament dels *packages* del Java, és important tenir clar com es representa una classe Java dins el sistema de fitxers quan no intervenen els *packages* (o sigui, tal com hem treballant amb classes fins ara), tant a nivell de codi font com un cop compilada. D'aquesta manera, és més senzill entendre el seu impacte dins l'estructura d'un programa fet en Java. Això es deu al fet que, en usar un IDE, tot aquest procés de gestió dels fitxers de codi font i compilats és transparent al desenvolupador, però és igualment important saber quins fitxers estan jugant algun rol en la fase de desenvolupament d'una aplicació en Java.

Cada classe dins un programa es representa normalment dins un fitxer amb extensió `.java` i amb un nom idèntic (incloent majúscules i minúscules) al de la pròpia classe tal com s'ha definit al codi font (`public class NomClasse {...}`). Quan una classe es compila, es genera un fitxer amb extensió `.class` amb el mateix nom de la classe. Aquest fitxer es genera al mateix directori que el fitxer `.java` si s'usa el compilador amb intèrpret de comandes, però els IDE habitualment els ordenen en carpetes diferents dins els seus projectes. Per exemple, el Netbeans ubica els fitxers `.java` dins la carpeta `src`, mentre que els fitxers `.class` els ubica a la carpeta `build\classes`.

2.4.1 Packages

La pertinença d'una classe a un paquet s'indica amb la sentència package a l'inici del fitxer font en què resideix la classe i afecta a totes les classes definides en el fitxer. La sentència *package* ha de ser la primera sentència del fitxer font. Abans hi pot haver línies en blanc i/o comentaris, però res més.

Cal seguir la sintaxi següent:

```
package <nomPaquet>;
```

Els noms dels paquets (per conveni, amb minúscules) poden ser paraules separades per punts, fet que provoca que els corresponents `.class` s'emmagatzemin en una estructura jeràrquica de directoris que coincideix, en noms, amb les paraules que constitueixen el nom del paquet.

La inexistència de la sentència `package` implica que les classes del fitxer font es consideren en el paquet per defecte (sense nom) i els corresponents `.class` s'emmagatzemen en el mateix directori que el fitxer font.

Un paquet està constituït pel conjunt de classes dissenyades en fitxers font que incorporen la sentència `package` amb un nom de paquet idèntic. El paquet per defecte està constituït per totes les classes dissenyades en fitxers font que no incorporen la sentència *package*.

En el cas del Netbeans, les classe estaran a la carpeta "build/classes/xxx/yyy/zzz".

Totes les classes d'un paquet anomenat "xxx.yyy.zzz" resideixen dins la subcarpeta "zzz" de l'estructura de directoris "xxx/yyy/zzz", però podem tenir físicament aquesta estructura en diferents ubicacions. És a dir, donades les classes `C1` i `C2` del mateix paquet "xxx.yyy.zzz", es podria donar el cas que el fitxer `.class` corresponent a `C1` residís en path "xxx/yyy/zzz/C1" i que el fitxer `.class` corresponent a `C2` residís en path "xxx/yyy/zzz/C2".

Recordem que el codi incorporat en una classe (iniciadors i mètodes) té accés a tots els membres sense modificador d'accés de totes les classes del mateix paquet (a més de l'accés als membres amb modificador d'accés public).

En el disseny d'una classe es té accés a totes les classes del mateix paquet, però per accedir a classes de diferents paquets cal emprar un dels dos mecanismes següents:

- Utilitzar el nom de la classe precedit del nom del paquet cada vegada que s'hagi d'utilitzar el nom de la classe, amb la sintaxi següent:

```
1 nomPaquet.NomClasse
```

- Explicitar les classes d'altres paquets a les quals es farà referència amb una sentència import abans de la declaració de la nova classe, seguint la sintaxi següent:

```
1 import <nomPaquet>.<NomClasse>;
```

És factible carregar totes les classes d'un paquet amb una única sentència utilitzant un asterisc:

```
1 import <nomPaquet>.*;
```

Les sentències import en un fitxer font han de precedir a totes les declaracions de classes incorporades en el fitxer.

Així, doncs, si tenim una classe C en un paquet xxx.yyy.zzz i l'hem d'utilitzar en una altra classe, tenim dues opcions:

- Escriure `xxx.yyy.zzz.C` cada vegada que haguem de referir-nos a la classe C.
- Utilitzar la sentència `import xxx.yyy.zzz.C` abans de cap declaració de classe i utilitzar directament el nom `C` per referir-nos a la classe.

Exemple de definició de paquets de classes i accés corresponent

Considerem les classes dissenyades en el fitxer següent:

```
1  //Fitxer ClasseC1.java
2  package xxx.yyy.zzz;
3
4  public class ClasseC1 {
5    int mc1=10;
6  }
7
8  class ClasseC1Bis {
9    int mc1=20;
10 }
```

Veiem que aquest fitxer defineix les classes `ClasseC1` i `ClasseC1Bis` dins un paquet anomenat "xxx.yyy.zzz". Fixem-nos que una d'elles té el modificador public perquè s'hi pugui accedir des de fora del paquet, i recordem que en un fitxer .java només hi pot haver una classe `public`.

Considerem un nou fitxer .java que crea més classes en el mateix paquet "xxx.yyy.zzz": ClasseC2.java

```
1  //Fitxer ClasseC2.java
2  package xxx.yyy.zzz;
3
4  public class ClasseC2 {
5    int mc2=10;
6  }
7
8  class ClasseC2Bis {
9    int mc2=20;
10 }
```

Vegem, en primer lloc, que qualsevol classe d'un paquet té accés a totes les classes del mateix paquet i als membres de les que no hagin estat declarades `private`. Som-hi:

```
1  //Fitxer: AccesIntern.java
2  package xxx.yyy.zzz;
3
4  class AccesIntern {
5    public static void main (String args[]) {
6     ClasseC1 c1 = new ClasseC1();
7     ClasseC1Bis c1b = new ClasseC1Bis();
8     ClasseC2 c2 = new ClasseC2();
9     ClasseC2Bis c2b = new ClasseC2Bis();
10    System.out.println ("c1.mc1 = " + c1.mc1);
11    System.out.println ("c1b.mc1 = " + c1b.mc1);
12    System.out.println ("c2.mc2 = " + c2.mc2);
13    System.out.println ("c2b.mc2 = " + c2b.mc2);
14   }
15 }
```

Procedim a compilar els fitxers ClasseC1.java i ClasseC2.java. Veiem que la compilació no dóna cap error i podem comprovar l'estructura de directoris que hem generat dins de la carpeta del projecte del vostre IDE, amb aquestes compilacions:

```
\xxx
\xxx\yyy
\xxx\yyy\zzz
\xxx\yyy\zzz\ClasseC1.class
\xxx\yyy\zzz\ClasseC1Bis.class
\xxx\yyy\zzz\ClasseC2.class
\xxx\yyy\zzz\ClasseC2Bis.class
```

Si procedim a executar el programa de la classe `AccesIntern` obtenim:

```
c1.mc1 = 10
c1b.mc1 = 20
c2.mc2 = 10
c2b.mc2 = 20
```

Veiem que la classe `AccesIntern` té accés a totes les classes del mateix paquet i a les seves dades membres, ja que no s'havien definit com a `private`.

Comprovem ara què cal fer per accedir a les classes del paquet "xxx.yyy.zzz" des d'una classe d'un altre paquet. Comprovarem que no podem accedir a les classes no públiques del paquet "xxx.yyy.zzz" ni als membres no públics de les classes públiques. Per fer aquestes comprovacions, considerem la classe `AccesExtern` següent:

```
//Fitxer AccesExtern.java
import xxx.yyy.zzz.*;

class AccesExtern {
   public static void main (String args[]) {
      ClasseC1 c1 = new ClasseC1();
      //ClasseC1Bis c1b = new ClasseC1Bis();    // No és classe pública
      ClasseC2 c2 = new ClasseC2();
      //ClasseC2Bis c2b = new ClasseC2Bis();    // No és classe pública
      //System.out.println ("c1.mc1 = " + c1.mc1); // No són membres públics
      //System.out.println ("c2.mc2 = " + c2.mc2); // No són membres públics
   }
}
```

Veiem que les instruccions comentades donarien error pels motius següents:

- Les classes `ClasseC1Bis` i `ClasseC2Bis` no són públiques i, per tant, no s'hi té accés des de fora del paquet "xxx.yyy.zzz".
- El membre "mc1" de la classe `ClasseC1` i el membre "mc2" de la classe `ClasseC2` no són públics i, per tant, no s'hi té accés des de fora del paquet "xxx.yyy.zzz".

En el desenvolupament d'aplicacions en Java cal tenir especial cura a utilitzar noms que siguin únics i així poder-ne assegurar la reutilització en una gran organització i, encara més, en qualsevol lloc del món. Això pot ser una tasca difícil en una gran organització i absolutament impossible dins la comunitat d'Internet.

Per això es proposa que tota organització utilitzi el nom del seu domini, invertit, com a prefix per a totes les classes. És a dir, els paquets de classes desenvolupats per la Generalitat de Catalunya, que té el domini "gencat.cat", podrien començar per "cat.gencat".

2.4.2 Arxius jar

Una aplicació Java normalment es compon dels compilats de molts fitxers .java, la majoria dels quals formaran part de diferents paquets i, per tant, a l'hora de distribuir l'aplicació caldria mantenir l'estructura de directoris corresponent als paquets, cosa que pot convertir-se en una feina feixuga.

L'entorn JDK de Java ens proporciona l'eina "jar", executada sobre línia de comandes, per empaquetar totes les estructures de directoris i els fitxers .class en un únic arxiu d'extensió.jar, que no és més que un arxiu que conté a l'interior altres fitxers, similar als .zip del compressor WinZip o als .rar del compressor WinRAR.

Per crear un fitxer .jar cal seguir les indicacions que la mateixa eina ens dóna si l'executem sense passar-li cap informació referent al que cal empaquetar:

```
G:\>jar
Uso: jar {ctxui}[vfm0Me] [archivo–jar] [archivo–manifiesto] [punto–entrada] [–C
     dir] archivos...

Opciones:
    –c crear archivo de almacenamiento
    –t crear la tabla de contenido del archivo de almacenamiento
    –x extraer el archivo mencionado (o todos) del archivo de almacenamiento
    –u actualizar archivo de almacenamiento existente
    –v generar salida detallada de los datos de salida estándar
    –f especificar nombre del archivo de almacenamiento
    –m incluir información de un archivo de manifiesto especificado
    –e especificar punto de entrada de la aplicación para aplicación autónoma
      que se incluye dentro de un archivo jar ejecutable
    –0 sólo almacenar; no utilizar compresión ZIP
    –M no crear un archivo de manifiesto para las entradas
    –i generar información de índice para los archivos jar especificados
    –C cambiar al directorio especificado e incluir el archivo siguiente
Si algún archivo coincide también con un directorio, ambos se procesarán.
El nombre del archivo de manifiesto, el nombre del archivo de almacenamiento y
    el nombre del pun
to de entrada se especifican en el mismo orden que las marcas 'm', 'f' y 'e'.

Ejemplo 1: para archivar dos archivos de clases en un archivo de almacenamiento
     llamado classes.
jar:
        jar cvf classes.jar Foo.class Bar.class
Ejemplo 2:utilice un archivo de manifiesto ya creado, 'mymanifest', y archive
    todos los
        archivos del directorio foo/ en 'classes.jar':
        jar cvfm classes.jar mymanifest –C foo/ .
```

Així, doncs, per obtenir un arxiu .jar cal executar quelcom similar a:

```
jar cf nomArxiu.jar fitxer1.class fitxer2.class... directori1 directori2...
```

Un fitxer .jar es pot descomprimir i generar tota l'estructura de directoris en la ubicació en què es vulgui tot executant quelcom similar a:

```
jar xf nomArxiu.jar
```

També hi ha possibilitats d'extreure únicament el(s) fitxer(s) desitjat(s).

El gran avantatge dels fitxers .jar és que la màquina virtual permet l'execució dels fitxers que conté sense necessitat de desempaquetar, amb la sintaxi següent:

```
java –cp nomArxiu.jar fitxerQueContéMètodeMain
```

Però, tot i així, cal saber quin és el `fitxerQueContéMètodeMain`. Per evitar haver de recordar el nom de la classe amb el `main` es pot indicar en un fitxer especial, anomenat **fitxer de manifest**, i incloure aquest fitxer dins l'arxiu .jar. Per aconseguir-ho, generem un fitxer de text amb qualsevol nom (per exemple, manifest.txt) amb el contingut següent i, importantíssim, amb un salt de línia al final:

```
Main–Class: fitxerQueContéMètodeMain
```

Una vegada tenim el fitxer, l'hem d'incloure en l'arxiu .jar fent:

```
jar cmf manifest.txt nomArxiu.jar fitxer1.class fitxer2.class... directori1
    directori2...
```

L'opció "mf" indica que s'indica el nom del fitxer de manifest i el nom del fitxer empaquetat en aquest ordre; podem invertir les opcions:

```
jar cfm nomArxiu.jar manifest.txt fitxer1.class fitxer2.class... directori1
    directori2...
```

Un cop un conjunt de classes són empaquetades dins un fitxer .jar, aquest es pot afegir directament a l'entorn de treball, de manera que al fer-ho es considera que totes les seves classes formen part del programa que s'està generant. Gestionant un únic fitxer és possible gestionar-ne en realitat molts.

El fitxer de manifest pot contenir més informació. Deixem per a vosaltres la seva investigació.

Exemple de generació i utilització d'arxiu **.jar**

Netbeans genera automàticament fitxers JAR per als vostres projectes si useu l'opció *Build* de la barra d'eines. El fitxer es troba a la carpeta `dist`.

Considerem els fitxers `ClasseC1.java`, `ClasseC2.java` i `AccesIntern.java` que formen part del paquet "xxx.yyy.zzz" i el fitxer `AccesExtern.java`. Suposem que estem en una ubicació en què tenim el compilat AccesExtern.class i d'on penja l'estructura de directoris xxx/yyy/zzz amb els fitxers `ClasseC1.class`, `ClasseC2.class` i `AccesIntern.class`.

Per generar un fitxer JAR que contingui els quatre .class amb l'estructura de directoris indicada:

```
G:\>jar cf paquet.jar AccesExtern.class xxx/yyy/zzz
```

Aquesta ordre ens ha generat un arxiu JAR que conté totes les classes indicades i que podem utilitzar per distribuir la nostra aplicació. Per comprovar-ne la funcionalitat, podem moure el fitxer JAR generat a una altra ubicació, situar-nos-hi i executar.

Utilització avançada de classes

Joan Arnedo Moreno

Índex

Introducció

Tard o d'hora, al llarg de la vostra vida com a desenvolupadors d'aplicacions arriba el moment que cal modificar algun dels programes que heu fet. Els motius poden ser molt diversos i poden anar des del fet que s'ha trobat una errada que cal arreglar tan aviat com sigui possible, fins haver de crear una nova versió amb noves funcionalitats, partint de la feina que ja s'ha fet fins ara i sense haver de començar de zero. En realitat, molt poques vegades, un cop s'ha finalitzat el desenvolupament d'una aplicació, us en podreu oblidar totalment, així com de tota la feina feta.

Atès aquest fet, un punt molt important dins el procés de disseny d'una aplicació és l'ús d'estratègies per poder generar programari que sigui modular, reaprofitable i fàcil d'adaptar davant canvis imprevistos en el futur, tant en aspectes de disseny com d'implementació. En un món dinàmic, en què les aplicacions han de satisfer demandes canviants, és imprescindible que els dissenys generats portin a implementacions en les quals fer modificacions o afegir noves funcions impliqui una dedicació mínima de temps i d'esforç. A més a més, cal que aquestes tasques les puguin fer fàcilment desenvolupadors que no han estat necessàriament implicats en el desenvolupament original.

Un altre aspecte destacable i que cal tenir present en crear mòduls per a les nostres aplicacions és fer que aquests mòduls siguin, tot i que sembli redundant dir-ho, realment modulars. O sigui, que cada mòdul sigui vertaderament una peça fàcilment intercanviable que pot ser reaprofitada en altres aplicacions amb necessitats semblants, sense haver de fer cap modificació. Ja sigui en aquesta aplicació nostra, o en una altra duta a terme per una altra persona que no ha tingut res a veure en el procés de desenvolupament del mòdul, i que, per tant, no té la més petita idea de com és el codi.

Aquesta unitat se centra en quines tècniques ofereix l'orientació a objectes per assolir tots aquests objectius, de manera que, si el dia de demà cal afegir noves funcionalitats a una aplicació, la quantitat de codi que calgui modificar sigui el mínim possible i resulti molt més senzill l'ús dels vostres mòduls per part d'altres desenvolupadors. Concretament, a l'apartat "Creació d'aplicacions escalables" veureu els tres mecanismes més importants dins l'orientació a objectes: l'encapsulació/ocultació d'informació, l'herència i el polimorfisme.

Resultats d'aprenentatge

En finalitzar aquesta unitat l'alumne/a:

1. Desenvolupa programes aplicant característiques avançades dels llenguatges orientats a objectes i de l'entorn de programació

- Identifica els conceptes d'herència, superclasse i subclasse.
- Utilitza modificadors per bloquejar i forçar l'herència de classes i mètodes.
- Reconeix la incidència dels constructors en l'herència.
- Reconeix la incidència dels destructors i/o mètodes de finalització en l'herència.
- Crea classes heretades que sobreescriguin la implementació de mètodes de la superclasse.
- Sap de l'existència de l'herència múltiple i els problemes derivats.
- Dissenya i aplica jerarquies de classes.
- Prova i depura les jerarquies de classes.
- Realitza programes que implementin i utilitzin jerarquies de classes.
- Comenta i documenta el codi.
- Entén, defineix i implementa interfícies

1. Creació d'aplicacions escalables

Existeixen diferents mecanismes bàsics per garantir l'escalabilitat de les aplicacions. D'una banda, hi ha el principi d'encapsulació i ocultació d'informació, a partir del qual els mòduls de què es compon una aplicació es generen com una capsa negra, on es defineixen les seves funcionalitats però no com les duen a terme internament. Aquesta important estratègia és aplicable a qualsevol tipus de llenguatge, i no està vinculada exclusivament a l'orientació a objectes, si bé ens centrarem en la seva aplicació en aquesta metodologia i en com usar-la en la definició de classes.

D'altra banda, l'orientació a objectes proporciona dos mecanismes exclusius, i especialment potents, per desenvolupar aplicacions fàcilment extensibles: l'herència i el polimorfisme. La seva particularitat principal és que només poden ser implementats amb un llenguatge que suporti orientació a objectes; no es poden implementar mitjançant altres llenguatges. Per aquest motiu, és en aquests mecanismes on es fa especial èmfasi.

Entre els seus avantatges més importants trobem:

- Ofereixen una manera fàcil de minimitzar la duplicació de codi.
- Tenen capacitat d'afegir noves funcionalitats, de manera senzilla, a classes ja definides.
- Es poden definir operacions en què es poden passar com a paràmetre objectes de classes que encara no s'han definit en el moment d'especificar l'operació.
- Aporten noves vies a la idea de poder manipular i organitzar els objectes de manera més semblant al pensament humà.

Aquests mecanismes són d'enorme utilitat quan es desenvolupa programari de certa complexitat, que s'ha de poder ampliar en el futur de manera ràpida i senzilla. De fet, actualment, tant l'herència com el polimorfisme són considerades característiques indispensables en qualsevol llenguatge orientat a objectes.

1.1 Encapsulació

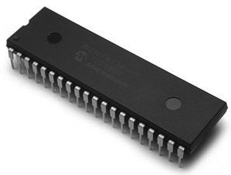
Una aplicació generada mitjançant l'orientació a objectes té molt en comú amb una altra que s'ha generat mitjançant un llenguatge clàssic: en darrera instància, tot està format per un conjunt de variables en les quals es desa l'estat del programa i un seguit de blocs de sentències que en componen la lògica. Així és com realment es representen els atributs i les operacions en el programa. L'aspecte diferencial

és l'estratègia d'organització del programa, que distribueix aquestes variables i blocs de sentències dins de classes, de manera que sigui molt més entenedora per a la manera de pensar humana i permeti maximitzar la reutilització de codi; dit d'una altra manera, es converteix el programa en una simulació. Ara bé, a fi de generar programari de qualitat, hi ha un principi que cal tenir molt present i aplicar acuradament durant l'etapa de disseny: el **principi d'encapsulació**. Si bé també cal tenir-lo present en el procés de desenvolupament de programari mitjançant llenguatges clàssics, és especialment beneficiós dins el camp de l'orientació a objectes.

En aplicar l'orientació a objectes, les classes s'han d'especificar de manera que actuïn com a caixes negres. Algú que no ha participat en el seu procés de disseny o de desenvolupament ha de poder usar-les sense que li calgui saber-ne l'estructura interna: com s'emmagatzema l'estat dels objectes (els atributs) o de quina manera s'ha decidit implementar cert mètode. En aquest aspecte, es pot considerar que la manera ideal d'operar d'una classe ha de ser com ho faria un aparell autònom com, per exemple, un caixer automàtic: la persona que el fa servir pot treure'n profit sense haver de saber com funciona internament. El cas és que, donada una interacció possible amb l'aparell, aquest compleix l'ordre de la manera esperada i res més, per molt complex que sigui dur-la a terme internament. Per a qui l'està usant, aquesta complexitat necessària per dur a terme la tasca o accedir al que hi ha a dins del caixer són qüestions irrellevants, sempre que els diners acabin sortint al final del procés.

Un mètode és la implementació, el codi que s'executa, d'una operació.

Quan es defineix una classe, només s'ha de mostrar *què* poden fer els seus objectes, però no *com* ho fan, o mitjançant quines dades.



Un microxip, un exemple clàssic d'encapsulació i ocultació en informàtica.

Els objectius d'aplicar aquest principi són en gran mesura els següents:

- Minimitzar les implicacions de qualsevol modificació posterior a una classe, fent que aquest canvi es propagui el mínim possible dins de tot el programari desenvolupat.
- Permetre aplicar **programació defensiva**: forçar un major control d'errors, de manera que s'eviti que un objecte estigui en un estat que es pugui considerar invàlid o inconsistent.

El principi **d'encapsulació**, o d'**ocultació d'informació**, es basa en l'ocultació de les decisions de disseny, de manera que canvis en una classe afectin al mínim possible el programari ja desenvolupat.

Si bé en les publicacions s'usen els termes *encapsulació* i *ocultació d'informació* de manera indistinta, hi ha una certa discòrdia sobre si realment són exactament el mateix concepte. El principal motiu és la relació que hi ha entre els termes: l'encapsulació és un dels principis necessaris per assolir l'ocultació d'informació. Mentre que el segon es pot considerar la fita, el primer és la tècnica emprada.

Interfície d'una classe és el conjunt d'operacions que ofereix.

Un dels mecanismes bàsics per obtenir classes encapsulades correctament és establir una interfície estable en cada classe i no exposar cap camp de dades de

manera que sigui directament accessible. Això vol dir que, per defecte, tots els atributs d'una classe cal que siguin sempre privats, a menys que hi hagi un motiu molt bo per no fer-ho. Només ha de ser possible accedir als atributs per mitjà d'operacions accessores, tant per a la seva lectura com per modificar-ne el valor.

1.1.1 Minimització dels efectes de canvis

Suposem el cas en què es vol definir una classe en què s'emmagatzema una data concreta: la classe `Data`. Una de les decisions que el dissenyador ha de prendre en algun moment en definir-la és com s'emmagatzema la informació associada a un objecte d'aquesta classe, en forma d'atributs d'un tipus concret. Hi ha diverses estratègies per emmagatzemar una data, com per exemple:

- Tres atributs independents de tipus enter: dia, mes i any.
- Un únic atribut `String` amb un format concret, per exemple: "dia-mes-any".
- El nombre de dies que han passat des d'una data concreta. Tot i que aquesta opció sembla molt rebuscada, el llenguatge Java usa un sistema molt semblant per a la seva classe Date.

Si mantenim de manera estricta el principi d'ocultació d'informació oferint un conjunt d'operacions que ens permetin obtenir la informació necessària a cada moment, aquesta decisió serà irrellevant a l'hora de fer ús dels objectes de la classe `Data` i cridar operacions. Així, doncs, si es defineix un mètode getAny(): enter, quan s'implementi el mètode associat:

- Si s'ha triat la primera opció, n'hi ha prou de consultar l'atribut en què es guarda l'any.
- Si s'ha triat la segona, cal processar la cadena de text, recuperar l'any i transformar-la en enter.
- Amb la darrera opció, caldrà fer el càlcul de quants anys han passat, controlant els anys de traspàs, i retornar el resultat.

En qualsevol cas, per als objectes que criden el mètode `getAny`, el resultat és que s'obté l'any representat per aquell objecte de la classe `Data`, sense haver de preocupar-se de la complexitat real del procés ni de la seva estructura interna.

Un cop presa aquesta decisió, implementada la classe i utilitzada en diferents aplicacions, arriba el dia en què el dissenyador es veu forçat a canviar els tipus dels atributs per algun motiu. Un bon exemple, que va passar en la realitat, va ser l'efecte 2000.

L'efecte 2000

En una gran quantitat de sistemes informàtics, des de feia temps, la informació relativa a l'any, a les dates, s'emmagatzemava només en dues xifres, per tal d'estalviar espai de memòria (per exemple: 17 de maig del 1984, 17/05/84). En el seu moment, aquesta decisió era lògica, ja que la memòria era escassa i cara. Però aquesta decisió va provocar que, en apropar-se l'any 2000, s'haguessin de modificar tots els sistemes, ja que el dia 1 de gener del 2000 no serien capaços de distingir entre aquesta data o l'1 de gener del 1900, amb conseqüències incertes. Si bé s'anunciava un futur catastròfic, afortunadament al final no n'hi va haver per tant (tot i que molts programadors van estar molt entretinguts durant una bona temporada). Aquest esdeveniment es va anomenar *l'efecte 2000*.

L'efecte 2038

Els sistemes Unix emmagatzemen la data comptant el nombre de segons des de l'1 de gener del 1970. Donat que en aquests sistemes la xifra més alta que es pot representar és un nombre enter de 32 bits, quan el nombre de segons arribés a aquest màxim representable el comptador es desbordaria; això està previst que succeeixi l'any 2038.

Suposem que en la classe `Data` s'ha fet una elecció en què l'efecte 2000 té implicacions directes (per exemple, la segona opció). Si s'ha aplicat ocultació i només s'hi accedeix mitjançant la crida de mètodes, canviar el contingut de la classe `Data` és suficient. Caldrà modificar el tipus dels atributs i el codi dels mètodes accessors de manera que, mantenint el valor de retorn de les operacions, ara funcionin amb els nous tipus dels atributs. El canvi serà transparent per al codi de qualsevol programa en què s'usi la classe `Data` . Davant la crida del mètode `getAny`, se segueix obtenint exactament el mateix resultat, un enter amb l'any.

Si no s'ha aplicat ocultació i en el codi de totes les aplicacions s'ha accedit lliurement als atributs de la classe `Data`, en haver-los declarat públics, els desenvolupadors, a més de canviar els atributs d'aquesta classe, hauran de repassar tot el codi de tots els programes i modificar les línies en què s'accedeix a l'atribut d'acord amb el nou tipus. Això és moltíssima més feina.

Els que hagin decidit emmagatzemar la nova data en només quatre xifres tornaran a trobar el problema l'any 10000.

Mai no podem assumir que l'estructura interna d'una classe no canviarà en el futur.

Aquest cas també s'aplica en operacions, no de consulta, sinó de modificació d'atributs. Per exemple, el dissenyador pot haver decidit especificar una operació `setAny(any: enter)` per assignar un any a una data, com també haver especificat que la informació es desi en un únic atribut corresponent als dies que han passat des d'una data concreta. En aquest cas, el mètode a implementar haurà de preveure tots els càlculs necessaris en el codi, de manera que el valor final de l'atribut sigui correcte. Novament, la complexitat del procés queda totalment oculta.

1.1.2 Programació defensiva

Una altra de les motivacions per ocultar la informació emmagatzemada dins els objectes és la programació defensiva.

Programar defensivament implica garantir que l'estat d'un objecte sempre serà consistent i els seus atributs tindran assignats valors considerats correctes.

Cal garantir aquesta consistència independentment dels paràmetres amb què es cridin les seves operacions. S'utilitza aquest terme perquè la consistència s'ha de garantir fins i tot en el cas imaginari que les operacions es cridessin usant paràmetres expressament incorrectes amb l'únic objectiu de deixar l'objecte en un estat inconsistent, i no simplement per error involuntari. Tot i que aquest cas no sempre té sentit, sempre cal desenvolupar programari com si es considerés cert.

Aquest objectiu s'assoleix exclusivament en l'àmbit de la implementació, no del disseny, ja que es fa mitjançant el control dels paràmetres en el codi dels mètodes accessors. La seva ubicació aquí és coherent amb el principi d'ocultació d'informació, ja que el format real dels atributs d'un objecte és ocult i, per tant, el format que es considera correcte per a cada atribut només es pot saber internament en la classe. En darrera instància, el codi de cada classe és el responsable final de controlar quins valors es consideren correctes o incorrectes per a cada atribut, i ha de garantir que mai no s'assignarà cap valor incorrecte.

La programació defensiva té molt sentit quan es vol garantir la seguretat d'un sistema.

Programació defensiva de la classe "Data"

En l'exemple de la classe `Data`, hi ha un gran nombre de valors que es consideren incorrectes: valors negatius, els números de mes que no estan entre 1 i 12, dies superiors a 30 per a alguns mesos, o superiors a 28, 29 o 31 per a d'altres, etc. Cal evitar que, per cap concepte, s'arribi a propagar cap d'aquests valors als atributs dels objectes de la classe `Data`, sigui quin sigui el format final amb què el dissenyador ha triat representar una data internament. Per tant, cada mètode accessor ha de contenir codi que faci aquest control abans de modificar definitivament el valor de cap atribut.

1.1.3 Aplicació d'ocultació de dades

Un dels objectius de la programació orientada a objectes és l'**encapsulació** de dades i de mètodes de manera que els programadors usuaris d'una classe només poden accedir a les dades mitjançant els mètodes que la mateixa classe proporciona.

Ocultació de mètodes

Pot tenir sentit l'ocultació de mètodes? La resposta és afirmativa, ja que en el disseny d'una classe pot interessar desenvolupar un mètode intern per ser cridat en el disseny d'altres mètodes de la classe i no es vol donar a conèixer a la comunitat de programadors que utilitzaran la classe.

Amb l'encapsulació de dades i de mètodes s'aconsegueix:

- Protegir les dades de modificacions impròpies.
- Facilitar el manteniment de la classe, ja que si per algun motiu es creu que cal efectuar alguna reestructuració de dades o de funcionament intern, es podran efectuar els canvis pertinents sense afectar les aplicacions desenvolupades (sempre que no es modifiquin els prototipus dels mètodes existents).

Així, doncs, ens interessa ocultar les dades i, potser, alguns mètodes.

Donada la següent classe Persona, si desenvolupem un programa que instanciï objectes de la classe, no hauríem de tenir accés directe a les dades dni, nom i edat. Però, hi tenim accés?

```
//Fitxer Persona.java

public class Persona {
   public String dni;
   public String nom;
   public short edat;

   // Retorna: 0 si s'ha pogut canviar el dni
   //          1 si el nou dni no és correcte – No s'efectua el canvi
   public int setDni(String nouDni) {
      // Aquí hi podria haver una rutina de verificació del dni
      // i actuar en conseqüència. Com que no la incorporem,
      // retornem sempre 0
      dni = nouDni;
      return 0;
   }

   public  void setNom(String nouNom) {
      nom = nouNom;
   }

   // Retorna: 0 si s'ha pogut canviar l'edat
   //          1 : Error per passar una edat negativa
   //          2 : Error per passar una edat "enorme"

   public  int setEdat(int novaEdat) {
      if (novaEdat<0) return 1;
      if (novaEdat>Short.MAX_VALUE) return 2;
      edat = (short)novaEdat;
      return 0;
   }

   public String getDni() { return dni; }

   public String getNom() { return nom; }

   public short getEdat() { return edat; }

   public void visualitzar() {
      System.out.println("Dni...........:" + dni);
      System.out.println("Nom...........:" + nom);
      System.out.println("Edat..........:" + edat);
   }

   public static void main(String args[]) {
      Persona p1 = new Persona();
      Persona p2 = new Persona();
      p1.setDni("00000000");
      p1.setNom("Pepe Gotera");
      p1.setEdat(33);
      System.out.println("Visualització de persona p1:");
      p1.visualitzar();
      System.out.println("El dni de p1 és " + p1.getDni());
      System.out.println("El nom de p1 és " + p1.getNom());
      System.out.println("L'edat de p1 és " + p1.getEdat());
      System.out.println("Visualització de persona p2:");
      p2.visualitzar();
   }
}
```

Per comprovar si es garanteix l'ocultació de dades en aquesta classe, considerem el programa següent en què es creen objectes de la classe Persona dissenyada en l'arxiu Persona.java.

```
//Fitxer CridaPersona.java

public class CridaPersona{
```

```
4     public static void main(String args[]) {
5        Persona p = new Persona();
6        p.dni = "--$%#@--";
7        p.nom = "";
8        p.edat = -23;
9        System.out.println("Visualització de la persona p:");
10       p.visualitzar();
11    }
12 }
```

En aquest cas estem en un programa extern a la classe Persona i es veu com accedim directament a les dades "dni", "nom" i "edat" de la persona creada, i podem fer autèntiques animalades. El compilador no es queixa (cal haver compilat també l'arxiu Persona.java en el mateix directori) i l'execució dóna el resultat:

```
1 Visualització de la persona p:
2 Dni..........:--$%#@--
3 Nom..........:
4 Edat.........:-23
```

Acabem de veure, doncs, que la nostra versió de la classe `Persona` no oculta les dades i això és perquè en la definició d'aquestes dades no s'ha posat al davant el modificador adequat que controla l'ocultació. És a dir, la definició d'una dada i/o un mètode pot incloure un modificador que indiqui el tipus d'accés que es permet a la dada i/o mètode, segons la sintaxi següent:

```
1 [<modificadorAccés>] [<altresModificadors>] <tipusDada> <nomDada>;
2
3 [<modificadorAccés>] [<altresModificadors>] <tipusRetorn> <nomMètode> (<
      llistaArgs>)
4 {...}
```

El modificador d'accés pot prendre quatre valors:

- **public**, que dóna accés a tothom;
- **private**, que prohibeix l'accés a tothom menys pels mètodes de la pròpia classe;
- **protected**, que es comporta com a public per a les classes derivades de la classe i com a private per a la resta de classes;
- **sense modificador**, que es comporta com a public per a les classes del mateix paquet i com a private per a la resta de classes.

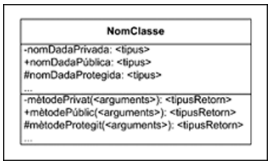
La classe `Persona` ha usat el modificador `public` per especificar els seus atributs. Per tant, la classe `CridaPersona` hi ha tingut accés total.

Sembla lògic, doncs, fer evolucionar la versió actual de la classe Persona cap a una classe que tingui les dades declarades com a privades i els mètodes com a públics. Fixem-nos que el mètode `main` per comprovar el funcionament d'una classe sempre ha estat declarat public.

A continuació presentem una versió evolucionada de la classe Persona que inclou els modificadors d'accés adequats: dades a private i mètodes a public.

Paquets o "packages"

Les classes es poden organitzar en paquets i aquesta possibilitat s'acostuma a utilitzar quan tenim un conjunt de classes relacionades entre elles. Totes les classes no incloses explícitament en cap paquet i que estan situades en un mateix directori es consideren d'un mateix paquet.



Notació d'una classe

Notació

A la notació per a una classe s'aprecien dues zones:

- Zona per a les dades.
- Zona per als mètodes.

Notació per als modificadors d'accés als membres:

- Prefix - per als privats.
- Prefix + per als públics.
- Prefix # per als protegits.

```
//Fitxer Persona.java

public class Persona
{
   private String dni;
   private String nom;
   private short edat;

   ...
```

Amb aquesta versió de la classe `Persona`, vegem què succeeix quan intentem compilar la classe `CridaPersona` que crea una persona i intenta accedir directament a les dades:

```
CridaPersona.java:11: dni has private access in Persona
   p.dni = "--$%#@--";
   ^
CridaPersona.java:12: nom has private access in Persona
   p.nom = "";
   ^
CridaPersona.java:13: edat has private access in Persona
   p.edat = -23;
   ^
3 errors
```

Fixem-nos que el compilador ja detecta que no hi ha accés a les dades. Hem aconseguit el nostre objectiu: protegir les dades tot ocultant-les a qui no les ha de veure.

1.2 Herència

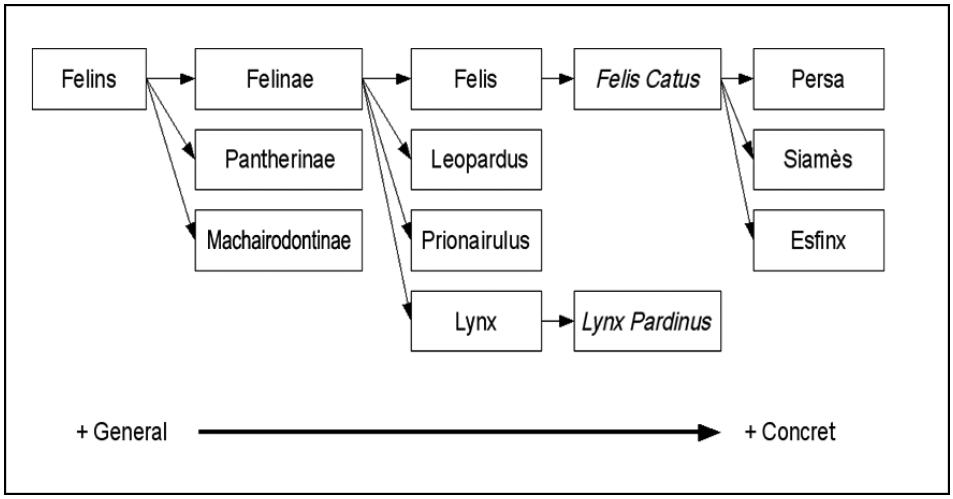
La paraula taxonomia prové del grec *taxis* que vol dir divisió i *nomos* que significa d'acord al que el que està establert, a la llei.

Una eina molt útil per fer aplicacions escalables és l'herència. Aquest concepte està força vinculat a la taxonomia, la ciència de la classificació mitjançant l'organització dels elements en grups segons les seves relacions de similitud. Això és molt útil en el camp de la biologia en la classificació dels éssers vius. Quan s'usa en aquest camp, sol tenir una estructura jeràrquica en funció d'espècies, gèneres, famílies, etc. La figura 1.1 mostra un exemple senzill d'aquest concepte, sense cap intenció de ser complet.

Crear taxonomies és útil perquè permet definir les característiques d'un conjunt d'elements partint d'una descripció general i, a poc a poc, concretar fins a arribar als elements més concrets. En la figura, es parteix des de l'esquerra, amb el major grau de generalització, i a mesura que ens desplacem cap a la dreta s'arriba a un major grau de concreció. Dins un mateix nivell de la taxonomia, cada conjunt es distingeix dels altres per certes diferències, però tots comparteixen les propietats del nivell anterior. Així, doncs, un gat comú (*Felis catus*) comparteix tot un conjunt de característiques amb un linx ibèric (*Lynx pardinus*): és un mamífer placentari, té pupil·les verticals per veure de nit i urpes, etc. Totes aquestes característiques comunes venen donades pel fet que ambdós pertanyen a la família dels felins. Però, tot i així, hi ha certes diferències: la mida, costums d'alimentació, hàbitat, període de gestació, etc. Per aquest motiu els biòlegs han decidit que

pertanyen a espècies diferents. El resultat final és que, un cop establertes les característiques d'un felí, ja no cal tornar-les a especificar per als nivells més concrets, es donen per conegudes. Igualment, donat un gat comú i un linx, i independentment de les diferències, es considera que tots dos pertanyen a la família dels felins.

FIGURA 1.1. Una taxonomia per classificar felins



Atès que l'orientació a objectes es basa en la descomposició d'un problema en conjunts d'elements d'acord amb les seves propietats i el seu comportament, no és descabellada la idea d'aprofitar aquesta estratègia per obtenir exactament els mateixos beneficis que en la biologia. Concretament, se'n fa ús a l'hora d'especificar classes.


Un gat és alhora un felí.

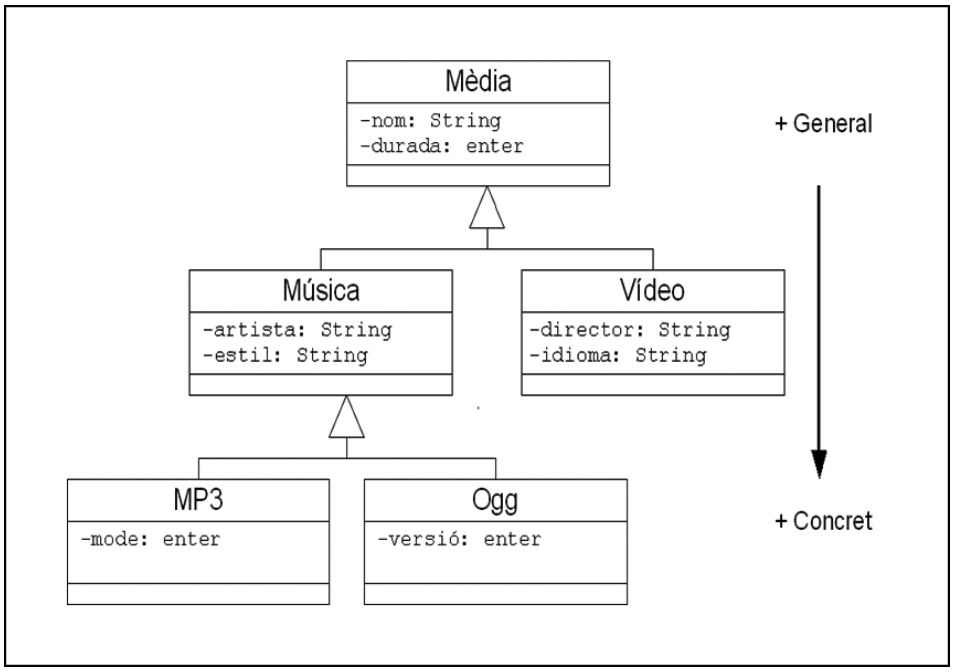
S'anomena **herència** la capacitat de definir una classe tan sols especificant-ne les diferències respecte a una altra classe prèviament definida. En la resta d'aspectes, es considera que es comporta exactament igual.

1.2.1 Disseny amb herència

L'herència normalment es plasma de manera gràfica establint un esquema tal com es mostra en la figura 1.2. Quan s'aplica herència des d'una classe a una altra, es diu que la segona classe, la del nivell inferior, **hereta** de la primera. En la figura, per exemple, les classes `Música` i `Vídeo` hereten de la classe `Mèdia`. També se sol dir que, quan s'aplica herència, hi ha un cas o una relació de **generalització/especialització**.

MP3 i Ogg són dos formats diferents per codificar música. Mentre que el primer és propietari, el segon és lliure.

FIGURA 1.2. Representació gràfica d'herència



És possible definir diferents nivells d'herència, com també tenir diverses classes que hereten de la mateixa, de manera que es crea una **jerarquia de classes**. Els nivells superiors dins una jerarquia d'herència solen correspondre a descripcions més generals dels elements, mentre que els nivells inferiors corresponen a elements més concrets o especialitzats.

En l'herència també se sol dir que una superclasse és una classe pare i que una subclasse és una classe filla.

Donada una classe concreta dins la jerarquia, tota classe més general és la seva **superclasse**, mentre que en el cas d'una classe més concreta, es tracta de la **subclasse**.

Tant `Música` com `Mèdia` són superclasses de les classes `MP3` i `Ogg`. Recíprocament, aquestes dues classes són subclasses tant de `Música` com de `Mèdia`. `Vídeo` només és subclasse de `Mèdia`.

Crear una jerarquia de classes té dues conseqüències immediates a efectes pràctics, tant en el disseny com en la implementació. Aquests efectes són exactament els mateixos que s'han observat en la taxonomia de felins, però traduïts a classes i objectes:

1) Per una banda, qualsevol propietat o comportament definit en una classe es propaga a totes les seves subclasses. Això significa que qualsevol atribut o operació definida en una superclasse es considerarà que també existeix implícitament en totes les seves subclasses, sense necessitat de tornar-los a especificar en cada una. La propietat inversa no és certa: les operacions especificades en una subclasse no existeixen necessàriament en les superclasses.

Exemple: la classe "Mèdia"

En la classe `Mèdia` s'han especificat l'atribut i l'operació accessora associada:

- `- durada: enter`

- `+ getDurada(): enter`

Es considera que automàticament també existeixen en les classes `Música`, `MP3` i `Ogg`, sense que calgui especificar-ho en la llista d'atributs i operacions d'aquestes classes. Per tant, és possible cridar l'operació `getDurada` sobre l'objecte **laMevaCançó: MP3**.

La propagació de propietats dins la jerarquia de classes també s'aplica a les relacions entre classes definides en el diagrama estàtic UML. Això és lògic atès que, en darrera instància, aquestes s'implementen com a atributs.

Aquesta propietat de l'herència resulta ideal quan es volen estendre les funcionalitats d'una classe a partir d'una altra que ja està creada. Només caldrà definir la nova classe com a subclasse de la primera i afegir els fets diferencials (nous atributs i operacions). Això permet un cert grau de reaprofitament de feina. Un dels avantatges principals d'aplicar herència per assolir aquesta fita és que, per heretar d'una classe (en el disseny o a la implementació), no cal haver participat en la creació de la superclasse. No cal saber res més que el nom de la classe i quines operacions públiques ofereix. En el cas de la implementació, per heretar d'una classe ni tan sols en cal el codi font.

AAC són els inicials d'advanced audio coding, un format destinat en el seu dia a reemplaçar l'MP3.

Exemple: nous formats de música

Si un dia es crea una nova classe per suportar nous formats de música en el reproductor multimèdia, per exemple la classe `AAC`, n'hi haurà prou de fer que aquesta nova classe hereti de `Música` per reaprofitar tota la feina feta tant en la classe `Música` com en la classe `Mèdia`. Tots els atributs i operacions de totes dues classes s'hereten.

Quan es genera una jerarquia d'herència, des de zero o partint d'una ja existent, cal estudiar quins atributs i operacions són comuns als diferents conjunts de subclasses i definir-los sempre en la classe més general possible. L'herència ja fa que aquesta definició es propagui a les subclasses. D'acord amb la jerarquia de la figura 1.2, la classe en què s'especifiquen els atributs "nom" i "durada" i, implícitament, les seves operacions accessores, és `Mèdia`, ja que són una propietat que té qualsevol arxiu de mèdia. Seria incorrecte especificar-ho repetidament en totes les classes de la jerarquia. De manera general, si al crear una aplicació orientada a objectes apareixen classes amb moltes de les seves característiques repetides, és senyal que hi manca aplicar una herència per agrupar-les.

L'herència permet fer "factor comú" en especificar classes.

Així, doncs, l'herència també serveix com a mecanisme per evitar la duplicació d'operacions idèntiques i, per extensió, a l'hora d'implementar-les, de codi.

2) D'altra banda, un aspecte molt important i útil que aporta l'herència és la capacitat de disposar d'objectes que pertanyen a diverses classes alhora: objectes amb més d'un tipus. Donat un objecte d'una classe concreta, aquest objecte no solament serà del tipus que marca la seva classe, sinó que també ho serà de totes les seves superclasses.

Donada la jerarquia de classes de la figura 1.2:

- Els objectes de la classe `MP3` pertanyen a tres classes alhora: `MP3`, `Música` i `Mèdia`.
- Els objectes de la classe `Vídeo` pertanyen a dues classes: `Vídeo` i `Mèdia`.
- Els objectes de la classe `Mèdia` pertanyen a una única classe: `Mèdia`.

Exemple: la classe Lector

Suposem que es disposa dels següents atribut i operació definits en una classe Lector, que processa arxius de mèdia:

- `- mediaActual: Mèdia.`
- `+ reproduir (m: Mèdia).`

Donat un atribut, igual que amb totes les variables en qualsevol llenguatge de programació, orientat a objectes o no, sols és possible assignar-hi valors d'acord amb el tipus especificat. Per tant, a aquest atribut només es pot assignar un objecte de la classe Mèdia. El mecanisme d'herència permet assignar objectes que no són directament instàncies de la classe Mèdia (el tipus definit per l'atribut) i tot i així complir aquesta premissa: és possible assignar objectes que són instància de qualsevol subclasse de Mèdia, ja que es consideren objectes de la seva classe i Música alhora. Per tant, és correcte assignar tant l'objecte **unaMedia:Mèdia** com **unaCanço:Música**, **unaCançoMp3: MP3**, **unaCançoOgg:Ogg** o **unVideo:Vídeo**.

Aquest cas és aplicable també a crides de l'operació reproduir. Aquesta operació es pot cridar passant com a paràmetre qualsevol objecte de les classes `Mèdia`, `Música`, `Vídeo`, `MP3` i `Ogg` sense estar infringint cap norma respecte la concordança del tipus en el paràmetre d'entrada.

1.2.2 Modificadors d'accés i herència

Quan els modificadors d'accés defineixen atributs i operacions mantenen la seva validesa estrictament, fins i tot entre classes que hereten una de l'altra. Una superclasse pot continuar (i hauria de continuar) operant com una caixa negra amb vista a les seves subclasses sense cap problema. Aquest fet té una implicació interessant: si un atribut de la superclasse s'ha definit com a privat, la classe filla no podrà accedir-hi directament. Així, doncs, s'arriba a la situació que els objectes de la subclasse contenen els atributs de la superclasse, però no poden gestionar-los directament. La manera correcta és usar els mètodes accessors per fer-ho.

En el cas dels mètodes privats, tot i que també s'hereten, serà impossible cridar-los. Per tant, es tracta d'un altre cas especial, ja que tot i que es considera que la classe té l'operació especificada, no pot cridar-la.

Hi ha un tipus de visibilitat vinculat a les relacions d'herència: un **atribut o operació protegit**, associat a la paraula clau `protected` i que s'identifica amb el símbol #. Normalment, els llenguatges de programació interpreten aquesta visibilitat com si es considerés pública per a totes les subclasses i privada per a la resta. Per tant, totes les instàncies de subclasses poden accedir directament a l'atribut o cridar l'operació lliurement, però instàncies de qualsevol altra classe (incloent-hi superclasses), no.

Estrictament parlant, usar visibilitat protegida també va en contra del principi d'encapsulació. L'herència no és excusa per obviar que és molt millor que, per defecte, tots els atributs es defineixin com a privats i es generin les classes com a caixes negres.

1.2.3 Classes abstractes

Si es torna a fer una ullada a les taxonomies d'animals, hom es pot adonar d'una circumstància destacable: donada una classificació total, des del cas més general possible als més concrets de tots, en el món real només hi ha realment els elements de les classificacions més concretes. Així, doncs, hi ha gats perses, però no existeix cap animal que simplement sigui un felí i prou. Tot felí que existeix al món sempre és, en realitat i en darrera instància, algun cas més concret (un linx ibèric, un tigre de bengala, etc.). La majoria dels conjunts definits serveixen com a recull de propietats comunes, però no existeix cap element que pertanyi únicament i exclusivament als conjunts més generals.

Aquesta circumstància especial també es pot aplicar quan s'usa l'herència dins l'orientació a objectes: és possible definir classes que especifiquin un conjunt d'atributs i d'operacions, però de les quals no existeixin objectes. És a dir, classes que no es poden instanciar. Pel que fa a la implementació, qualsevol intent d'instanciar-les provocarà un error. Aquestes classes s'anomenen **classes abstractes** i el seu objectiu és facilitar la creació de jerarquies d'herència útils, que representin idees de les quals, si bé es pot definir un conjunt de propietats, no existeixen elements com a tals.

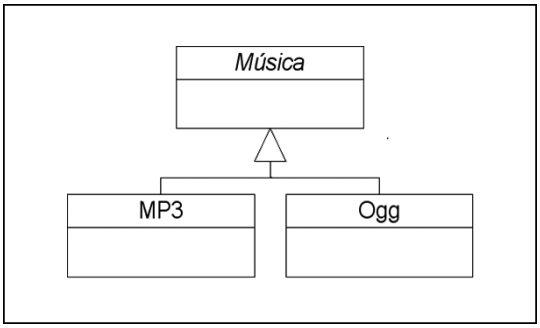
Una **classe abstracta** no es pot instanciar, ja que representa un concepte abstracte. Està pensada per operar com a superclasse d'altres classes en qualsevol nivell dins una jerarquia.

Exemple: música i dades en format "Ogg"

En una aplicació de reproducció de multimèdia que suporti els formats d'àudio MP3 i Ogg, tota la música que gestiona sempre serà o bé un conjunt de dades en format `MP3` o en format `Ogg`. És a dir, o bé un objecte de la classe `MP3` o un de la classe `Ogg`, però mai gestionarà objectes de la classe `Música` i prou. Per tant, `Música` es pot especificar com una classe abstracta. Fins i tot si se suporten nous formats de dades, els objectes sempre seran d'un format concret, mai simplement "de música".

Una classe abstracta es representa gràficament posant el seu nom en cursiva, d'acord amb la representació de la figura 1.3.

FIGURA 1.3. Representació gràfica de classe abstracta



L'existència de les classes abstractes en els nivells més generals d'una jerarquia no s'ha d'interpretar com que qualsevol superclasse ha de ser forçosament abstracta. Només ho serà quan es vulgui representar un concepte general, amb el qual es poden classificar altres elements, però que realment no existeix com a tal. En molts casos, al contrari que en una taxonomia, sí que té sentit que existeixin superclasses no abstractes.

DRM són les inicials de digital rights management (gestió de drets digitals).

El format Fairplay de la companyia informàtica Apple és una versió amb DRM del format AAC.

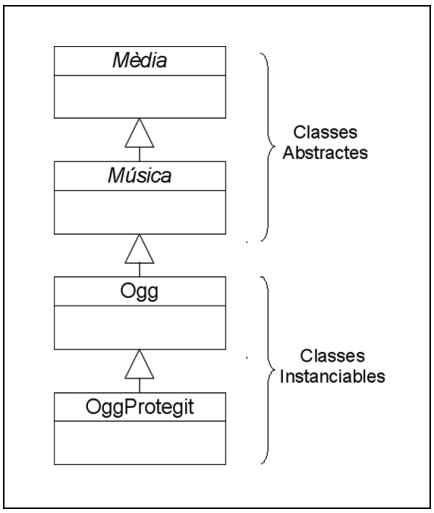
Exemple: extensió de la classe "Ogg"

Suposem que a l'aplicació de reproducció de música es vol afegir la funcionalitat d'interpretar un format nou de dades basat en l'Ogg, però amb sistemes de protecció de drets d'autor (DRM) incorporat. Un fitxer de música protegit per sistemes DRM està xifrat, de manera que un cop carregat dins un reproductor ja no és possible traspassar-lo a un altre mitjà de cap manera. Aquest nou format s'anomenarà `OggProtegit` i caldrà representar-lo amb una nova classe.

En aquest cas, el dissenyador pot decidir que aquest nou format no és res més que el format Ogg, ja que disposa exactament de les mateixes funcionalitats, però amb certes propietats addicionals: les dades estan xifrades i tenen una clau que les protegeix, calen operacions per xifrar/desxifrar les dades, etc. Per tant, és un cas ideal per aplicar herència, de manera que es reaprofita la feina feta a la classe `Ogg`. De fet, per fer-ho ni tan sols és necessari que el dissenyador hagi estat el creador original de la classe `Ogg`.

En la figura 1.4 es mostra com quedaria la nova jerarquia de classes per a aquest cas concret. Dins el programa podrien existir instàncies tant de la classe `Ogg`, que representarien peces musicals no protegides amb DRM, com de la classe `OggProtegit`, que representarien peces de música protegides. El que mai no hi ha són instàncies de les classes Música o Mèdia, ja que es tracta de classes abstractes.

FIGURA 1.4. Superclasses abstractes i no abstractes en diferents nivells d'una jerarquia.



Atès que una classe abstracta no es pot instanciar, mai no hi haurà objectes d'aquesta classe sobre els quals cridar operacions. Tot i així, especificar-hi operacions no és un fet il·lògic, ja que cal recordar que totes són heretades per les respectives subclasses i, si aquestes no són abstractes, es poden cridar. Aquest és, en definitiva, l'objectiu final de definir una classe abstracta.

1.2.4 Pèrdua d'identitat

En el moment en què un objecte pertany a més d'una classe alhora apareix la problemàtica anomenada **pèrdua d'identitat d'un objecte**. Aquesta està especialment vinculada a la implementació i, en conseqüència, és el programador qui haurà de tenir-la en compte. Per tant, es deixa momentàniament la visió del dissenyador i s'entra en la del programador (tot i que un dissenyador també ha de saber que existeix).

Atès que sota una jerarquia de classes un objecte té diversos tipus, és possible assignar-lo a qualsevol variable "x" d'algun dels tipus als qual pertany: la seva classe directament o qualsevol de les seves superclasses. Si x està definida com de la mateixa classe que l'objecte, no passa res; però en el moment en què "x" ha estat definida com d'una de les superclasses de l'objecte, el que passa és que, a nivell de codi, aquest objecte passa a comportar-se exactament com un objecte de tipus igual al de la variable. Tot i que internament l'objecte no ha variat, s'ha perdut la capacitat d'operar-hi amb totes les seves funcionalitats originals.

La implementació d'una operació s'anomena un *mètode*.

Aquesta particularitat s'entén molt millor amb un exemple concret:

Exemple: "Mèdia" i les seves subclasses

Suposem que existeix una classe `Lector`, que s'encarrega de processar arxius de mèdia, on s'ha especificat l'operació següent:

- `+reproduir (m: Mèdia)`

Respecte a aquesta operació, es compleix el següent:

a) Donada aquesta definició, en implementar-la, dins el codi del mètode associat hi haurà una variable m, que serà de la classe Mèdia. Dins el codi es podrà manipular l'objecte passat com a paràmetre per mitjà d'aquesta variable (bàsicament, cridar operacions). Això no és específic de l'orientació a objectes, passa en tots els llenguatges de programació.

b) Atès que m està definida com de la classe Mèdia, a través de m només es poden cridar operacions disponibles en aquesta classe (especificades directament o heretades de les seves superclasses).

c) Per les propietats de multiplicitat de tipus del mecanisme d'herència, a aquest mètode se li pot passar com a paràmetre un objecte de les classes `Música`, `Vídeo`, `MP3` o `Ogg` (o `OggProtegit`). En fer-ho, aquest objecte queda assignat a la variable "m".

Arribat a aquest punt, pels apartats b) i c) es pot veure que hi ha una circumstància especial: tot i que en la variable "m" hi ha assignat un objecte de tipus `Música`, `Vídeo`, `MP3` o `Ogg`, només es poden cridar les operacions disponibles per a la classe `Mèdia`. Les operacions que només existeixen en altres classes es tornen inaccessibles, tot i que els objectes internament no han canviat de classe.

L'objecte passat a través del paràmetre "m" ha "perdut la seva identitat".

La pèrdua d'identitat dels objectes és una conclusió lògica, ja que quan es disposa d'una variable "x" definida com d'una classe concreta és impossible saber *a priori* la classe real de l'objecte assignat quan hi ha herència implicada. Per tant, no ha de ser possible cridar operacions de subclasses, en no poder garantir el tipus de l'objecte realment assignat. Només es pot garantir que, sigui quin sigui l'objecte,

almenys serà de la classe exacta amb què s'ha definit la variable "x". Aquesta suposició ha de ser certa per força, ja que en cas contrari hi hauria una assignació errònia de tipus i el compilador hauria generat un error. Mai no s'arribaria a poder executar el programa sense complir aquesta condició.

De totes maneres, la majoria dels llenguatges orientats a objectes ofereixen mecanismes per esbrinar la classe original dels objectes assignats a una variable.

1.2.5 Herència múltiple

De la mateixa manera que una classe pot tenir diverses subclasses a un nivell immediatament inferior dins una jerarquia, res no impedeix el cas exactament invers. Aquest cas s'anomena herència múltiple, ja que una classe hereta directament de múltiples superclasses.

L'**herència múltiple** descriu la situació en què una classe hereta directament de més d'una classe.

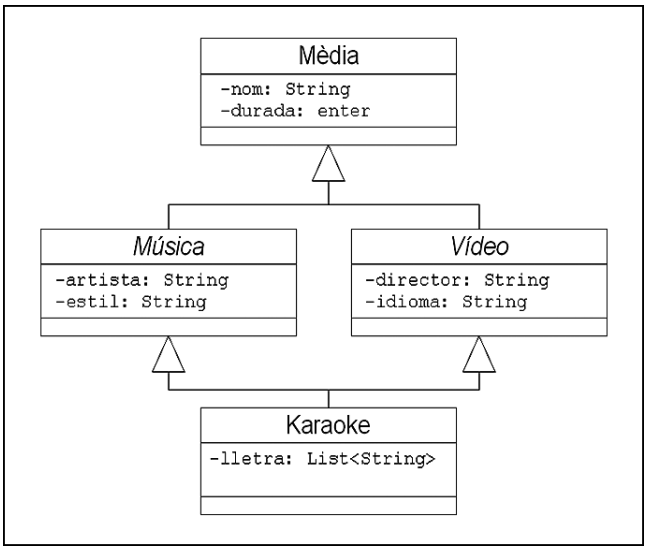
En un cas com aquest, totes les propietats de l'herència es mantenen: les successives subclasses hereten les propietats de totes les seves superclasses, independentment que sigui per una única via totalment lineal o per diversos camins.

Exemple: el karaoke

Afegint noves funcionalitats a un reproductor d'arxius multimèdia, el dissenyador decideix la possibilitat d'interpretar dades que codifiquin una cançó de karaoke. Com a decisió de disseny, considera que aquest element és una unió de les propietats d'un vídeo (una pista d'imatges en moviment, un director, un idioma, etc.) i les propietats d'una peça musical (d'un cert cantant i estil, amb una pista d'àudio independent, etc.). A més a més, té les seves característiques exclusives que el fan diferent d'aquestes dues classes ja definides (unes lletres sincronitzades en la pista d'àudio).

Per tant, té sentit aplicar herència múltiple per reaprofitar tota la feina ja feta en les classes `Música` i `Vídeo`, tal com es mostra en la figura 1.5. D'acord amb aquesta figura, les instàncies de la classe `Karaoke` tenen tots els atributs definits en cadascuna de les quatre classes.

FIGURA 1.5. Exemple d'herència múltiple.



1.2.6 Definició de subclasses

En el llenguatge Java, la definició de subclasses s'efectua amb la paraula reservada `extends` en la declaració de la classe, seguint la sintaxi següent:

```
[final] [public] class <NomClasse> extends <NomClasseBase>
{
<CosDeLaClasse>
}
```

De manera automàtica, tots els membres (dades i mètodes) que té la classe base també resideixen en les subclasses, amb excepció del(s) constructor(s), el(s) qual(s), en cas de ser necessari(s), s'haura(n) de dissenyar en les classes derivades.

Només hi ha dues maneres perquè una classe tingui constructor(s): que s'hi defineixi(n) o que, per manca de definició, la classe incorpori el constructor per defecte que proporciona el llenguatge Java.

La definició d'una subclasse ha d'incloure les dades, els iniciadors i els mètodes adequats a l'especialització de subclasse respecte a la classe base i s'afegeixen a les dades i mètodes heretats de la classe base.

En la definició de les dades i els mètodes d'una subclasse cal aplicar els modificadors d'accés (`private`, `public`, `protected` o inexistent) que corresponguin.

Recordem que el codi incorporat en una subclasse (iniciadors i mètodes) té accés a tots els membres `public` i `protected` de la classe base i, si ambdues classes estan en el mateix paquet, també tindrà accés a tots els membres que no incorporen cap modificador d'accés.

Indepedentment que, al generar una classe, s'usi la sentència `extends` per incloure-la dins una jerarquia d'herència, totes les classes del Java sempre hereten d'una classe especial, inclosa a les llibreries principals de Java. Java ho fa automàticament. Es tracta de la classe **Object**. Aquesta és la superclasse més general de qualsevol classe al Java.

1.2.7 Inicialització de subclasses

Els passos que segueix la màquina virtual en la construcció d'un objecte amb l'execució de l'operador `new` són els següents:

1. Reserva memòria per desar el nou objecte i totes les seves dades són inicialitzades amb valor zero pels tipus enters, reals i caràcter, amb valor `false` pel tipus booleà, i amb valor `null` per les variables referència.
2. S'executen les inicialitzacions explícites.
3. S'executen els iniciadors (blocs de codi sense nom) que existeixin dins la classe seguint l'ordre d'aparició dins la classe.
4. S'executa el constructor indicat en la construcció de l'objecte amb l'operador `new`.

Com incideix l'herència en aquest procés quan es crea un objecte d'una classe derivada? El model de seguretat de Java obliga a executar les tres darreres fases en cada classe implicada, seguint l'ordre que marca la jerarquia de classes i començant per la classe de dalt. És a dir, en una situació en què la classe C deriva de la classe B, i aquesta de la classe A, en la construcció d'un objecte de la classe C, primer es reserva memòria per a totes les dades (fase 1) i després s'executen les fases 2-3-4 per la part de l'objecte provinent de l'herència de la classe A, posteriorment les fases 2-3-4 per la part de l'objecte provinent de l'herència de la classe B, i finalment les fases 2-3-4 per la part de l'objecte provinent de les dades declarades en la classe C.

En dissenyar el constructor d'una classe derivada, apareix el problema de com podem indicar quin dels constructors de la classe base s'ha d'executar i amb quins paràmetres.

El llenguatge Java proporciona la paraula reservada super a utilitzar com a nom de mètode per cridar un constructor de la classe base, seguint la sintaxi:

```
1 super(<llistaParàmetres>)
```

La utilització de la paraula reservada `super` com a mètode per cridar un constructor de la classe base en el disseny d'un constructor en la classe derivada només es pot efectuar en la primera sentència del nou constructor.

Si en el disseny del constructor de la classe derivada, enlloc d'efectuar una crida a `super(...)`, s'efectua una crida a `this(...)`, Java traspassa tota la responsabilitat de construcció al constructor cridat amb la crida `this(...)`, el qual pot contenir una altra crida `this(...)` o una crida `super (...)` o cap de les dues.

Si en el disseny d'un constructor no s'explicita cap crida `super(...)` ni cap crida `this(...)`, Java crida implícitament el constructor per defecte de la classe base (constructor sense arguments) i, si no existeix, el compilador genera un error.

Disseny de la classe Alumne derivada de la classe Persona

Es vol dissenyar la classe `Alumne` com una especificació de la classe `Persona` afegint-hi el concepte corresponent al nivell d'estudis que cursa l'estudiant, el qual ha de permetre la gestió dels valors següents: B per a batxillerat, M per als cicles formatius de grau mitjà, S per als cicles formatius de grau superior, i ? per al cas en què el nivell d'estudis sigui desconegut. A continuació, proposem un possible disseny, que incorpora molts constructors per exemplificar la `Persona` utilització de les crides super(...) i `this(...)`. Així mateix, s'ha cregut oportú fer evolucionar la darrera versió de la classe Persona cap a una classe que tingui les dades declarades com a `protected`, de manera que la classe `Alumne` hi té accés directe i en el disseny no ens veiem obligats a utilitzar les funcions accessores (*getter* i *setter*).

```
1  //Fitxer Persona.java
2  public class Persona {
3     protected String dni;
4     protected String nom;
5     protected short edat;
6
7     public Persona () {}
8     public Persona (String sDni, String sNom, int nEdat) {
9        dni = sDni;
10       nom = sNom;
11       if (nEdat>=0 && nEdat<=Short.MAX_VALUE)
12          edat = (short)nEdat;
13    }
14    public Persona (Persona p) {
15       this (p.dni, p.nom, p.edat);
16    }
17    public Persona clonar () {
18       return new Persona (this);
19    }
20
21    // Retorna: 0 si s'ha pogut canviar el dni
22    //          1 si el nou dni no és correcte – No s'efectua el
          canvi
23    public int setDni(String nouDni) {
24       // Aquí hi podria haver una rutina de verificació del dni
25       // i actuar en conseqüència. Com que no la incorporem,
             retornem sempre 0
26       dni = nouDni;
27       return 0;
28    }
29
30    public void setNom(String nouNom) {
31       nom = nouNom;
32    }
33
34    // Retorna: 0 si s'ha pogut canviar l'edat
35    //          1 : Error per passar una edat negativa
36    //          2 : Error per passar una edat "enorme"
37    public int setEdat(int novaEdat) {
38       if (novaEdat<0) return 1;
39       if (novaEdat>Short.MAX_VALUE) return 2;
```

```
40       edat = (short)novaEdat;
41       return 0;
42    }
43
44    public String getDni() { return dni; }
45    public String getNom() { return nom; }
46    public short getEdat() { return edat; }
47
48    public void visualitzar() {
49       System.out.println("Dni..........:" + dni);
50       System.out.println("Nom..........:" + nom);
51       System.out.println("Edat.........:" + edat);
52    }
53
54    public final boolean equals (Object obj) {
55       if (obj == this) return true;
56       if (obj == null) return false;
57       if (obj instanceof Persona) return dni.equals(((Persona)obj
             ).dni);
58       return false;
59    }
60
61    public int hashCode() {
62         int hash = 3;
63         hash = 89 * hash + Objects.hashCode(this.dni);
64         return hash;
65    }
66
67    public String toString() {
68       return dni + " - " + nom;
69    }
70 }
```

```
1  //Fitxer Alumne.java
2  public class Alumne extends Persona {
3     private char nivell = '?';
4     /** Valors vàlids:
5         B = Batxillerat
6         M = Cicle Fromatiu Mitjà
7         S = Cicle Formatiu Superior
8         ? = Desconegut
9     */
10
11    public Alumne () {}
12    public Alumne (String sDni, String sNom, int nEdat, char
          cNivell) {
13       super (sDni, sNom, nEdat);
14       nivell = validarNivell (cNivell);
15    }
16   public Alumne (Persona p, char cNivell) {
17       super (p);
18       nivell = validarNivell (cNivell);
19    }
20    public Alumne (Persona p) {
21       this (p, '?');
22    }
23    public Alumne (String sDni, String sNom, int nEdat) {
24       this (sDni, sNom, nEdat, '?');
25    }
26    public Alumne (Alumne a) {
27       this (a.dni, a.nom, a.edat, a.nivell);
28    }
29
30    public void setNivell (char nouNivell) {
31       nivell = validarNivell (nouNivell);
32    }
33
```

```
34 /** Valida el "nivell" passat per paràmetre, de manera que
       retorna el valor
35 vàlid i en majúscula. Si no era vàlid, retorna '?'. */
36    private char validarNivell (char nivell)  {
37       nivell = Character.toUpperCase(nivell);
38       if (nivell!='B' && nivell!='S' && nivell!='M') nivell='?';
39       return nivell;
40    }
41    public char getNivell () { return nivell; }
42    public void visualitzar () {
43       super.visualitzar ();
44       System.out.println("Nivell........:" + nivell);
45    }
46    public String toString() {
47       String s = "Dni: " + dni + " - Nom: " + nom + " - Edat: " +
             edat + " - Nivell: ";
48       switch (nivell) {
49          case 'B': s = s + "Batxillerat"; break;
50          case 'M': s = s + "Cicle F. Mitjà"; break;
51          case 'S': s = s + "Cicle F. Superior"; break;
52          default : s = s + "???"; break;
53       }
54       return s;
55    }
56    public static void main(String args[]) {
57       Alumne t[] = new Alumne[6];
58       t[0] = new Alumne();
59       t[1] = new Alumne("00000000","Pepe Gotera",33,'b');
60       t[2] = new Alumne("00000000","Pepe Gotera",33);
61       t[3] = new Alumne(new Persona("00000000","Pepe Gotera",33),
             'b');
62       t[4] = new Alumne(new Persona("00000000","Pepe Gotera",33))
             ;
63       t[5] = new Alumne(t[3]);
64       for (int i=0; i<t.length; i++) System.out.println(t[i]);
65    }
66 }
```

Com es veu, aquesta classe `Alumne` conté un munt de constructors. La quantitat de constructors possiblement és excessiva i n'hi ha d'innecessaris, però s'han incorporat per exemplificar la utilització de les crides super(...) i `this(...)`, i el mètode `main()` que inclou la classe `Alumne` efectua la creació de sis objectes de la classe per comprovar el funcionament de tots els constructors dissenyats. Si executem aquesta classe passa això:

```
1 Dni: null - Nom: null - Edat: 0 - Nivell: ???
2 Dni: 00000000 - Nom: Pepe Gotera - Edat: 33 - Nivell: Batxillerat
3 Dni: 00000000 - Nom: Pepe Gotera - Edat: 33 - Nivell: ???
4 Dni: 00000000 - Nom: Pepe Gotera - Edat: 33 - Nivell: Batxillerat
5 Dni: 00000000 - Nom: Pepe Gotera - Edat: 33 - Nivell: ???
6 Dni: 00000000 - Nom: Pepe Gotera - Edat: 33 - Nivell: Batxillerat
```

1.3 Polimorfisme

Com s'ha pogut veure, la idea darrera del mecanisme d'herència és poder definir fàcilment noves classes a partir de la suposició que un seguit de propietats comunes, especificades en una superclasse, es propaguen automàticament a tot el conjunt de les subclasses. A partir d'aquest punt, en la nova classe només cal definir-ne els trets diferencials que la concreten.

Fins aquest moment, s'ha assumit que aquests trets diferencials són incrementals: s'especifiquen nous atributs o noves operacions en la classe, que se sumen als heretats. Però, què passa si la diferenciació que es vol fer és modificar el comportament d'una operació ja prèviament definida en una superclasse perquè es comporti de manera diferent? En aquesta situació és quan entra en joc el mecanisme del polimorfisme.

El mot *polimorfisme* prové del grec *poly*, que significa ‚moltes', i *morfos* que significa ‚formes'. Per tant, significa "moltes formes".

El **polimorfisme** consisteix en la possibilitat d'aplicar una mateixa operació a objectes de diferents classes, però cridant una implementació diferent segons la classe a la qual pertanyi l'objecte sobre el qual es crida.

Els felins i els polimorfisme

Un exemple del concepte de polimorfisme de caràcter molt general és el següent. Si es pren la taxonomia dels felins, tots tenen la capacitat d'emetre sons amb la boca; tots comparteixen aquest comportament. Tot i així, el resultat d'aquesta propietat és diferent segons quin felí concret l'executa: els gats miolen i les panteres o els lleons rugeixen. El motiu és que es usen un os especial (l'hioide) que les panteres i els lleons tenen en forma de cartílag. No és una simple qüestió de to o volum.**Com** emeten l'efecte sonor és diferent. Per tant, la capacitat d'emetre sons dels felins és polimòrfica: es fa de maneres diferents segons a quin tipus pertany l'individu que l'executa.

1.3.1 Sobreescriptura d'operacions

El primer pas per veure com s'aplica aquest mecanisme dins una jerarquia de classes és entendre el concepte de **sobreescriptura** d'operacions (*override*, en el terme original anglès).

Sobreescriure una operació vol dir tornar-la a definir en una subclasse, de manera que el mètode associat i, per tant, el codi que s'executa, tingui un comportament diferent al de la superclasse.

Atès que s'està parlant de mètodes, els efectes del polimorfisme es veuen reflectits exclusivament en la implementació. De totes maneres, en el disseny cal indicar que una operació s'ha sobreescrit. Això es fa tornant-la a especificar en les subclasses, exactament igual, tot i ja existir en una superclasse. Un cop una operació ha estat sobreescrita, les subclasses successives, a mesura que es descendeix per la jerarquia, hereten la darrera versió, a menys que elles també decideixin sobreescriure-la. No hi ha límit al nombre de vegades que una operació pot ser sobreescrita dins una jerarquia de classes.

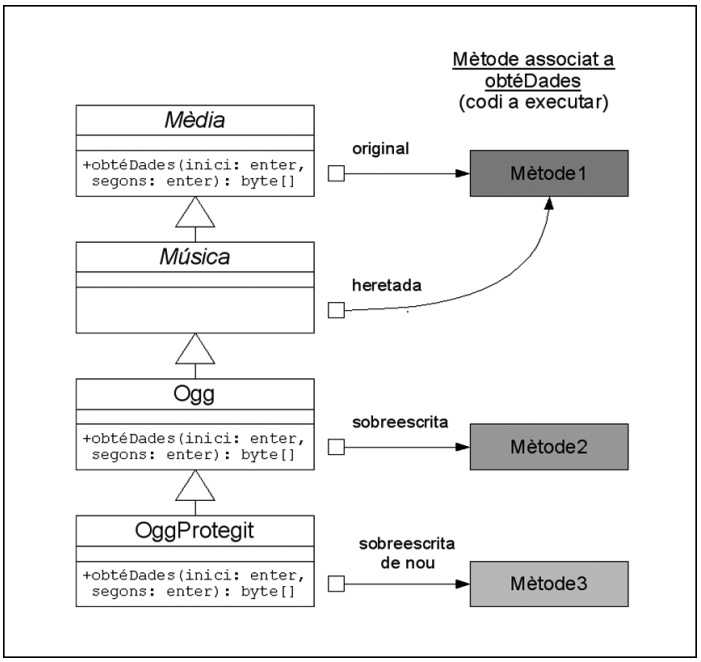
Classes abstractes

Mai no es donarà el cas que es cridi una operació sobre un objecte de la classe `Música`, ja que és abstracta i no es pot instanciar.

Exemple: la jerarquia de formats de música

Per exemple, segons la classe `Mèdia` a la figura 1.6, l'operació `obtéDades`, s'ha redefinit en la classe `Ogg`. Aquest exemple té sentit, ja que segons el format de les dades a tractar, aquestes s'han de processar de diferent manera per tal d'extreure un bloc d'una durada concreta. Així, doncs, el mètode que realment s'executa quan es crida aquesta operació sobre un objecte de la classe `Ogg` és diferent del que s'ha definit en la classe Mèdia. La classe Música conserva l'especificació i el mètode de Mèdia, heretats. En la classe `OggProtegit` s'ha tornat a redefinir i, per tant, el mètode associat és novament diferent.

FIGURA 1.6. Sobreescriptura d'operacions.



Mitjançant la redefinició, és possible usar herència sense estar forçats a usar tot el comportament heretat de les superclasses tal com s'ha especificat en aquestes. És possible mantenir només una part, indicant quines són les diferències respecte al comportament inicial.

1.3.2 Operacions polimòrfiques

Quan sobre un objecte donat es crida una operació que la seva classe ha sobreescrit, el mètode que s'executa sempre és el definit en la classe a què pertany l'objecte, independentment del tipus de la variable on estigui contingut. Aquest comportament és el que es coneix com a *polimorfisme* dins l'orientació a objectes.

La propietat polimòrfica de les operacions és especialment rellevant quan un objecte ha perdut la identitat: ha estat assignat a una variable definida amb el tipus d'una superclasse, de manera que només és possible cridar operacions especificades en aquesta superclasse. Tot i aquesta circumstància, quan es crida una operació polimòrfica, el codi que s'acaba executant és el relatiu al tipus real de l'objecte, no el de la superclasse.

Exemple: el lector de formats de música

En la classe `Lector`, encarregada de processar arxius de mèdia, s'ha especificat l'operació següent:

- `+ reproduir (m: Mèdia)`

Independentment del tipus d'objecte que es passi com a paràmetre, aquest queda emmagatzemat en una variable de tipus Mèdia (la variable m). Pel fenomen de la pèrdua d'identitat, només és possible cridar sobre aquesta variable operacions especificades en

la classe Mèdia independentment del tipus real de l'objecte, però atès que l'operació `obtéDades` està especificada en la classe `Mèdia`, és possible cridar-la. Quan això es fa, succeeix el següent, prenent com a referència la figura 1.6:

Si l'objecte emmagatzemat en m és de tipus `Ogg`, quan es cridi `obtéDades` s'executarà `Mètode2`.

Si l'objecte emmagatzemat en m és de tipus `OggProtegit`, quan es cridi `obtéDades`, s'executarà `Mètode3`.

En cap cas s'executa `Mètode1`, tot i que m està definida de tipus `Mèdia` en l'operació. Perquè s'executés el codi de `Mètode1` caldria passar un objecte o bé de tipus `Mèdia`, bé de tipus `Música` (en heretar el mètode, ja que no el sobreescriu), però en aquest exemple concret mai no serà possible, ja que són classes abstractes i no es poden instanciar.

Així, doncs, tot i que l'objecte està emmagatzemat en una variable de tipus Mèdia i l'operació que s'ha cridat és una d'especificada en aquesta classe, mitjançant polimorfisme, el mètode que realment s'executa és l'associat al tipus real de l'objecte emmagatzemat.

Així, doncs, el mecanisme de polimorfisme es fonamenta en el fet que, donada una mateixa operació especificada a diferents classes, tot i no saber el tipus real d'un objecte sobre la qual es crida (a causa de la pèrdua d'identitat) fa que s'executi el mètode.

És interessant saber que en les publicacions també es considera polimorfisme la propietat dels objectes de poder ser usats en operacions en què el tipus dels paràmetres pertany a alguna de les seves superclasses: la capacitat que un objecte tingui més d'un tipus i, per tant, "moltes formes". Es considera una mena especial de polimorfisme, anomenada **polimorfisme paramètric**. En contraposició, el cas vinculat a la crida d'operacions es pot anomenar **polimorfisme ad hoc**. Tot i així, quan es parla simplement de polimorfisme, sense especificar res més, normalment es fa referència a polimorfisme *ad hoc*.

1.3.3 Operacions abstractes

Al implementar mètodes polimòrfics donada una jerarquia d'herència, pot donar-se una circumstància un xic estranya. Suposem que ha arribat el moment d'implementar el mètode associat a l'operació `obtéDades` de la classe `Música`. Què hauria de fer el seu codi? En principi, processar les dades depèn del format final de la música, de manera que aquesta és una pregunta que no té resposta quan es parla de música com a concepte general. Només és possible respondre-la per a les subclasses que indiquen un format concret de les dades: `Ogg`, `OggProtegit`, `MP3`, etc. En casos com aquest és possible especificar una operació com a abstracta.

Llenguatges orientats a objectes

Cada llenguatge orientat a objectes té la seva sintaxi per indicar que un mètode correspon a una operació abstracta, i, per tant, no s'ha de codificar.

En especificar una **operació** com a **abstracta**, s'indica que no té cap mètode associat, és a dir, que a l'hora d'implementar-la no es codifica. El codi a executar s'obtindrà a partir de la sobreescriptura duta a terme per alguna de les seves subclasses.

Una operació abstracta s'especifica escrivint-la en cursiva:

+obtéDades(inici: enter, segons: enter): byte[]

Tota classe que conté alguna operació abstracta s'ha d'especificar com a classe abstracta. Atès que no és possible instanciar objectes d'una classe abstracta, s'impedeix que es doni el cas que una operació sense cap mètode associat es pugui cridar.

Error en la programació

Si no es sobreescriuen totes les operacions abstractes heretades, el compilador del llenguatge de programació sempre retorna un error.

En el moment d'especificar una subclasse d'una classe amb operacions abstractes, hi ha dues opcions: o bé se sobreescriuen totes les operacions abstractes heretades, o bé només es fa per a una part d'elles (o per a cap). En el darrer cas, atès que es considera que conté les operacions abstractes heretades, s'ha d'especificar aquesta classe també com a abstracta. Un cop una operació abstracta s'ha sobreescrit amb una versió no abstracta dins la jerarquia de classes, les subclasses successives ja no estan obligades a sobreescriure-la.

Mai no es pot donar el cas que sigui possible instanciar una classe que no té un mètode associat per a cadascuna de les seves operacions.

1.3.4 Polimorfisme i herència múltiple

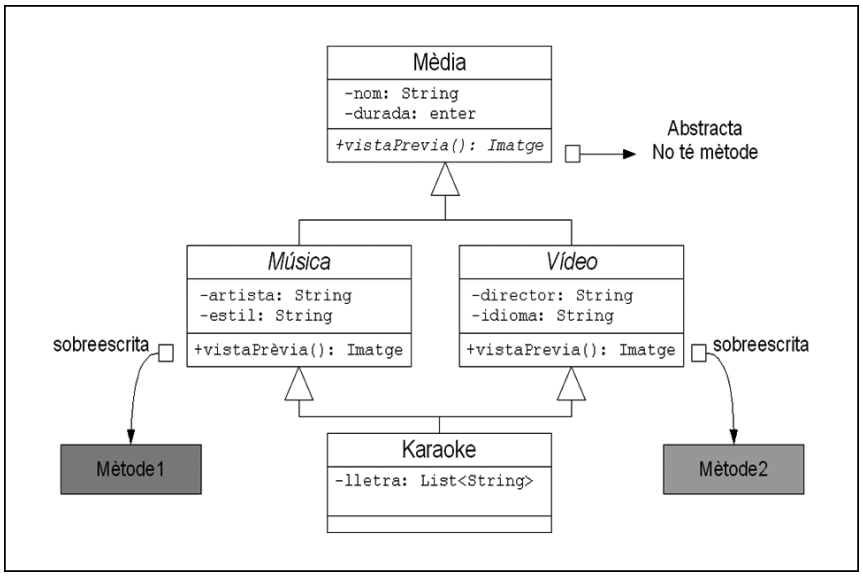
Tot i que l'herència múltiple pot servir per multiplicar la capacitat d'estendre codi i reaprofitar feina, hi ha un problema greu en usar-la quan es permeten mètodes polimòrfics: el problema del diamant. Aquest problema consisteix en la incapacitat de determinar quin mètode cal executar realment quan hi ha duplicitat de noms en una operació redefinida dins una jerarquia de classes.

El problema del diamant

L'origen del nom del problema del diamant prové del fet que el conjunt de línies que representen les relacions d'herència múltiple en notació UML formen la figura d'un rombe.

En la figura 1.7 es pot apreciar aquesta problemàtica. Tant en la classe `Música` com en la classe `Vídeo` l'operació `vistaPrèvia` especificada originalment en `Mèdia` ha esta sobreescrita. Per a cada cas el mètode associat és diferent, ja que cal fer una acció diferent per generar la imatge resultant (per exemple, la informació sobreimpresa depèn dels atributs de cadascú). Per herència, és possible cridar aquesta operació sobre els objectes de la classe `Karaoke`. El problema és que, en aquesta situació, resulta impossible decidir quin dels dos mètodes cal executar realment.

FIGURA 1.7. El problema del diamant.



Per implementar herència múltiple en Java s'utilitza el que s'anomenen *interfaces*.

En conseqüència, cal tenir molt present aquesta problemàtica quan s'utilitza l'herència múltiple. Una manera senzilla de resoldre-la és sobreescriure novament l'operació conflictiva en la subclasse en què dóna problemes (`Karaoke`, a la figura), de manera que aquesta estigui associada a un tercer mètode diferent. Tot i existir aquesta solució, no tots els llenguatges orientats a objectes suporten herència múltiple directament, cosa que pot comportar bastants problemes a l'hora de fer la implementació. Per exemple, si bé C++ la suporta, fer-ho pot arribar a tenir un cert grau de complexitat. Altres llenguatges, com el Java, directament no la suporten, i ofereixen mecanismes alternatius per obtenir una funcionalitat semblant (si bé no igual). Sempre val la pena valorar si realment la solució creada només es pot fer mitjançant herència múltiple.

1.3.5 Aplicacions del polimorfisme

Un cop s'ha presentat la manera d'utilitzar el concepte de polimorfisme, val la pena veure amb un exemple concret com la seva aplicació correcta aporta una gran escalabilitat a un sistema, a la vegada que permet seguir fàcilment els principis de cohesió i d'ocultació d'informació.

Suposem el cas d'un reproductor multimèdia, amb una classe `Lector` que processa arxius de mèdia de diferents formats. Si, sense necessitat d'entrar a nivell de codi, es reflexiona sobre l'operació `reproduir` de la classe `Lector`, bàsicament el que fa és anar obtenint dades des de la classe `Mèdia` mitjançant l'operació `obtéDades` en un format que és capaç d'entendre. Aquestes dades les va recuperant a poc a poc en blocs d'una durada concreta i les va enviant als altaveus.

Suposem que es vol donar suport progressivament a diferents formats de mèdia. De fet, no solament de música, sinó també de vídeo. Atès que cada format de

dades és diferent, l'operació `obtéDades` ha de fer tasques diferents segons quines dades està processant. Per tant, en darrera instància, el mètode associat serà un enorme bloc condicional. Cada cop que es vulgui suportar un nou format, caldrà tenir el codi font de la classe Mèdia i afegir una nova condició al codi. El resultat és:

- Poca cohesió, ja que hi ha una classe que fa moltes coses molt diferents (gestiona molts formats de dades diferents).
- Cal tenir el codi font per afegir funcionalitats.

Si s'aplica herència i polimorfisme, es pot crear una aplicació més escalable. En aquest cas, quan es vulgui suportar un nou format de dades, n'hi ha prou de generar una nova classe que hereti de `Música`. Atès que la nova classe hereta de `Mèdia`, els seus objectes també són d'aquest tipus i, per tant, la nova classe pot ser passada com a paràmetre al mètode reproduir de `Lector`. Aquesta nova classe sobreescriurà l'operació `obtéDades`, i assignarà un mètode que tractarà les dades segons el nou format. Atès que `obtéDades` s'ha especificat com a operació abstracta en `Música`, es garanteix que la nova classe l'haurà de sobreescriure forçosa- ment si vol ser instanciada. Per tant, el codi associat a reproduir pot garantir que sempre existeix un mètode associat a `obtéDades` per a qualsevol instància d'una subclasse de Mèdia. El resultat és:

- Un gran manteniment dels principis de cohesió i d'ocultació d'informació. Cada classe vinculada a un format de música gestiona únicament i exclusivament el seu format de música sense haver de conèixer res de la resta.
- Si es vol donar suport a un nou format, es genera una nova classe, però no cal modificar en absolut cap de les classes ja existents. Això vol dir que ni tan sols cal el codi font de totes les parts ja programades per afegir funcionalitats al programa.

Aquest darrer punt és especialment important, ja que s'està assolint una operació (reproduir) que pot ser codificada de manera que funcioni fins i tot amb classes que encara no s'han generat, ja que ni tan sols han estat ideades. Aquesta és la potència del polimorfisme: d'una banda, les operacions abstractes permeten garantir que, donada una classe, totes les seves subclasses també la tindran sempre. D'altra banda, la multiplicitat de tipus de l'herència permet processar instàncies de classes concretes (`Ogg`, `MP3`) a partir de paràmetres definits com de classes generals (`Mèdia`).

1.3.6 Exemples de sobreescriptura de mètodes

En el llenguatge Java, per tal d'aplicar sobreescriptura d'algun mètode a una subclasse cal recordar les següents regles, que cal tenir en compte:

- El nom i la llista i ordre dels arguments han de ser iguals al del mètode de la classe base que es vol sobreescriure.
- El tipus de retorn de tots dos mètodes ha de ser igual.
- El mètode de la classe derivada no pot ser menys accessible que el de la classe pare.
- El mètode de la classe derivada no pot provocar més excepcions que el mètode del pare.

Polimorfisme de dades

Quan es pot accedir a un objecte d'una classe amb una variable de referència d'una classe situada per damunt de segons la jerarquia de classes, es diu que les variables de referència són polimòrfiques.

Com a resultat, la versió heretada d'un mètode sobreescrit desapareix en la classe derivada, però no desapareixen els mètodes heretats que eren sobrecàrregues del mètode sobreescrit. Respecte a l'accés als mètodes, hem de tenir en compte també que:

- Si dins una classe cal accedir a la versió de la superclasse per un mètode sobreescrit, disposem de la paraula reservada `super` amb la sintaxi: `super.nomMètode(<paràmetres>)`.
- En el llenguatge Java, els mètodes són polimòrfics. Sobre un objecte d'una classe `Z` al qual s'accedeix amb una variable de referència d'una classe `X` situada per damunt de `Z` segons la jerarquia de classes, es pot cridar qualsevol mètode dels definits en la classe `X` (a la qual pertany la variable de referència) però s'executarà la versió del mètode existent en la classe `Z` (a la qual pertany l'objecte).

És a dir, quin mètode s'executa en les quatre darreres línies del fragment de codi següent?

```
1  public class X {
2    met1 () {...codi X...}
3  }
4
5  public class Z extends X {
6    met1 () {...codi Z...} // Sobreescriptura de met1() d'X
7    met2 () {...}  // Mètode inexistent a la classe X
8  }
9
10 X ox = new X ();
11 X oz = new Z ();
12
13 ox.met1();  // (1)
14 oz.met1();  // (2)
15 ox.met2();  // (3)
16 oz.met2();  // (4)
```

És clar que en la instrucció (1) s'executarà la versió de `met1()` de la classe `X`, ja que tant la variable de referència "ox" com l'objecte són de la classe `X`. En la instrucció (2) s'executarà la versió de `met(1)` de la classe `Z`, ja que preval la classe a la qual pertany l'objecte per damunt de la classe a la qual pertany la variable "oz" emprada per fer referència a l'objecte. Les instruccions (3) i (4) són errònies i el compilador no les accepta perquè, en la classe a què pertanyen les variables de referència "ox" i "oz", no existeix cap mètode anomenat `met2()`. Si es vol

aplicar el mètode `met2()` de la classe `Z` a l'objecte apuntat per "oz", cal aplicar una conversió cast de la variable "oz" cap a la classe `Z` tot escrivint:

```
((Z)oz).met2();
```

Per veure millor com funciona la sobreescriptura, el millor és estudiar alguns exemples concrets,. Primer veiem un exemple general. Tot seguit , és interessant conèixer quatre mètodes de la classe `Object` (heretats en totes les classes del Java) per als quals en pot ser necessària o convenient la sobreescriptura: `finalize()`, `equals()` i `toString()`.

Exemple general de sobreescriptura de dades i de mètodes

El programa següent declara tres classes: `A`, `B` derivada d'`A` i `C` derivada de `B`. La classe `A` conté la dada "d" i el mètode `xxx()`, que són sobreescrits en les classes `B` i `C`. La classe `A` també conté el mètode `xxx(int x)`, que no és sobreescrit en cap classe.

```
public class A {
   protected int d = 10;

   public void xxx() { System.out.println ("d en A = " + d); }

   public void xxx(int x) {
      char aux;
      if (this instanceof C) aux = 'C';
      else if (this instanceof B) aux = 'B';
      else aux = 'A';
      System.out.println ("Sóc xxx d'A aplicat sobre un objecte de la classe "
          + aux );
   }
}

public class B extends A
{
   protected int d=20;

   public void xxx() {
      System.out.println ("d en B = " + d);
      super.xxx();
   }
}

public class C extends B
{
   protected int d = 30;

   void xxx()
   {
      System.out.println ("d en C = " + d);
      super.xxx();
   }

   public void visibilitat () {
      System.out.println ("Des del mètode \"visibilitat\" en C:");
      System.out.println ("d en C = " + d);
      System.out.println ("d en B = " + super.d);
   }

   public static void main (String args[]) {
      int aux;
      A oa = new A();
```

```
44        A ob = new B();
45        A oc = new C();
46        ((C)oc).visibilitat();   // (1)
47        System.out.println("Crides al mètode xxx() existent a les tres classes:")
              ;
48        oa.xxx();   // (2)
49        ob.xxx();   // (3)
50        oc.xxx();   // (4)
51        System.out.println("Crides al mètode xxx(int x) existent a les tres
              classes:");
52        oa.xxx(0);   // (5)
53        ob.xxx(0);   // (6)
54        oc.xxx(0);   // (7)
55     }
56  }
```

En el mètode `main()` de la classe `C` declarem un objecte per a cadascuna de les classes `A`, `B` i `C`, apuntat cada un per variables de la classe `A`.

El mètode `visibilitat()` de la classe `C` exemplifica com es pot utilitzar la paraula super per accedir a una dada heretada sobreescrita. La seva execució (1) ens ho demostra.

En cridar (2) el mètode `xxx()` per a l'objecte de la classe `A` apuntat per "oa" s'executa el mètode `xxx()` de la classe `A`.

En cridar (3) el mètode `xxx()` per a l'objecte de la classe `B` apuntat per "ob" s'executa el mètode `xxx()` de la classe `B` el qual, al seu torn, mitjançant la paraula `super`, crida el mètode `xxx()` de la classe `A`.

En cridar (4) el mètode `xxx()` per a l'objecte de la classe `C` apuntat per "oc" s'executa el mètode `xxx()` de la classe `C` el qual, al seu torn, mitjançant paraula super, crida el mètode `xxx()` de la classe `B`, que mitjançant la paraula super, crida el mètode `xxx()` de la classe `A`.

Les crides (5), (6) i (7) del mètode `xxx(int x)` executen, en qualsevol cas, el mètode `xxx(int x)` de la classe `A`, heretat en les classes `B` i `C`.

Per demostrar totes aquestes afirmacions només cal compilar i executar el fitxer i observar els missatges que es visualitzen:

```
1  Des del mètode "visibilitat" en C:
2  d en C = 30
3  d en B = 20
4  Crides al mètode xxx() existent a les tres classes:
5  d en A = 10
6  d en B = 20
7  d en A = 10
8  d en C = 30
9  d en B = 20
10 d en A = 10
11 Crides al mètode xxx(int x) existent a les tres classes:
12 Sóc xxx d'A aplicat sobre un objecte de la classe A
13 Sóc xxx d'A aplicat sobre un objecte de la classe B
14 Sóc xxx d'A aplicat sobre un objecte de la classe C
```

Sobreescriptura del mètode finalize()

El mètode `finalize()`, definit en la classe `Object` i, per tant, existent per herència en totes les classes, és cridat de manera automàtica pel recuperador de memòria just abans de destruir un objecte i cal sobreescriure'l en les classes en què pertoqui efectuar alguna actuació abans de destruirne els objectes.

Si, a banda d'indicar-hi les instruccions corresponents a l'actuació que pertoqui, cal mantenir les instruccions de finalització que hi pogués haver dissenyades en la classe base, cal dissenyar el mètode de manera similar a:

```
void finalize() {
  <codi_corresponent_a_l'actuació>
  super.finalize();
}
```

Sobreescriptura del mètode equals()

El llenguatge Java proporciona l'operador de comparació "==", que, aplicat sobre dades de tipus primitius, compara si les dues dades contenen el mateix valor, i aplicat sobre referències a objectes compara si les dues referències fan referència a un mateix objecte. Aquest operador s'ha d'usar amb una mica més de cura quan s'opera amb objectes.

Exemple de comparació de cadenes mitjançant l'operador de comparació

Sovint tindrem la necessitat de comparar cadenes. Considerem, com a exemple, el programa següent, en el qual tenim diferents dni i volem comparar-los:

```
public class CompararStringsViaOperadorComparacio
{
   public static void main (String args[]) {
      String dni1 = "00000000";
      String dni2 = "00000000";
      String dni3 = new String("00000000");
      char t[] = {'0','0','0','0','0','0','0','0'};
      String dni4 = new String(t);
      System.out.println("dni1 : " + dni1);
      System.out.println("dni2 : " + dni2);
      System.out.println("dni3 : " + dni3);
      System.out.println("dni4 : " + dni4);
      System.out.println("dni1 == dni2 : " + (dni1 == dni2));
      System.out.println("dni1 == dni3 : " + (dni1 == dni3));
      System.out.println("dni1 == dni4 : " + (dni1 == dni4));
      System.out.println("dni3 == dni4 : " + (dni3 == dni4));
   }
}
```

Si executem el programa, obtenim:

```
dni1 : 00000000
dni2 : 00000000
dni3 : 00000000
dni4 : 00000000
dni1 == dni2 : true
dni1 == dni3 : false
dni1 == dni4 : false
dni3 == dni4 : false
```

Les quatre primeres visualitzacions ens deixen clar, per si teníem algun dubte, que les quatre referències "dni1", "dni2", "dni3" i "dni4" a objectes `String` fan referència a objectes amb el mateix contingut ("00000000"). Però, com ens expliquem els resultats de les quatre comparacions posteriors?

La resposta està en que cal tenir en compte que l'operador de comparació "==" compara el valor de les referències als objectes i dóna resultat cert únicament si les referències comparades apunten el mateix objecte. Però no el contingut dels objectes en si. Per tant, és lògic el resultat `false` de les tres darreres comparacions ("dni1 == dni3" i "dni1 == dni4" i "dni3 == dni4"), ja que les referències "dni3" i "dni4" apunten a objectes `String` creats amb l'operador `new`, fet que provoca que veritablement s'estigui creant un nou objecte `String` a partir del paràmetre indicat. Segurament de vegades ens interessarà comparar el contingut de les cadenes, de manera que el resultat de les quatre darreres comparacions sigui cert.

Així, doncs, donat el funcionament de l'operador "==", el resultat de les tres darreres comparacions és correcte, però, per què la comparació "dni1 == dni2" dóna resultat `true`? La resposta és que davant l'aparició, en un codi font, d'un literal `String` com ha succeït en l'exemple ("00000000"), el compilador crea un objecte per al literal i totes les aparicions del literal es converteixen en referències a l'objecte. Per aquest motiu, dni1 i dni2 estan apuntant al mateix objecte. Aquest comportament no és perillós en el llenguatge Java, ja que cal recordar que els objectes `String` són immutables, és a dir, no es poden canviar una vegada creats.

Sembla que ja tenim clar el funcionament de l'operador "==". Ara ens cal algun mecanisme per poder comparar el contingut dels objectes apuntats per referències enlloc de les referències pròpiament.

És clar que en moltes classes (per no dir totes) pot ser necessari disposar d'algun mecanisme per comprovar si dos objectes són iguals o no, a partir d'un criteri determinat respecte al seu contingut, i això no ho proporciona l'operador "==". Amb aquest propòsit, Java proporciona un mètode a la classe `Object`, que s'hereta en totes les classes i ens proposa la seva utilització en les diverses classes després de la sobreescriptura. És el mètode següent:

```
1 public boolean equals (Object obj)
```

La implementació d'aquest mètode en la classe `Object` (que és la que s'hereta en cas de no sobreescriure'l) retorna el resultat de la comparació "==" entre la referència que apunta l'objecte sobre el qual s'aplica el mètode i la referència passada com a paràmetre. És a dir, si no se sobreescriu, resulta que `x.equals(y)` dóna el mateix resultat que "x == y".

La classe `String` incorpora una versió del mètode `equals()` que haurem d'utilitzar sempre que necessitem saber si el contingut dels objectes apuntats per dues referències a `String` coincideix o no. Per exemple, el següent programa ens mostra la utilització del mètode `equals()` per comparar cadenes i la diferència de resultats respecte la utilització de l'operador "==".

```
1  public class CompararStringsViaMetodeEquals {
2     public static void main (String args[]) {
3        String dni1 = "00000000";
4        String dni2 = "00000000";
5        String dni3 = new String("00000000");
6        char t[] = {'0','0','0','0','0','0','0','0'};
7        String dni4 = new String(t);
8        System.out.println("dni1 : " + dni1);
9        System.out.println("dni2 : " + dni2);
10       System.out.println("dni3 : " + dni3);
11       System.out.println("dni4 : " + dni4);
12       System.out.print("dni1 == dni2 : " + (dni1 == dni2));
```

```
13        System.out.println("\tdni1.equals(dni2) : " + dni1.equals(dni2));
14        System.out.print("dni1 == dni3 : " + (dni1 == dni3));
15        System.out.println("\tdni1.equals(dni3) : " + dni1.equals(dni3));
16        System.out.print("dni1 == dni4 : " + (dni1 == dni4));
17        System.out.println("\tdni1.equals(dni4) : " + dni1.equals(dni4));
18        System.out.print("dni3 == dni4 : " + (dni3 == dni4));
19        System.out.println("\tdni3.equals(dni4) : " + dni3.equals(dni4));
20     }
21  }
```

Si executem el programa, obtenim els resultats esperats:

```
dni1 : 00000000
dni2 : 00000000
dni3 : 00000000
dni4 : 00000000
dni1 == dni2 : truedni1.equals(dni2) : true
dni1 == dni3 : falsedni1.equals(dni3) : true
dni1 == dni4 : falsedni1.equals(dni4) : true
dni3 == dni4 : falsedni3.equals(dni4) : true
```

Com a primera aplicació de la sobreescriptura del mètode `equals()`, podem pensar en la seva sobreescriptura en una classe anomenada `Persona`, tenint en compte que considerarem que dos objectes Persona són iguals si tenen el mateix dni:

```
public boolean equals (Object obj) {
  if (obj == this) return true;
  if (obj == null) return false; (3)
  if (obj.getClass() != this.getClass()) return false; // (1)
  return dni.equals(((Persona)obj).dni); // (2)
}
```

Veiem que, perquè es tracti de la sobreescriptura del mètode `equals` heretat, cal que el paràmetre es declari de la classe Object, fet que fa possible la comparació d'un objecte de la nostra classe (`Persona`) amb un objecte apuntat per una referència a qualsevol classe.

Però, llavors, es necessita comprovar si els dos objectes són de la mateixaget classe (1), utilitzant el mètode `getClass()` de la classe `Object`, i si ho són cal fer la conversió `cast` (2) de la referència a `Object` passada per paràmetre per tractar l'objecte apuntat com un objecte de la classe `Persona` i poder accedir al seu dni.

També és important no oblidar les comprovacions sobre si la referència passada per paràmetre és `null` (3), ja que no comprovar-ho provocaria un error en temps d'execució si el valor del paràmetre fos `null`.

Però, la implementació presentada us sembla correcta? Suposem que tenim la classe `Alumne`, que hereta de la classe `Persona`, i plantegem-nos el següent:

```
Persona p = new Persona (...);
Alumne a = new Alumne (...);
```

Si en algun moment decidim cridar `p.equals(a)` per saber si tots dos objectes són iguals segons la definició convinguda (mateix dni), esperarem que el resultat sigui cert si ambdós objectes tenen el mateix dni, i fals en cas contrari. La implementació anterior del mètode `equals()` sempre donaria fals, ja que la

comparació (1) referent a si són objectes de la mateixa classe té resultat fals. Per tant, si volem que la comparació de dni sigui efectiva en les classes derivades de `Persona`, cal canviar la implementació:

```
public boolean equals (Object obj) {
  if (obj == this) return true;
  if (obj == null) return false;
  if (obj instanceof Persona) return dni.equals(((Persona)obj).dni);
  return false;
}
```

Per acabar, cal comentar que també seria possible la implementació següent:

```
public final boolean equals (Persona obj) {
  if (obj == this) return true;
  if (obj == null) return false;
  return dni.equals(obj.dni);
}
```

Aquesta implementació difereix de l'anterior en el fet que el paràmetre és una referència a `Persona` i, per tant, no cal comprovar si l'objecte passat per paràmetre és comparable amb l'objecte sobre el qual s'està aplicant el mètode equals(). Però aquest mètode no és la sobreescriptura del mètode `equals()` de la classe `Object` i, per tant, amb aquesta implementació, la nostra classe disposaria de dos mètodes equals():

```
public boolean equals (Object obj); // Heretat d'Object
public boolean equals (Persona obj); // Nou a la classe
```

Normalment, s'aconsella sobreescriure el mètode `equals()` de la classe `Object` enlloc de crear nous mètodes `equals()`. En el cas de sobreescriure aquest mètode, la documentació oficial de Java recomana sobreescriure també el mètode `hashCode()` per assegurar que dos objectes que han resultat iguals amb `equals()`, donaran el mateix resultat amb `hashCode()`.

Sobreescriptura del mètode toString()

¿Alguna vegada heu provat d'executar `System.out.println(obj)` en què "obj" fa referència a un objecte d'una classe qualsevol dissenyada per vosaltres, com la classe Persona? El compilador s'ho empassa? En cas afirmatiu, què es visualitza? Provem-ho!

Dissenyem el mètode main()següent a la classe `Persona`:

```
public static void main (String args[]) {
   Persona p = new Persona ("00000000", "Pepe Gotera", 33);
   System.out.println (p);
}
```

Les ordres següents ens mostren que la compilació s'efectua sense problemes i l'execució també s'efectua, però visualitzant una informació desconeguda per nosaltres:

```
Persona@3e25a5
```

El mètode `System.out.println()` està pensat per mostrar una cadena i, si com a paràmetre se li indica una referència a un objecte, la màquina virtual Java crea una representació String de l'objecte, aplicant automàticament sobre l'objecte el mètode `toString()` de la classe `Object` que, per herència, existeix en totes les classes dissenyades i que caldrà tenir sobreescrit en les diverses classes amb la implementació que correspongui.

A banda que la màquina virtual cridi automàticament el mètode `toString()` sobre un objecte quan ho consideri convenient, sempre que es vulgui es pot cridar com un mètode qualsevol: `obj.toString()`.

La màquina virtual Java utilitza el mètode `toString()` en qualsevol lloc on necessiti tenir una representació en cadena d'un objecte i, és clar, la conversió proporcionada pel mètode heretat de la classe Object no acostuma a ser útil. Ens convé, doncs, sobreescriure'l.

El mètode toString

Segons la documentació de Java, la implementació d'aquest mètode en la classe `Object` (que és la que s'hereta en cas de no sobreescriure'l) retorna una cadena igual al nom de la classe i la seva referència.

La sobreescriptura del mètode `toString()` en la classe `Persona` podria ser la següent si convenim que la representació en cadena d'una persona sigui la concatenació del seu dni i del seu nom separats per un guió:

```
1 public String toString() {
2   return dni + " – " + nom;
3 }
```

1.4 Interfaces Java

Suposem una situació en què ens interessa deixar constància que certes classes han d'implementar una funcionalitat teòrica determinada, diferent en cada classe afectada. Estem parlant, doncs, de la definició d'un mètode teòric que algunes classes hauran d'implementar.

Un exemple real pot ser el mètode `calculImportJubilacio()` aplicable, de manera diferent, a moltes tipologies de treballadors i, per tant, podríem pensar a dissenyar una classe `Treballador` en què un dels seus mètodes fos `calculImportJubilacio()`. Aquesta solució és vàlida si estem dissenyant una jerarquia de classes a partir de la classe `Treballador` de la qual pengin les classes corresponents a les diferents tipologies de treballadors (metal·lúrgics, hostaleria, informàtics, professors...). A més, disposem del concepte de classe abstracta perquè cada subclasse implementi obligatòriament el mètode `calculImportJubilacio()`.

Però, i si resulta que ja tenim les classes `Professor`, `Informatic`, `Hostaleria` en altres jerarquies de classes? La solució consistent a fer que aquestes classes derivessin de la classe `Treballador`, sense abandonar la derivació que poguessin tenir, seria factible en llenguatges orientats a objectes que suportessin l'herència múltiple, però això no és factible en el llenguatge Java.

Per superar aquesta limitació, Java proporciona les *interfaces*.

Una **interface** és una maqueta contenidora d'una llista de mètodes abstractes i dades membre (de tipus primitius o de classes). Els **atributs**, si existeixen, són implícitament considerades static i final. Els **mètodes**, si existeixen, són implícitament considerats public.

Per entendre en què ens poden ajudar les *interface*, ens cal saber:

- Una *interface* pot ser implementada per múltiples classes, de manera similar a com una classe pot ser superclasse de múltiples classes.
- Les classes que implementen una *interface* estan obligades a sobreescriure tots els mètodes definits en la *interface*. Si la definició d'algun dels mètodes a sobreescriure coincideix amb la definició d'algun mètode heretat, aquest desapareix de la classe.
- Una classe pot implementar múltiples *interfaces*, a diferència de la derivació, que només es permet d'una única classe base.
- Una *interface* introdueix un nou tipus de dada, per la qual mai no hi haurà cap instància, però sí objectes usuaris de la *interface* -objectes de les classes que implementen la *interface*. Totes les classes que implementen una *interface* són compatibles amb el tipus introduït per la *interface*.
- Una *interface* no proporciona cap funcionalitat a un objecte (ja que la classe que implementa la *interface* és la que ha de definir la funcionalitat de tots els mètodes), però en canvi proporciona la possibilitat de formar part de la funcionalitat d'altres objectes (passant-la per paràmetre en mètodes d'altres classes).
- L'existència de les *interfaces* possibilita l'existència d'una jerarquia de tipus (que no s'ha de confondre amb la jerarquia de classes) que permet l'herència múltiple.
- Una *interface* no es pot instanciar, però sí s'hi pot fer referència.

Així, si I és una ***interface*** i C és una classe que implementa la *interface*, es poden declarar referències al tipus I que apuntin objectes de C:

```
1 I obj = new C (<paràmetres>);
```

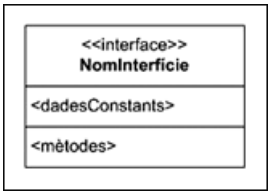
- Les *interfaces* poden heretar d'altres *interfaces* i, a diferència de la derivació de classes, poden heretar de més d'una *interface*.

Així, si dissenyem la *interface* `Treballador`, podem fer que les classes ja existents (`Professor`, `Informatic`, `Hostaleria`...) la implementin i, per tant, els objectes d'aquestes classes, a més de ser objectes de les superclasses respectives, passen a ser considerats objectes usuaris del tipus `Treballador`. Amb aquesta actuació

ens veurem obligats a implementar el mètode `calculImportJubilacio()` a totes les classes que implementin la *interface*.

Algú no experimentat en la gestió d'*interfaces* pot pensar: per què tan enrenou amb les *interfaces* si haguéssim pogut dissenyar directament un mètode anomenat `calculImportJubilacio()` a les classes afectades sense necessitat de definir cap *interface*?

La resposta rau en el fet que la declaració de la *interface* porta implícita la declaració del tipus `Treballador` i, per tant, podrem utilitzar els objectes de totes les classes que implementin la *interface* en qualsevol mètode de qualsevol classe que tingui algun argument referència al tipus Treballador com, per exemple, en un hipotètic mètode d'una hipotètica classe anomenada `Hisenda`:



Notació d'una interface.

```
public void enviarEsborranyIRPF(Treballador t) {...}
```

Pel fet d'existir la *interface* `Treballador`, tots els objectes de les classes que la implementen (`Professor`, `Informatica`, `Hostaleria`...) es poden passar com a paràmetre en les crides al mètode `enviarEsborranyIRPF (Treballador t)`.

La sintaxis per declarar una *interface* és:

```
[public] interface <NomInterface> [extends <Nominterface1>, <Nominterface2>...]
     {
   <CosInterface>
}
```

Les *interfaces* també es poden assignar a un paquet. La inexistència del modificador d'accés public fa que la *interface* sigui accessible a nivell del paquet.

Per als noms de les *interfaces*, s'aconsella seguir el mateix criteri que per als noms de les classes. En la documentació de Java, les *interfaces* s'identifiquen ràpidament entre les classes perquè estan en cursiva.

El cos de la *interface* és la llista de mètodes i/o constants que conté la *interface*. Per a les constants no cal indicar que són static i final i per als mètodes no cal indicar que són public. Aquestes característiques s'assignen implícitament.

La sintaxi per declarar una classe que implementi una o més *interfaces* és:

```
[final] [public] class <NomClasse> [extends <NomClasseBase>]
                implements <NomInterfície1>, <NomInterfície2>... {
   <CosDeLaClasse>
}
```

Els mètodes de les *interfaces* a implementar en la classe han de ser obligatòriament d'accés public.

Per acabar, cal comentar que, com que per definició totes les dades membre que es defineixen en una *interface* són `static` i `final`, i atès que les *interface* no es poden instanciar, també resulten una bona eina per implantar grups de constants. Així, per exemple:

```
public interface DiesSetmana
```

```
{
int DILLUNS = 1, DIMARTS=2, DIMECRES=3, DIJOUS=4;
int DIVENDRES=5, DISSABTE=6, DIUMENGE=7;
String [] NOMS_DIES = {"", "Dilluns", "Dimarts", "Dimecres",
   "Dijous", "Divendres", "Dissabte", "Diumenge"};
}
```

Aquesta definició ens permet utilitzar les constants declarades en qualsevol classe que implementi la *interface*, de manera tan simple com:

```
System.out.println (DiesSetmana.NOMS_DIES[DILLUNS]);
```

1.4.1 Exemple de disseny d'interface i implementació en una classe

Es presenten un parell d'*interfaces* que incorporen dades (de tipus primitiu i de referència a classe) i mètodes i una classe que les implementa. En la declaració de la classe es veu que només implementa la *interface* B, però com que aquesta *interface* deriva de la *interface* A resulta que la classe està implementant les dues *interfaces* .

```
//Fitxer: ExempleInterficie.java

import java.util.Date;
public interface A {
   Date DARRERA_CREACIO = new Date(0,0,1);   // 1-1-1900

   void metodeA ();
}

public interface B extends A
{
   int VALOR_B = 20;

   void metodeB ();
}

public class ExempleInterface implements B
{
   private long b;
   private Date dataCreacio = new Date();

   public ExempleInterface (int factor)
   {
      b = VALOR_B * factor;
      DARRERA_CREACIO.setTime(dataCreacio.getTime());
   }

   public void metodeA ()
   {
      System.out.println ("En metodeA, DARRERA_CREACIO = " + DARRERA_CREACIO);
   }

   public void metodeB ()
   {
      System.out.println ("En metodeB, b = " + b);
   }

   public static void main (String args[])
   {
      System.out.println("Inicialment, DARRERA_CREACIO = " + DARRERA_CREACIO);
```

```
41        ExempleInterface obj = new ExempleInterface(5);
42        obj.metodeA();
43        obj.metodeB();
44        A pa = obj;
45        B pb = obj;
46     }
47  }
```

L'exemple serveix per il·lustrar uns quants punts:

- Comprovem que les dades membre de les *interfaces* són `static`, ja que en el mètode `main()` fem referència a la dada membre `DARRERA_CREACIO` sense indicar cap objecte de la classe.
- Si haguéssim intentat modificar les dades `VALOR_B` o `DARRERA_CREACIO` no hauríem pogut perquè és final, però en canvi sí podem modificar el contingut de l'objecte Date apuntat per `DARRERA_CREACIO`, que correspon al moment temporal de la darrera creació d'un objecte i a cada nova creació se n'actualitza el contingut.
- En les dues darreres instruccions del mètode `main()` veiem que podem declarar variables "pa" i "pb" de les *interfaces* i utilitzar-les per fer referència a objectes de la classe `ExempleInterface`.

Classes fonamentals

Joan Arnedo Moreno

Índex

Introducció

Fins ara heu après com utilitzar aquells elements que us ofereix la sintaxi del llenguatge Java per poder definir les vostres pròpies estructures de dades, les classes, i poder assignar com han de dur a terme un seguit de tasques mitjançant atributs i mètodes. Ara bé, a l'hora de desenvolupar aplicacions, sovint es poden identificar tasques genèriques que caldrà dur a terme a gairebé totes les aplicacions, però que impliquen la creació d'alguna classe, o de diverses. En un cas com aquest, el que no tindria sentit és que cada desenvolupador del món s'hagués d'inventar les seves pròpies classes pel seu compte i anar-les reutilitzant dins els seus diferents programes. El més còmode i lògic seria que els propis creadors del llenguatge Java identifiquessin aquestes tasques tan comunes i oferissin un conjunt de classes que permetin dur-les a terme. En lloc de partir sempre de zero, les classes que genereu haurien de ser només per fer les tasques concretes que només apliquen al cas del vostre programa. Doncs resulta que, afortunadament, al llenguatge Java això es compleix. Junt amb les eines per generar i executar programes s'incorpora una llibreria de classes programades pels desenvolupadors de Java llestes per al vostre ús, accessibles mitjançant l'anomenada API del Java. Aquesta unitat se centra en aquelles classes considerades fonamentals, i que solucionen les tasques més simples i comunes.

L'API de Java està formada per una gran jerarquia de classes que cobreixen una gran quantitat d'aspectes relacionats amb el desenvolupament de programari. Està organitzada en paquets (*package*) ordenats per temes. Els entorns de desenvolupament J2SE, J2EE i J2ME permeten la utilització de tots els paquets que se subministren en el desenvolupament de programes Java, i l'entorn d'execució JRE permet l'execució de programes que utilitzen qualsevol de les classes de l'API. La documentació que subministra l'entorn de desenvolupament corresponent conté un manual de referència complet, ordenat per paquets i classes, de tot el contingut de l'API; consultar-lo resulta imprescindible en qualsevol desenvolupament.

Cal dir que l'API de Java és immensa. A títol d'exemple, la versió a 1.6 conté, segons la seva documentació, 203 paquets que inclouen 3.973 elements entre classes, interfícies... Per tant, el coneixement en profunditat de l'API no és una tasca trivial, i és imprescindible en el desenvolupament d'aplicacions. Cal fer una aproximació de manera progressiva, adquirint un coneixement general de les capacitats globals de l'API, per especialitzar-se cada vegada més en funció de les necessitats. Abans d'intentar resoldre un problema de programació, és convenient valorar si hi ha algun paquet de l'API que doni directament la solució del problema o, si no, que ajudi a obtenir la solució definitiva.

Dins l'apartat "Classes Fonamentals" d'aquesta unitat estudiareu una petita part d'aquesta API, centrant-vos en el que es podrien considerar els dos aspectes imprescindibles de cara a dur a terme aplicacions en Java.

Per una banda, veureu com dur a terme la gestió d'errors mitjançant un conjunt de classes especials anomenades "Excepcions". Aquestes classes permeten dur a terme el codi dels vostres programes de manera que s'eviti haver de comprovar, després de cada crida a un mètode, si aquest ha dut a terme la seva tasca correctament o ha succeït un error. Això fa el codi molt més llegible i fàcil de gestionar.

Per altra banda, es presentaran les classes que representen les anomenades "Col·leccions" del Java. Aquestes permeten emmagatzemar conjunts arbitraris d'objectes, aportant una alternativa molt més còmoda i, en alguns aspectes, més eficient que la utilització d'*arrays*. Cadascuna d'aquestes classes es basa en una estratègia d'emmagatzematge diferent que la fa més adient o menys segons diferents circumstàncies.

També es presenta breument el conjunt de *packages* que permeten facilitar el tractament de dades dins les vostres aplicacions mitjançant el seu processament en format XML, un estàndard en l'actualitat per a la representació d'informació, i l'ús de mecanismes més potents de cerca dins cadenes de text mitjançant expressions regulars.

Finalment, s'explica com treballar amb *arrays* de múltiples dimensions en Java. Tot i no ser estrictament un conjunt de classes englobades dins els *packages* fonamentals del Java, sí que es tracta de tipus de dades de certa complexitat que mereixen ser estudiats amb una mica de detall.

Resultats d'aprenentatge

En finalitzar aquesta unitat l'alumne/a:

1. Escriu programes que manipulin informació seleccionant i utilitzant els tipus avançats de dades facilitats pel llenguatge

- Escriu programes que utilitzin taules (arrays).
- Reconeix les llibreries de classes relacionades amb la representació i manipulació de col·leccions.
- Utilitza les classes bàsiques (vectors, llistes, piles, cues, taules de Hash) per emmagatzemar i processar informació.
- Utilitza iteradors per recórrer els elements de les col·leccions.
- Reconeix les característiques i avantatges de cada una de les col·leccions de dades disponibles.
- Crea classes i mètodes genèrics.
- Utilitza expressions regulars en la recerca de patrons en cadenes de text.
- Identifica les classes relacionades amb el tractament de documents XML.
- Dissenya programes que realitzen manipulacions sobre documents XML.

2. Gestiona els errors que poden aparèixer en els programes, utilitzant el control d'excepcions facilitat pel llenguatge.

- Reconeix els mecanismes de control d'excepcions facilitats pel llenguatge.
- Implementa la gestió d'excepcions en la utilització de classes facilitades pel llenguatge.
- Implementa el llançament d'excepcions en les classes que desenvolupa.
- Reconeix la incidència de l'herència en la gestió d'excepcions.

1. Classes fonamentals

El llenguatge Java proporciona una gran quantitat de paquets de classes, la seva API, que són referència bàsica per a qualsevol programador en Java i, per tant, ens correspon endinsar-nos en el seu coneixement. Val a dir, però, que això només és possible quan ja es dominen els mecanismes de la programació orientada a objectes (encapsulament, herència i polimorfisme).

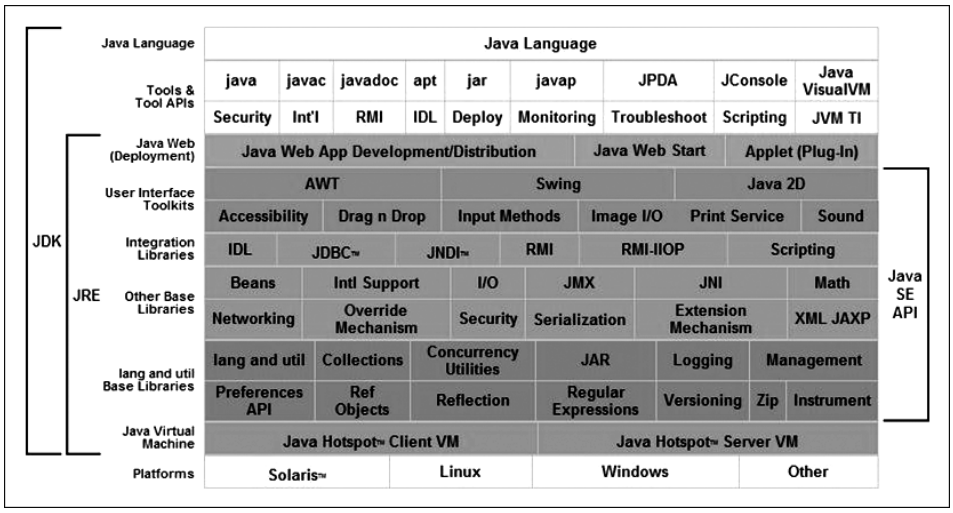
La nomenclatura dels paquets és uniforme i ajuda a categoritzar les classes. El primer qualificador és `java` o `javax` (per a les classes dedicades a la interfície gràfica) o `org` (per a les classes dedicades a donar suport a paquets de determinades organitzacions). Pels paquets `java.*` i `javax.*`, el segon qualificador dóna una idea de la matèria que cobreix el paquet, com io (entrada/sortida) i math (funcions matemàtiques). Hi ha temes que contenen subpaquets, amb un tercer qualificador més específic (per exemple, `javax.sound.midi`, amb un nom prou significatiu).

I, com ens podem plantejar el coneixement de l'extensa API de Java? Intentar fer un recorregut per tots els paquets i classes seria, gairebé, un suïcidi. El consell és plantejar l'aprenentatge dels paquets agrupats per funcionalitats, començant per les més usuals en qualsevol tipus de desenvolupament: representació i manipulació de col·leccions d'objectes, informació referent a les classes i llur estructura, control d'errors, gestió d'entrades i sortides, desenvolupament d'interfícies gràfiques...

En alguns casos, la documentació del mateix llenguatge Java agrupa, sota un nom determinat, el conjunt de paquets relacionats amb una funcionalitat determinada. Les agrupacions d'aquest tipus són simplement documentals i no es tradueixen en cap concepte dins el llenguatge Java. I respecte a la nomenclatura d'aquestes agrupacions, la plataforma Java no és uniforme i tant utilitza el mot API (per referir-se a un subconjunt de l'API de Java) com el mot *framework* com no utilitza cap mot especial. Com a exemples d'aquesta no uniformitat, podem visitar la pàgina principal de la documentació HTML de la plataforma Java SE 6, en la qual trobarem una imatge com la de la figura 1.1 que mostra tots els components de la plataforma Java.

Un escombratge del ratolí per damunt dels diversos components ens mostra el nom oficial que Java dóna als components. Entre d'altres, hi podem veure *Collections Framework (Collections)*, *Reflection API (Reflection)*, *Input/Output API (I/O)*, *Abstract Window Toolkit (AWT)*, *Graphical User Interface Components (Swing)*, *Java Database Connectivity API (JDBC)*, *Java archive file format (JAR)*, *Input Method Framework (Input Methods)*, etc.

FIGURA 1.1. Components de la plataforma Java Se



Com es veu, hi ha funcionalitats batejades amb el sufix API, altres amb el sufix *framework* i altres sense cap sufix especial. Queda clar, doncs, que Java utilitza el mot *framework* per batejar alguns components de la plataforma. Aquest concepte no és propietat de la plataforma Java sinó que s'utilitza força en el món de la programació orientada a objectes.

> Un ***framework*** (entorn o marc de treball) és un concepte utilitzat en la programació orientada a objectes per designar un conjunt de classes que defineixen un disseny abstracte i una estructura per solucionar un conjunt de problemes relacionats.

De tot el mapa presentat a la figura 1.1, en aquest apartat ens centrarem en només dos parts molt concretes. Per una banda, una part molt important de "lang i util", la centrada en el control i gestió d'errors: les excepcions. I d'altra banda, la "*Collections Framework*".

1.1 Excepcions

En el desenvolupament de programes en qualsevol llenguatge de programació, el programador ha de disposar de mecanismes (proporcionats pel llenguatge) per detectar els errors que es puguin produir en temps d'execució. Així, per exemple, si un programa ha d'accedir a un fitxer que es troba en un dispositiu determinat, el programador ha d'haver previst que el fitxer pot no ser accessible i, per tant, ha d'haver establert les alternatives d'execució enlloc de provocar una aturada brusca del programa.

S'anomena **gestió d'excepcions** el conjunt de mecanismes que un llenguatge de programació proporciona per detectar i gestionar els errors que es puguin produir en temps d'execució.

La gestió d'excepcions no preveu mai els errors de sintaxi que es detecten en temps de compilació i, molt poques vegades, els anomenats *errors de programació* que no s'haurien de produir mai: intent d'accés a una posició inexistent en una taula, divisió per zero, intent d'accés per una referència nul·la...

Hi ha dues maneres de gestionar les excepcions:

- De la manera tradicional, dissenyant els mètodes de manera que retornin un codi d'error que es revisa després de la crida a la funció o mètode amb l'ajut d'instruccions condicionals per tal de prendre la decisió adequada. Aquesta tècnica no dóna bons resultats quan l'error és fatal (ha de provocar la finalització del programa) i es pot produir en diferents nivells de crides internes, ja que la funció o el mètode dissenyats per nosaltres es pot cridar dins d'altres funcions o mètodes, fet que fa impossible saber la cascada de crides fins al punt en el qual s'ha produït l'error.
- Utilitzant construccions especials per a la gestió d'errors proporcionades pel llenguatge de programació, com és habitual en els llenguatges moderns com C++, Visual Basic i Java, que acostumen a proporcionar mecanismes de propagació de l'error cap a les funcions o mètodes que han cridat la funció o mètode en què s'ha produït l'error, de manera que és possible conèixer la cascada de crides fins el punt en el què s'ha produït l'error.

El model de gestió d'excepcions que proporciona el llenguatge Java és simple: en produir-se un error, la màquina virtual llança (*throw*) un avís que el programador hauria de poder capturar (*catch*) per resoldre la situació problemàtica.

1.1.1 Tipus d'excepcions

El llenguatge Java distingeix entre error i excepció. Els errors corresponen a situacions irrecuperables, que no tenen solució i que no depenen del programador, el qual no s'ha de preocupar de capturar. No s'haurien de produir mai, però quan tenen lloc provoquen la finalització brusca del programa. Tenim exemples d'errors quan la màquina virtual es queda sense recursos per continuar amb l'execució del programa, quan alguna cosa va malament en la càrrega d'un servei d'un proveïdor, quan deixa de respondre un canal d'entrada/sortida...

En canvi, les excepcions corresponen a situacions excepcionals que els programes es poden trobar en temps d'execució, i s'hi poden incloure, fins i tot, els errors de programació. El programador pot preveure cada tipus d'excepció i escriure el codi adequat per a la seva gestió.

El llenguatge Java engloba tots els possibles errors en la classe `Error` i totes les possibles excepcions en la classe `Exception`. És a dir, cada possible situació

problemàtica té associada una classe (derivada d'`Errror` o d'`Exception`) de manera que, en el moment en què es produeix la situació problemàtica, es crea un objecte de la subclasse corresponent que conté la informació del context en què s'ha produït el problema.

Les classes `Error` i `Exception` deriven, a la vegada, de la classe `Throwable`, la qual proporciona mecanismes comuns per a la gestió de qualsevol tipus d'error i excepció, entre els quals convé conèixer:

- L'existència de quatre constructors per a qualsevol classe derivada, similars al següent:

```
Throwable()
   /* Construeix un objecte Throwable amb missatge null */
Throwable(String message)
   /* Construeix un objecte Throwable amb el missatge indicat */
Throwable(String message, Throwable cause)
   /* Construeix un objecte Throwable amb el missatge indicat i amb la causa que ha
      provocat la situació */
Throwable(Throwable cause)
   /* Construeix un objecte Throwable amb la causa que l'ha provocat i com a missatge,
      el resultat de: cause==null ? null : cause.toString() */
```

Fixem-nos que hi ha dos constructors que incorporen la possibilitat de crear un objecte `Throwable` indicant un altre objecte `Throwable` com a causant (cause) del nou objecte, fet que permet encadenar els errors i/o excepcions.

- L'existència de mètodes per conèixer el context en el qual s'ha produït la situació problemàtica i, per tant, poder actuar en conseqüència, com, per exemple, els següents:

```
Throwable getCause();  /* Retorna la causa o null */
String getMessage();   /* Retorna el missatge o null */
void printStackTrace();   /* Visualitza pel canal d'errors, el context en el que s'ha
                             produït l'error i la cascada de crides des del mètode main()
                             que han portat al punt en el que s'ha produït l'error */
String toString();   /* Retorna una curta descripció de l'objecte */
```

Per tal de desenvolupar aplicacions Java amb una bona gestió d'excepcions, en primer lloc ens hem de centrar en el coneixement de la jerarquia de classes que neix a partir de la classe `Exception` i, en segon lloc, en els mecanismes de gestió d'excepcions que proporciona Java.

En la classe `Exception` cal distingir dos grans subtipus d'excepcions:

1. Les excepcions implícites que la mateixa màquina virtual s'encarrega de comprovar durant l'execució d'un programa i que el programador no té l'obligació de capturar i gestionar. Estan agrupades en la classe `RuntimeException` i normalment estan relacionades amb errors de programació, que podríem categoritzar en els següents:

- **Errors que normalment no es revisen en el codi d'un programa** com, per exemple, rebre una referència `null` en un mètode, quan el dissenyador del mètode ha suposat que qui la cridi ja haurà passat una referència no nul·la.
- **Errors que el programador hauria d'haver revisat en escriure el codi** com, per exemple, sobrepassar la grandària assignada a una taula.

  En realitat seria possible comprovar aquests dos tipus d'errors, però el codi es complicaria excessivament. Hem de pensar en el *savoir faire* del programador, no?

2. Les excepcions explícites (totes les de la classe `Exception` que no pertanyen a la subclasse `RuntimeException`) que el programador està obligat a tenir en compte allà on es puguin produir.

En referència als mecanismes de gestió d'excepcions que proporciona Java, ens cal saber com es gestionen les excepcions, com es generen (*llancen*, en terminologia Java) excepcions, com es creen noves excepcions per donar suport a una gestió d'excepcions per a les classes que dissenyem i quins efectes té l'herència en la gestió d'excepcions.

1.1.2 Captura i tractament

El llenguatge Java proporciona el mecanisme "try - catch" per capturar una excepció i definir l'actuació que correspongui. Aquest consisteix a col·locar el codi susceptible de generar (llançar) l'excepció que es vol capturar dins un bloc de codi precedit per la paraula reservada `try` i seguit de tants blocs de codi `catch` com excepcions diferents es volen capturar, segons la sintaxi següent:

```
try {
<bloc_de_codi_susceptible_de_llançar_excepció>
} catch (nomClasseExcepció1 e1) {
<bloc_de_codi_a_executar_si_en_el_bloc_try_s'ha_produït_una_nomClasseExcepció1>
} catch (nomClasseExcepció2 e2) {
<bloc_de_codi_a_executar_si_en_el_bloc_try_s'ha_produït_una_nomClasseExcepció2>
} ...
} finally {
<bloc_de_codi_a_executar_en_qualsevol_cas–s'hagi_produït_o_no_una_excepció–>
}
```

El bloc `try` pot anar seguit d'un o més blocs `catch` cadascun dels quals va precedit d'una declaració (`nomClasseExcepció e`) que defineix l'excepció (o conjunt d'excepcions corresponents a totes les classes derivades de `nomClasseExcepció`) a la qual el bloc dóna resposta.

En cas de produir-se una excepció en el codi del bloc `try`, la màquina virtual avorta l'execució del codi del bloc `try` (no s'acabarà en cap cas) i comença a avaluar els diversos blocs `catch`, en l'ordre en què estiguin situats, fins a trobar el primer bloc que en la seva classe d'excepcions inclogui l'excepció produïda en el bloc `try`. Per tant, en cas que entre les excepcions a capturar n'hi hagi d'emparentades

per la relació de derivació (unes siguin subclasses d'altres), cal situar en primer lloc els blocs `catch` per gestionar les excepcions corresponents a les classes que ocupen el lloc més baix en la jerarquia de classes.

En cas d'existir un bloc `catch` que correspongui a l'excepció produïda en el bloc `try`, la màquina virtual executa el codi del bloc `catch` (que podria ser buit!) i, en finalitzar, executa el codi del bloc `finally`, en cas d'existir, per posteriorment prosseguir l'execució del programa.

En cas de no existir cap bloc `catch` que correspongui a l'excepció produïda, la màquina virtual executa el codi del bloc `finally`, en cas d'existir, i posteriorment avorta el mètode en què s'ha produït l'excepció i propaga l'excepció al mètode immediatament superior (en el punt en què s'havia produït la crida al mètode actual) perquè sigui allí on es capturi l'excepció. Si l'excepció tampoc no és capturada, s'avorta el mètode i es propaga l'excepció al mètode immediatament superior i així successivament fins que l'excepció és gestionada. Si una excepció no es gestiona i arriba al mètode `main()` i ni tant sols aquest la gestiona, es produeix una finalització anormal de l'execució del programa.

El bloc opcional `finally` s'executa sempre, s'hagi produït una excepció o no i, si s'ha produït, hagi estat capturada o no. Aquest bloc fins i tot s'executa si dins els blocs "try - catch" hi ha alguna sentència `continue`, `break` o `return`. L'única situació en què el bloc `finally` no s'executa és quan es crida el mètode `System.exit()` que finalitza l'execució del programa.

Com a exemple de conveniència d'utilització del bloc `finally` podem pensar en un bloc `try` dins del qual s'obre un fitxer per a lectura i escriptura de dades i, en finalitzar, es vol tancar el fitxer obert. El fitxer obert s'ha de tancar tant si es produeix una excepció com si no es produeix, ja que deixar un fitxer obert pot provocar problemes. Per assegurar el tancament del fitxer, caldria situar les sentències corresponents dins el bloc `finally`.

En la majoria de casos, un bloc `try` anirà seguit d'un o més blocs `catch`, però també és possible que no hi hagi cap bloc catch però sí un bloc finally per assegurar l'execució de certes accions. El codi següent il·lustra aquesta situació:

```
try
{
  obrirAixeta();
  regarGespa();
} finally {
  tancarAixeta();
}
```

En el moment en què es produeix la situació excepcional, es crea un objecte de la classe corresponent a l'excepció que conté la informació del context en què s'ha produït el problema. Aquest objecte és apuntat per la referència e indicada en la declaració de l'excepció que gestiona el bloc `catch (nomClasseExcepció e)` corresponent i conté informació que pot ser d'importància per al programador. Recordem que els mètodes se güents heretats de la classe `Throwable` ens permeten obtenir informació del context en què s'ha produït l'excepció a partir de l'objecte generat:

```
String getMessage();   /* Retorna el missatge o null */
void printStackTrace();   /* Visualitza pel canal d'errors, el context en el
    que s'ha produït l'error i la cascada de crides des del mètode main() que
    han portat al punt en el que s'ha produït l'error */
String toString();   /* Retorna una curta descripció de l'objecte */
```

Exemple de llançament d'excepció no capturada

L'exemple següent mostra un programa que conté un error de programació, ja que s'intenta efectuar un recorregut per una taula sortint dels límits permesos. Aquesta és una excepció catalogada sota `RuntimeException` i, si el programador hagués estat atent, no s'hauria d'haver produït mai.

```
//Fitxer Excepcio01.java

public class Excepcio01 {
   public static void main(String args[]) {
      String t[]={"Hola","Adéu","Fins demà"};
      for (int i=0; i<=t.length; i++)
         System.out.println("Posició " + i + " : " + t[i]);
      System.out.println("El programa s'ha acabat.");
   }
}
```

Si executem el programa, obtenim:

```
Posició 0 : Hola
Posició 1 : Adéu
Posició 2 : Fins demà
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: 3
    at Excepcio01.main(Excepcio01.java:12)
```

Veiem que la màquina virtual va executant el programa i efectuant el recorregut per les diverses posicions de la taula fins que intenta accedir a la posició indexada amb el valor 3, inexistent en la taula. En aquest moment la màquina virtual llança l'excepció `ArrayIndexOutOfBoundsException`, i com que el codi en què s'ha produït l'excepció no es troba dins cap bloc "try - catch", i ja estem en el main(), s'avorta el programa. Fixeu-vos que el darrer missatge "El programa s'ha acabat" no apareix en la consola perquè no s'arriba a executar la instrucció que el visualitza.

La màquina virtual informa, pel canal de sortida d'errors (que per defecte és la consola), de l'error produït i el nom del mètode i número de línia en què s'ha produït l'excepció.

Exemple de llançament d'excepció capturada

L'exemple següent mostra un programa que conté un error de programació que és capturat. Aquests errors no s'acostumen a capturar, però ens serveix com a exemple senzill de captura d'una excepció. L'exemple anterior ens ha mostrat que el compilador no ens obliga a capturar aquest tipus d'excepció.

```java
//Fitxer Excepcio02.java

public class Excepcio02 {
   public static void main(String args[]) {
      String t[]={"Hola","Adéu","Fins demà"};
      try {
         System.out.println("Abans d'executar el for");
         for (int i=0; i<=t.length; i++)
            System.out.println("Posició " + i + " : " + t[i]);
         System.out.println("Després d'executar el for");
      } catch (ArrayIndexOutOfBoundsException e) {
         System.out.println("El programador estava a la lluna... S'ha sortit de
              límits!!!");
      }
      System.out.println("Final del programa");
   }
}
```

L'execució del programa mostra:

```
Abans d'executar el for
Posició 0 : Hola
Posició 1 : Adéu
Posició 2 : Fins demà
El programador estava a la lluna... S'ha sortir de límits!!!
Final del programa
```

Veiem que la màquina virtual va executant el programa i efectuant el recorregut per les diverses posicions de la taula fins que intenta accedir a la posició indexada amb el valor 3, inexistent en la taula. En aquest moment la màquina virtual llança l'excepció `ArrayIndexOutOfBoundsException` i, com que el codi en què s'ha produït l'excepció es troba dins un bloc "try - catch" que captura l'excepció produïda, la màquina virtual executa el codi del bloc catch i després continua el programa. Fixeu-vos que el darrer missatge "Després d'executar el for" del bloc try no s'executa, ja que l'excepció provoca l'avortament de l'execució del codi del bloc en el moment en què es produeix.

La màquina virtual no diu, per enlloc, que s'ha produït una excepció. El programador ho sap per què el flux d'execució ha entrat en el bloc `catch` que gestiona l'excepció.

Si ens informem, en la documentació de Java sobre l'excepció `ArrayIndexOutOfBoundsException`, veurem que és subclasse de la classe `IndexOutOfBoundsException`, que al seu torn és subclasse de `RuntimeException`, i aquesta, de la classe `Exception`. Per tant, si en el bloc catch haguéssim declarat qualsevol d'aquestes classes, l'excepció també hauria estat capturada. En general només utilitzem superclasses de l'excepció a capturar si no hem de proveir diferents actuacions per a diferents excepcions.

Exemple de llançament d'excepció amb intent erroni de captura i mètode finally

L'exemple següent mostra un programa que conté un error de programació amb un intent de captura que falla perquè l'excepció que s'indica en el bloc `catch` no és

adequada per a l'excepció que es produeix. Així mateix incorpora un bloc finally per demostrar que el codi introduït en aquest bloc s'executa en qualsevol cas.

```
//Fitxer Excepcio03.java

public class Excepcio03 {
   public static void main(String args[]) {
      String t[]={"Hola","Adéu","Fins demà"};
      try {
         System.out.println("Abans d'executar el for");
         for (int i=0; i<=t.length; i++)
            System.out.println("Posició " + i + " : " + t[i]);
         System.out.println("Després d'executar el for");
      } catch (StringIndexOutOfBoundsException e) {
         System.out.println("El programador estava a la lluna... S'ha sortir de
              límits!!!");
      } finally {
         System.out.println("Aquest codi s'executa, peti qui peti!!!");
      }
      System.out.println("Final del programa");
   }
}
```

L'execució d'aquest programa provoca la sortida:

```
Abans d'executar el for
Posició 0 : Hola
Posició 1 : Adéu
Posició 2 : Fins demà
Aquest codi s'executa, peti qui peti!!!
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: 3
    at Excepcio03.main(Excepcio03.java:16)
```

Veiem que l'excepció que es produeix `ArrayIndexOutOfBoundsException` no és subclasse de la classe `StringIndexOutOfBoundsException` i, per tant, el bloc catch existent no captura l'excepció que es produeix. Així mateix podem veure que el bloc finally s'executa abans de la finalització brusca del programa.

Exemple de llançament d'excepció en mètode interior sense captura en cap mètode

El programa següent mostra un exemple de propagació de l'excepció cap al mètode superior sense cap tipus de tractament, de manera que la propagació arriba al mètode main().

```
//Fitxer Excepcio04.java

public class Excepcio04 {
   public static void met02() {
      String t[]={"Hola","Adéu","Fins demà"};
      for (int i=0; i<=t.length; i++)
         System.out.println("Posició " + i + " : " + t[i]);
      System.out.println("El mètode met02 s'ha acabat.");
   }

   public static void met01() {
      System.out.println("Entrem en el mètode met01 i anem a executar met02");
      met02();
      System.out.println("Tornem a estar en met02 després de finalitzar met02")
          ;
   }

```

```
17    public static void main(String args[]) {
18       System.out.println("Iniciem el programa i anem a executar met01");
19       met01();
20       System.out.println("Tornem a estar en el main després de finalitzar met01
             ");
21    }
22 }
```

L'execució d'aquest programa dóna el resultat:

```
1  Iniciem el programa i anem a executar met01
2  Entrem en el mètode met01 i anem a executar met02
3  Posició 0 : Hola
4  Posició 1 : Adéu
5  Posició 2 : Fins demà
6  Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException:
7  3
8      at Excepcio04.met02(Excepcio04.java:15)
9      at Excepcio04.met01(Excepcio04.java:22)
10     at Excepcio04.main(Excepcio04.java:29)
```

L'execució del programa demostra que, en el moment en què es produeix l'error dins el mètode `met02`, es finalitza l'execució d'aquest mètode i l'excepció es passa a la instrucció del mètode `met01` en què s'havia cridat el mètode met02. Com que l'excepció tampoc no es captura dins `met01`, aquest mètode també avorta la seva execució i l'excepció es passa a la instrucció del mètode main en què s'havia cridat el mètode `met01`. En no haver-hi, tampoc, captura, el programa avorta.

Fixem-nos en el missatge que proporciona la màquina virtual: informa de l'excepció que s'ha produït, i la cascada de crides, en ordre invers, des del programa principal fins a la instrucció en què s'ha produït l'excepció.

Exemple de llançament d'excepció capturada en mètode superior

El programa següent mostra un exemple de propagació de l'excepció cap al mètode superior en què, abans d'arribar al `main()`, es troba un mètode que captura l'excepció.

```
1  //Fitxer Excepcio05.java
2
3  public class Excepcio05 {
4     public static void met03() {
5        String t[]={"Hola","Adéu","Fins demà"};
6        for (int i=0; i<=t.length; i++)
7           System.out.println("Posició " + i + " : " + t[i]);
8        System.out.println("El mètode met03 s'ha acabat.");
9     }
10
11    public static void met02() {
12       System.out.println("Entrem en el mètode met02 i anem a executar met03");
13       met03();
14       System.out.println("Tornem a estar en met02 després de finalitzar met03")
             ;
15    }
16
17    public static void met01() {
18       try {
19          System.out.println("Entrem en el mètode met01 i anem a executar met02"
                );
20          met02();
```

```
21          System.out.println("Tornem a estar en met01 després de finalitzar
                met02");
22       } catch (ArrayIndexOutOfBoundsException e) {
23          System.out.println("El programador estava a la lluna... S'ha sortir de
                límits!!!");
24       }
25    }
26
27    public static void main(String args[]) {
28       System.out.println("Iniciem el programa i anem a executar met01");
29       met01();
30       System.out.println("Tornem a estar en el main després de finalitzar met01
              ");
31    }
32 }
```

L'execució d'aquest programa dóna la sortida:

```
1  Iniciem el programa i anem a executar met01
2  Entrem en el mètode met01 i anem a executar met02
3  Entrem en el mètode met02 i anem a executar met03
4  Posició 0 : Hola
5  Posició 1 : Adéu
6  Posició 2 : Fins demà
7  El programador estava a la lluna... S'ha sortir de límits!!!
8  Tornem a estar en el main després de finalitzar met01
```

L'execució d'aquest programa ens demostra que la propagació cap als mètodes superiors finalitza quan trobem un mètode que captura l'excepció. En el cas que ens ocupa, l'excepció es produeix en el mètode `met03` que no captura l'excepció, i aquesta es propaga fins al mètode `met02` que, en no capturar-la, la propaga cap al mètode met01. En aquest mètode sí que es captura l'excepció i el programa continua la seva execució normal a partir de la instrucció següent a la captura.

Exemple d'obtenció d'informació d'una excepció

Modifiquem el mètode `met01()` de manera que en el bloc `catch` que captura l'excepció utilitzem els mètodes que ens permeten obtenir informació sobre l'excepció.

```
1  static void met01()
2  {
3     try
4     {
5        System.out.println("Entrem en el mètode met01 i anem a executar met02");
6        met02();
7        System.out.println("Tornem a estar en met01 després de finalitzar met02")
              ;
8     }
9     catch (ArrayIndexOutOfBoundsException e)
10    {
11       System.out.println("Estem dins el bloc catch que ha capturat l'excepció."
              );
12       System.out.println("Informació que dona el mètode getMessage():");
13       System.out.println(e.getMessage());
14       System.out.println("Informació que dona el mètode printtStackTrace():");
15       e.printStackTrace();
16       System.out.println("Informació que dona el mètode toString():");
17       System.out.println(e);
18    }
19 }
```

L'execució del programa ens mostra la informació que proporciona cada mètode:

```
Iniciem el programa i anem a executar met01
Entrem en el mètode met01 i anem a executar met02
Entrem en el mètode met02 i anem a executar met03
Posició 0 : Hola
Posició 1 : Adéu
Posició 2 : Fins demà
Estem dins el bloc catch que ha capturat l'excepció.
Informació que dona el mètode getMessage():           Excepcio06.java
3                               Excepcio05.java
Informació que dona el mètode printStackTrace():
java.lang.ArrayIndexOutOfBoundsException: 3
    at Excepcio06.met03(Excepcio06.java:13)
    at Excepcio06.met02(Excepcio06.java:20)
    at Excepcio06.met01(Excepcio06.java:29)
    at Excepcio06.main(Excepcio06.java:47)
Informació que dona el mètode toString():
java.lang.ArrayIndexOutOfBoundsException: 3
Tornem a estar en el main després de finalitzar met01
```

Veiem que la informació que dóna el mètode printStackTrace() és la mateixa que mostra la màquina virtual pel canal de sortida d'errors en cas d'avortar el programa perquè no s'ha capturat l'excepció.

1.1.3 Gestió: captura o delegació

El llenguatge Java obliga el programador a gestionar totes les excepcions derivades de la classe Exception exceptuant les de la classe `RuntimeException`.

Per exemple, el compilador no obliga a gestionar l'excepció `ArrayIndexOutOfBoundsException`, ja que és una excepció de la classe `RuntimeException` i és generada (llançada) directament per la màquina virtual en el control de l'execució del programa. Però la resta d'excepcions de la classe Exception(que no siguin RuntimeException) no són generades (llançades) per la màquina virtual, sinó per diversos mètodes de diverses classes, proporcionades pel llenguatge Java o dissenyades pel programador.

Així, doncs, el programador es troba amb la necessitat de saber quins mètodes de quines classes poden llançar una excepció amb obligatorietat de gestió. Sembla una tasca impossible atesa la gran quantitat de classes que proporciona el llenguatge Java més les classes dissenyades pel mateix equip de programació o per tercers. Això no és cap problema! Tenim dues maneres d'informar-nos sobre les excepcions que pot llançar un mètode:

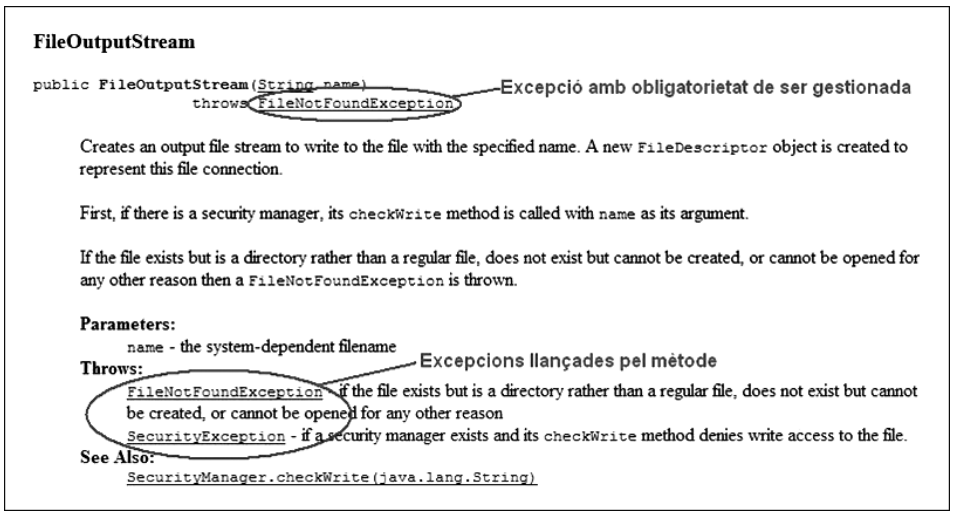
1) Fent una ullada a la documentació del mètode. La figura 1.2 mostra la informació detallada d'un dels mètodes constructors de la classe `FileOutputStream` que serveix per crear arxius per escriptura. Hi ha dues informacions relacionades amb la gestió d'excepcions a tenir en compte:

- En la part final de l'explicació hi ha l'apartat "Throws" que informa sobre les excepcions que pot llançar aquest mètode. N'hi trobem dues:

`FileNotFoundException` i `SecurityException`. La primera no deriva de la classe `RuntimeException`, però la segona sí. Això vol dir que el programador que cridi el mètode `FileOutputStream` haurà de gestionar obligatòriament l'excepció `FileNotFoundException` i podrà gestionar l'excepció SecurityException o no.

- En la part inicial de l'explicació, en què es mostra la capçalera del mètode, veiem que la zona de paràmetres va seguida de la paraula throws seguida de l'excepció `FileNotFoundException`. Aquesta línia ens diu que qualsevol crida del mètode ha de gestionar aquesta excepció.

FIGURA 1.2. Informació, en la documentació dels mètodes, relativa a la gestió d'excepcions



2) Observant els errors de compilació que es produeixen si no es gestiona una excepció que no sigui `RuntimeException`.

En efecte, el compilador de Java comprova, per totes les crides de mètodes, si totes les excepcions no `RuntimeException` que el mètode pot llançar són gestionades pel programa. Si no és així, informa d'un error similar al següent:

```
1 unreported exception nomExcepció; must be caught or declared to be thrown
```

El programador es veu obligat a gestionar les excepcions no `RuntimeError`, i per fer-ho disposa de dos mecanismes, tal com indica el compilador quan detecta una excepció no gestionada: "must be caught or declared to be thrown":

1. Gestionar l'excepció dins el mètode en què es pugui produir capturant-la amb la utilització de blocs "try - catch".

2. Delegar la gestió de l'excepció al mètode superior, fet que s'indica en la capçalera del mètode, declarant les excepcions que no es gestionaran en el mètode amb la clàusula `throws` i seguint aquesta sintaxi següent:

```
1 [modificadors] nomMètode (<arguments>) [throws exc1, exc2...]
```

Fixem-nos que la paraula reservada `throws` ha d'anar seguida de totes les excepcions (exc1, exc2...) que el mètode hauria de gestionar, però que opta per no gestionar i delega la gestió al mètode superior que el cridi.

D'aquesta manera, quan el compilador avalua un mètode que conté la clàusula `throws` no té en compte les excepcions de gestió obligatòria que es poden produir dins el mètode, no capturades, i que estiguin declarades en la clàusula `throws`. En contrapartida, el compilador obliga qualsevol mètode que cridi un mètode amb clàusula `throws` a gestionar les excepcions indicades en la clàusula, capturant-les o delegant-les.

Exemple de programa que no gestiona les excepcions de gestió obligatòria

Considerem el següent fitxer .java:

```
1  //Fitxer Excepcio07.java
2
3  import java.io.*;
4
5  public class Excepcio07 {
6     public static void main (String args[]) {
7        FileOutputStream f = new FileOutputStream ("C:\\arxiu.txt");
8        f.close();
9     }
10 }
```

La seva compilació detecta els errors següents:

```
1  Excepcio07.java:13: unreported exception java.io.FileNotFoundException; must be
       caught or
2  declared to be thrown
3     FileOutputStream f = new FileOutputStream ("C:\\arxiu.txt");
4         ^
5  Excepcio07.java:14: unreported exception java.io.IOException; must be caught or
       declared to be thrown
6     f.close();
7         ^
8  2 errors
```

Veiem que tots dos errors s'han produït perquè s'han cridat mètodes, `FileOutputStream()` i `close()`, que poden llançar excepcions de gestió obligatòria i no les hem gestionat. També veiem que, en cada cas, el compilador ens informa sobre l'excepció no gestionada.

Exemple de delegació de gestió d'excepcions

El fitxer .java següent mostra un mètode que en el seu interior crida els mètodes `FileOutputStream (String)` i `close()`. El primer obliga la gestió de l'excepció `FileNotFoundException` i el segon obliga la gestió de l'excepció `IOException`. S'ha optat per no gestionar les excepcions dins el mètode sinó delegar-les als mètodes que el cridin.

```
1  //Fitxer Excepcio08.java
2
3  import java.io.*;
```

```
4
5  class Excepcio08
6  {
7     public void metodeAmbClausulaThrows (String nomFitxer)
8                 throws FileNotFoundException, IOException
9     {
10       FileOutputStream f = new FileOutputStream (nomFitxer);
11       f.close();
12       System.out.println ("El metodeAmbClausulaThrows ha finalitzat.");
13    }
14 }
```

Fixem-nos que com que l'excepció `FileNotFoundException` és subclasse de l'excepció `IOException`, hauríem pogut indicar, a la clàusula trhows, únicament l'excepció `IOException` i el compilador hauria traduït el programa, però, d'aquesta manera, els mètodes que cridin metode `AmbClausulaThrows()` no podrien veure quina excepció s'ha produït: sigui quina sigui, només podrien capturar-la mitjançant `IOException`.

El fitxer .java següent intenta cridar `metodeAmbClausulaThrows()` sense gestionar les possibles excepcions.

```
1  //Fitxer Prova01Excepcio08.java
2  import java.io.*;
3
4  public class Prova01Excepcio08 {
5    public static void main (String args[]) {
6      Excepcio08 exc = new Excepcio08();
7      exc.metodeAmbClausulaThrows("c:\\arxiu.txt");
8      System.out.println ("Hem tornat del metodeAmbClasulaThrows");
9      System.out.println ("El programa ha finalitzat.");
10   }
11 }
```

Veiem l'informe del compilador:

```
1  Prova01Excepcio08.java:13: unreported exception java.io.FileNotFoundException;
       must be caught or
2  declared to be thrown
3  exc.metodeAmbClausulaThrows("c:\\arxiu.txt");
4  ^
5  1 error
```

Fixem-nos que el compilador només ens informa de la primera excepció indicada a la clàusula `throws` del `metodeAmbClausulaThrows()` que no és gestionada.

Millorem el codi anterior gestionant l'excepció `FileNotFoundException`:

```
1  //Fitxer Prova02Excepcio08.java
2
3  import java.io.*;
4
5  public class Prova02Excepcio08 {
6    public static void main (String args[]) {
7        Excepcio08 exc = new Excepcio08();
8        try {
9           exc.metodeAmbClausulaThrows("c:\\arxiu.txt");
10       } catch (FileNotFoundException e) {
11          System.out.println("S'ha capturat l'excepció FileNotFoundException");
12       }
13       System.out.println("El programa ha finalitzat.");
14   }
```

```
15 }
```

El programador ja ha gestionat l'excepció `FileNotFoundException` però s'ha oblidat de l'excepció `IOException` i l'informe del compilador és clar:

```
Prova02Excepcio08.java:15: unreported exception java.io.IOException; must be
    caught or declared to
be thrown
      exc.metodeAmbClausulaThrows("c:\\arxiu.txt");
        ^
1 error
```

Queda clar, doncs, que la crida del mètode `metodeAmbClausulaThrows()` obliga a la gestió de les dues excepcions. Vegem la versió de programa següent:

```
//Fitxer Prova03Excepcio08.java

import java.io.*;

public class Prova03Excepcio08 {
   public static void main (String args[]) {
      if (args.length!=1) {
         System.out.println("La crida del programa ha d'indicar un paràmetre:")
              ;
         System.out.println("   nomArxiu amb el corresponent camí de directoris
              .");
         System.exit(1);
      }
      Excepcio08 exc = new Excepcio08();
      try {
         exc.metodeAmbClausulaThrows(args[0]);
      } catch (FileNotFoundException e) {
         System.out.println("S'ha capturat l'excepció FileNotFoundException,
              amb informació:");
         System.out.println(e);
      } catch (IOException e) {
         System.out.println("S'ha capturat l'excepció IOException, amb
              informació:");
         System.out.println(e);
      }
      System.out.println("El programa ha finalitzat.");
   }
}
```

Aquesta versió ja no té cap error de compilació i podem procedir a executar-la. Abans, però, de dur a la pràctica l'execució del programa indicant un nom de fitxer (c:\arxiu.txt), comproveu que aquest fitxer no existeix, ja que el programa el substituirà per un nou fitxer buit. En l'exemple d'execució següent, una vegada creat (segona acció) hi donem accés de només lectura (tercera acció) de manera que la quarta acció falla.

```
C:\>java Prova03Excepcio08
La crida del programa ha d'indicar un paràmetre:
   nomArxiu amb el corresponent camí de directoris.

C:\>java Prova03Excepcio08 c:\arxiu.txt
El metodeAmbClausulaThrows ha finalitzat.
El programa ha finalitzat.

C:\>attrib +r c:\arxiu.txt

C:\>java  Prova03Excepcio08 c:\arxiu.txt
S'ha capturat l'excepció FileNotFoundException, amb informació:
```

```
java.io.FileNotFoundException: c:\arxiu.txt (Acceso denegado)
El programa ha finalitzat.

C:\>java Prova03Excepcio08 x:\arxiu.txt
S'ha capturat l'excepció FileNotFoundException, amb informació:
java.io.FileNotFoundException: x:\arxiu.txt (El sistema no puede
hallar la ruta especificada)
El programa ha finalitzat.
```

1.1.4 Llançament

Els mètodes de les classes que proporciona Java llencen excepcions, segons les seves necessitats. De la mateixa manera, els mètodes que desenvolupa un programador poden llançar excepcions. El mecanisme per llançar una excepció des d'un mètode és molt simple i consta de dos passos:

1. Es crea un objecte de la subclasse de la classe `Exception` que correspongui a l'excepció que es vol llançar (generar).
2. Es llança l'excepció amb la sentència `throw` seguida de l'objecte creat.

És a dir, els dos passos indicats serien:

```
nomExcepció e = new nomExcepció (...);
throw e;
```

Però també es pot fer en un sol pas:

```
throw new nomExcepció (...);
```

En el moment en què dins un mètode es llança una excepció que no és `RuntimeException`, el programador ha de decidir entre:

- **Gestionar l'excepció dins el mateix mètode**, fet que implicaria que la instrucció on es llança l'excepció hauria d'estar dins un bloc "try-catch" que capturés l'excepció. Això no és gaire usual, però de vegades el programador pot utilitzar aquest recurs per provocar un break en un bloc de codi.
- **Delegar la gestió de l'excepció a mètodes superiors**, fet que implica que la capçalera del mètode inclogui la declaració de l'excepció amb la clàusula `throws`. Aquesta és la manera més usual de treballar, i el llançament fa que el mètode finalitzi i l'excepció es propagui cap al mètode superior en la pila de crides, el qual l'haurà de capturar o delegar.

Exemple de llançament d'excepció de gestió obligatòria dins un mètode

El fitxer .java següent ens mostra un exemple de mètode que llança una excepció de gestió obligatòria.

```java
//Fitxer Excepcio09.java

public class Excepcio09 {

   /* Mètode que avalua si la taula t té n cel·les, provocant, en cas de ser
        avaluada com a fals,
      una excepció d'obligada gestió: Exception */
   public static void verificaLengthTaula (int n, String t[]) throws Exception
       {
      if (t.length!=n) throw new Exception ("La taula no té la llargada
          indicada.");
      System.out.println ("Sortida de verificaLengthTaula.");
   }

   public static void main (String args[]) {
      try {
        System.out.println("Punt 1.");
        verificaLengthTaula (4, new String[4]);
        System.out.println("Punt 2.");
        verificaLengthTaula (2, new String[4]);
        System.out.println("Punt 3.");
      } catch (Exception e) {
         e.printStackTrace();
      }
      System.out.println ("Programa finalitzat.");
   }
}
```

Fixem-nos que en llançar una excepció `Exception` estem obligats a declarar l'excepció en la capçalera del mètode (si eliminem la clàusula `throws` el fitxer no es pot compilar). Així mateix, dins el mètode `main()` estem obligats a incloure les crides al mètode dins un bloc "try-catch".

L'execució del programa genera la sortida següent:

```
Punt 1.
Sortida de verificaLengthTaula.
Punt 2.
java.lang.Exception: La taula no té la llargada indicada.
    at Excepcio09.verificaLengthTaula(Excepcio09.java:12)
    at Excepcio09.main(Excepcio09.java:23)
Programa finalitzat.
```

Veiem que la segona crida al mètode `verificaLengthTaula`, amb valor 2 com a primer paràmetre, provoca el llançament de l'excepció `Exception`, i s'avorta l'execució del mètode (el segon missatge de sortida no apareix) i es propaga l'excepció cap al mètode superior: el `main()`, que la captura adequadament. Fixem-nos que el missatge Punt 3 ja no apareix.

Exemple de llançament d'excepció RuntimeException dins un mètode

El fitxer .java següent ens mostra un exemple de mètode que llança una excepció `RuntimeException`. Es tracta del mateix programa que el de l'exemple anterior, però substituint l'excepció ExceptionExcepcio10.java que es llança per una excepció `RuntimeException`.

```java
//Fitxer Excepcio10.java

public class Excepcio10 {
```

```
4
5     /* Mètode que avalua si la taula t té n cel·les, provocant, en cas de ser
          avaluada com a fals,
6        una excepció RuntimeException */
7     public static void verificaLengthTaula (int n, String t[]) {
8        if (t.length!=n) throw new RuntimeException ("La taula no té la llargada
             indicada.");
9        System.out.println ("Sortida de verificaLengthTaula.");
10    }
11
12    public static void main (String args[]) {
13       System.out.println("Punt 1.");
14       verificaLengthTaula (4, new String[4]);
15       System.out.println("Punt 2.");
16       verificaLengthTaula (2, new String[4]);
17       System.out.println("Punt 3.");
18       System.out.println ("Programa finalitzat.");
19    }
20 }
```

Fixem-nos que en llançar una excepció `RuntimeException` no estem obligats a declarar l'excepció en la capçalera del mètode. Si decidim, però, afegir la clàusula `throws RuntimeException` el compilador no es queixa però no té cap efecte en els mètodes superiors. Així mateix, dins el mètode main() no estem obligats a incloure les crides al mètode dins un bloc "try-catch".

L'execució del programa genera la sortida següent:

```
1 Punt 1.
2 Sortida de verificaLengthTaula.
3 Punt 2.
4 Exception in thread "main" java.lang.RuntimeException: La taula no té la
      llargada indicada.
5     at Excepcio10.verificaLengthTaula(Excepcio10.java:12)
6     at Excepcio10.main(Excepcio10.java:21)
```

En aquest cas, l'execució del `main()` no finalitza correctament, ja que es produeix l'excepció no gestionada.

1.1.5 Creació

El programador pot crear les pròpies excepcions a partir de la derivació de la classe `Exception` o d'alguna de les seves classes derivades. El lògic és heretar de la classe de la jerarquia de classes que s'adapti millor al tipus d'excepció, i cal tenir en compte que, si s'hereta a partir de la classe `RuntimeException` o d'alguna de les seves subclasses, la nova excepció no serà de gestió obligatòria.

En tractar-se de classes, com qualsevol altra classe, pot contenir variables i mètodes nous que s'afegeixen als heretats de la classe de la qual derivi.

Exemple de classe excepció creada pel programador

```
1 //Fitxer Excepcio11.java
2
```

```
3  public class Excepcio11 {
4     public static void main (String args[]) {
5        try {
6           provocoExcepcio(0);
7           provocoExcepcio(10);
8        } catch (LaMevaExcepcio e) {
9           e.printStackTrace();
10       }
11       System.out.println ("El programa finalitza correctament");
12    }
13
14    public static void provocoExcepcio(int valor) throws LaMevaExcepcio {
15       System.out.println ("Valor: " + valor);
16       if (valor!=0) throw new LaMevaExcepcio (valor);
17       System.out.println ("No s'ha provocat l'excepció.");
18    }
19 }
```

```
1  //Fitxer LaMevaExcpecio.java
2
3  public class LaMevaExcepcio extends Exception {
4     private Integer valor;
5
6     public LaMevaExcepcio (int xxx) {
7        valor = new Integer(xxx);
8     }
9
10    public String toString () {
11       return "Exception LaMevaExcepcio: Error motivat per valor = " + valor.
             toString();
12    }
13 }
```

Observem l'execució del programa:

```
1  Valor: 0
2  No s'ha provocat l'excepció.
3  Valor: 10
4  Exception LaMevaExcepcio: Error motivat per valor = 10
5      at Excepcio11.provocoExcepcio(Excepcio11.java:25)
6      at Excepcio11.main(Excepcio11.java:13)
7  El programa finalitza correctament
```

1.1.6 Efecte de l'herència

Si un mètode d'una classe és una sobreescriptura d'un mètode de la classe base que incorpora la clàusula `throws`, el mètode sobreescrit no ha de llençar necessàriament les mateixes excepcions que el mètode de la classe base: pot llançar les mateixes excepcions o menys, però no més excepcions que el mètode sobreescrit.

Aquesta restricció existeix per permetre que els mètodes que treballen amb referències a una classe base també puguin treballar amb referències que en realitat apuntin a objectes de classes derivades mantenint la gestió d'excepcions.

1.2 Col·leccions

Una col·lecció és una agrupació d'elements (objectes) en la qual s'hi han de poder executar diverses accions: afegir, recórrer, cercar, extreure... Tradicionalment, les estructures pensades per a l'organització de la informació s'han classificat segons el tipus d'accés que proporcionen. El llenguatge Java, conscient de la importància d'aquesta organització, proporciona un conjunt complet d'estructures que abraça les diverses possibilitats d'organització de la informació, i constitueix el conegut *framework* de col·leccions (*Java collections framework* o JCF).

El ***framework* de col·leccions** de Java és una arquitectura unificada per representar i gestionar col·leccions, independent dels detalls de la implementació.

El *framework* de col·leccions de Java, que neix en la versió 1.2 de Java, i en la versió 1.5 incorpora els tipus genèrics, està format per tres tipologies de components:

- **Interfícies.** Tipus abstractes de dades (TAD) que defineixen la funcionalitat de les col·leccions i funcionalitats de suport.
- **Implementacions.** Classes que implementen les interfícies de les col·leccions, de manera que una interfície pot tenir més d'una implementació (classe que la implementi).
- **Algorismes.** Mètodes que efectuen càlculs (cerques, ordenacions...) en els objectes de les implementacions.

1.2.1 Classes genèriques

Les Col.leccions que proporciona el Java pertanyen a un tipus de classes especials, anomenades classes genèriques. Aquestes proporcionen a solució a un problema molt important que val la pena veure amb detall abans d'aprofundir-hi.

Les **classes genèriques** són classes que encapsulen dades i mètodes basats en tipus de dades genèrics i serveixen de plantilla per generar classes a partir de concretar els tipus de dades genèrics.

La utilització més habitual de les classes genèriques es dóna en el disseny de classes pensades per a la gestió de conjunts de dades (llistes, piles, cues, arbres...) en els quals, si no existís aquest concepte, hauríem de dissenyar tantes classes com diferents tipus de dades s'haguessin de gestionar. Així, per exemple, podríem tenir:

```
class llistaInteger.../* Gestió de llista d'objectes Integer
class llistaPersona.../* Gestió de llista d'objectes Persona
class llistaDate.../* Gestió de llista d'objectes Date
```

Fixem-nos que les tres llistes es diferencien en el tipus de dada que emmagatzemen (`Integer`, `Persona`, `Date`) però els mètodes a implementar en les tres llistes són idèntics (`afegirPerInici`, `afegirPelFinal`, `recorregut`, `extreurePrimerElement`, `extreureDarrerElement`, etc) i, si no disposéssim de les classes genèriques, hauríem d'implementar tres llistes diferents amb la repetició consegüent de codi idèntic.

Classes parametritzades

Les classes genèriques també es coneixen com a *classes parametritzades* o *classes plantilla* i el llenguatge Java les incorpora des de la versió 1.5(Java 5).

La sintaxi per definir una classe genèrica en Java és:

```
[public] [final|abstract] class NomClasse <T1, T2...>
{
...
}
```

`T1`, `T2`... fan referència als tipus de dades genèrics gestionats pels mètodes de la classe genèrica, i s'han d'explicitar en la declaració de les referències a la classe genèrica i en la creació dels objectes de la classe genèrica, emprant la sintaxi següent:

```
NomClasse obj<nomTipusConcret1,nomTipusConcret2...>;
obj = new NomClasse<nomTipusConcret1,nomTipusConcret2...>(...);
```

En la utilització de tipus genèrics és important, per a la seguretat del codi desenvolupat, indicar els tipus específics corresponents als tipus genèrics en la creació dels objectes. És a dir, tot i tractar-se d'una classe genèrica, podríem crear objectes sense indicar els tipus específics corresponents als tipus genèrics:

```
NomClasse obj = new NomClasse (...);
```

Exemple de definició de classe genèrica amb diversos tipus parametritzats

La classe següent és un exemple de classe genèrica amb dos tipus parametritzats, i en el mètode `main()` es veu que es pot cridar amb diferents tipus de dades. La classe que es presenta no té altra utilitat que servir d'exemple en l'inici de l'aprenentatge de les classes genèriques.

```
import java.util.Date;
import java.awt.Color;

public class ExempleClasseGenerica <T1, T2> {
   private T1 x1;
   private T2 x2;

   public ExempleClasseGenerica (T1 p1, T2 p2) {
      x1=p1;
      x2=p2;
   }

   public String toString() {
      return x1.toString()+" – " +x2.toString();
   }
```

```
16
17    public T1 getX1() { return x1; }
18
19    public T2 getX2() { return x2; }
20
21    public static void main (String args[]) {
22       ExempleClasseGenerica <Integer, Float> obj1 =
23          new ExempleClasseGenerica <Integer,Float> (new Integer(20), new Float
              (42.45));
24       ExempleClasseGenerica <Double, Double> obj2 =
25          new ExempleClasseGenerica <Double,Double> (new Double(4.32), new
              Double(7.45));
26       ExempleClasseGenerica obj3 = new ExempleClasseGenerica (new Integer(22),
             new Date());
27       ExempleClasseGenerica obj4 = new ExempleClasseGenerica (50, 'a');
28       System.out.println(obj1.toString());
29       System.out.println(obj2.toString());
30       System.out.println(obj3.toString());
31       System.out.println(obj4.toString());
32    }
33 }
```

Fixem-nos que la classe genèrica no incorpora, en la definició, cap crida a cap operació específica dels tipus T1 i T2. Si no, no hauríem pogut compilar el fitxer font.

Fixem-nos, també, en el següent:

- La referència "obj1" declarada com a "ExempleClasseGenerica <Integer, Float>" recull l'objecte creat en cridar el constructor de la classe genèrica passant-li, per paràmetres, una referència a un objecte `Integer`, que recull l'argument "T1", i una referència a un objecte `Float`, que recull l'argument T2. Aquesta crida provoca la creació de la classe `ExempleClasseGenerica <Integer, Float>` en temps d'execució.
- La referència "obj2" declarada com a "ExempleClasseGenerica <Double, Double>" recull l'objecte creat en cridar el constructor de la classe genèrica passant-li, per paràmetres, dues referències a objectes Double, que recullen els arguments "T1" i "T2". Aquesta crida provoca la creació de la classe `ExempleClasseGenerica <Double, Double>` en temps d'execució.
- La referència "obj3" declarada sense especificar els tipus corresponents a "T1" i "T2" recull l'objecte creat en cridar el constructor de la classe genèrica passant-li, per paràmetres, una referència a un objecte `Integer`, que recull l'argument "T1", i una referència a un objecte `Date`, que recull l'argument "T2". Aquesta crida provoca la creació, per la màquina virtual, de la classe `ExempleClasseGenerica <Integer, Date>` en temps d'execució.
- La referència "obj4" declarada sense especificar els tipus corresponents a "T1" i "T2" recull l'objecte creat en cridar el constructor de la classe genèrica passant-li, per paràmetres, una dada del tipus primitiu `int` que recull l'argument "T1" com si fos un `Integer`, i una referència a un objecte `String`, que recull l'argument "T"2. Aquesta crida provoca la creació, per la màquina virtual, de la classe `ExempleClasseGenerica <Integer, String>` en temps d'execució.

Veiem que la compilació d'aquest fitxer ens dóna un avís:

```
Note: ExempleClasseGenerica.java uses unchecked or unsafe operations.
Note: Recompile with –Xlint:unchecked for details.
```

El compilador ens està avisant de que estem cridant la classe `ExempleClasseGenerica` sense indicar el tipus parametritzat, i té raó, ja que en les creacions dels objectes apuntats per "obj3" i "obj4" no s'han explicitat els tipus parametritzats "T1" i "T2". L'avís ens diu que efectuant la compilació amb el paràmetre `-Xlint:unchecked` obtindrem més detalls:

```
ExempleClasseGenerica.java:34: warning: [unchecked] unchecked call to
     ExempleCla
sseGenerica(T1,T2) as a member of the raw type ExempleClasseGenerica
    ExempleClasseGenerica obj3 = new ExempleClasseGenerica (new Integer(22), n
ew Date());
        ^
ExempleClasseGenerica.java:35: warning: [unchecked] unchecked call to
     ExempleCla
sseGenerica(T1,T2) as a member of the raw type ExempleClasseGenerica
    ExempleClasseGenerica obj4 = new ExempleClasseGenerica (50, "Hola");
        ^
2 warnings
```

L'execució, però, es duu a terme sense problemes:

```
20 – 42.45
4.32 – 7.45
22 – Mon Mar 30 05:36:59 CEST 2009
50 – Hola
```

Tot i que el nostre exemple s'ha executat sense problemes, ara suposem que sobre el membre x2 de l'objecte apuntat per obj3, que sabem que és del tipus Date, volem conèixer el dia del mes de la data corresponent (que, segons el resultat vist en la darrera execució, hauria de ser el valor 30). Per aconseguir-ho, hauríem de fer una cosa així:

```
System.out.println(obj3.getX2().getDate());
```

La compilació del fitxer amb aquesta instrucció afegida al final, provoca l'error:

```
ExempleClasseGenerica.java:40: cannot find symbol
symbol  : method getDate()
location: class java.lang.Object
   System.out.println(obj3.getX2().getDate());
        ^
Note: ExempleClasseGenerica.java uses unchecked or unsafe operations.
Note: Recompile with –Xlint:unchecked for details.
1 error
```

És clar que el programador sap que l'objecte retornat per "obj3.getX2()" és del tipus `Date`, però el compilador no ho sap perquè no li hem dit en la creació de l'objecte "obj3" i, per tant, per obtenir la compilació, haurem d'explicitar una conversió *cast*:

```
System.out.println(((Date)obj3.getX2()).getDate());
```

Però, si el programador es confon i efectua una conversió *cast* errònia com la següent:

```
System.out.println(((Color)obj3.getX2()).getAlpha());
```

El compilador s'empassa sense cap problema aquesta conversió *cast*, ja que no té cap mecanisme, en temps de compilació, per saber si el membre "x2" de d'"obj3" és de la classe `Color`. L'error es produirà en temps d'execució.

La necessitat de la conversió *cast* i els possibles errors de conversió desapareixen si en la creació de l'objecte s'indica el tipus específic corresponent a cada tipus genèric:

```
ExempleClasseGenerica <Integer, Date> obj3 =
   new ExempleClasseGenerica <Integer, Date> (new Integer(22), new Date());
```

Havent creat l'objecte "obj3" com indica la darrera instrucció, no és necessària la conversió *cast* per aplicar el mètode "getDate()" directament sobre "x2.obj3". Però és que, a més, si el programador intenta fer una conversió *cast* similar a "(Color)obj3.getX2()", el compilador detecta la incoherència de la conversió i no compila el fitxer.

Per tant, per a la seguretat del codi desenvolupat, és important indicar els tipus específics corresponents als tipus genèrics en la creació dels objectes.

Abans de l'aparició dels tipus genèrics (Java 5) s'utilitzava la classe `Object` com a comodí per simular els tipus genèrics, però això provocava els problemes següents:

- En dissenyar estructures contenidores d'objectes (piles, cues, llistes, arbres...) no hi havia una manera senzilla de delimitar els tipus dels objectes a inserir i, si no s'anava amb compte, en una mateixa estructura contenidora hi podia haver objectes dispars.
- Sovint calia fer conversions *cast* de la classe Object a la classe corresponent, amb perill d'equivocacions per part del programador que no es detecten fins a l'execució del programa, com s'ha comprovat al final del darrer exemple.

De vegades és necessari implementar mètodes que admetin, per paràmetres, objectes de classes genèriques. La sintaxi a utilitzar és:

```
<modificadorAccés> [final|abstract] nomMètode ( NomClasse<?,?...> [,...])
{...}
```

Exemple de classe dissenyada amb tipus genèrics

L'exemple següent mostra el disseny d'un mètode que pot rebre, per paràmetre, objectes de la classe genèrica dissenyada en l'exemple anterior.

```java
import java.util.Date;

public class MetodeAmbClasseGenerica {
   public static void metode (ExempleClasseGenerica<?,?> obj) {
      System.out.println(obj);
   }

   public static void main (String args[]) {
      ExempleClasseGenerica <Integer, Float> obj1 =
         new ExempleClasseGenerica <Integer,Float> (new Integer(20), new Float
             (42.45));
      ExempleClasseGenerica <Double, Date> obj2 =
         new ExempleClasseGenerica <Double,Date> (new Double(4.32), new Date())
             ;
      metode(obj1);
      metode(obj2);
   }
}
```

L'execució del mètode "main()" dóna el resultat:

```
20 - 42.45
4.32 - Mon Mar 30 07:12:00 CEST 2009
```

Donada una classe genèrica, podem restringir els tipus específics amb els que es creïn els seus objectes? La resposta és afirmativa: es pot declarar un tipus genèric de manera que en instanciar la classe genèrica s'hagi de proporcionar una classe que sigui subclasse d'una classe `X`. La sintaxi és:

```
[public] [final|abstract] NomClasse <T1 extends X, T2 extends Y...> {
...
}
```

Veiem que s'utilitza la paraula reservada `extends`.

Exemple complet de classes genèriques

Vegem un exemple de classe genèrica amb tipus genèric restringit i amb mètodes que reben, per paràmetre, objectes de la classe genèrica.Es tracta del disseny d'una classe que gestiona taules de valors numèrics i proporciona mètodes per calcular la mitjana dels valors emmagatzemats en la taula, la dimensió de les taules i per comparar les mitjanes i les dimensions de dues taules de valors numèrics.

La taula ha de ser genèrica perquè pugui gestionar els diversos tipus de valors numèrics possibles (`Integer`, `Double`, `Float`...), i la volem restringir perquè les taules només siguin de valors numèrics. Els mètodes de comparació necessiten rebre, com a paràmetre, un objecte de classe genèrica.

```java
public class TaulaValorsNumerics <T extends Number> {
  private T t[];

  public TaulaValorsNumerics (T[] obj) { t = obj; }

  public double mitja () {
   double suma=0;
   int valorsNoNuls=0;
   for (int i=0; i<t.length; i++)
```

```
    if (t[i]!=null)
    {suma = suma + t[i].doubleValue();
     valorsNoNuls++;
    }

   if (valorsNoNuls==0) return 0;
   else return suma / valorsNoNuls;
  }

  public int dimensio () { return t.length; }

  public boolean mateixaMitja (TaulaValorsNumerics <?> obj) {
    return mitja() == obj.mitja();
  }

  public boolean mateixaDimensio (TaulaValorsNumerics <?> obj) {
   return dimensio() == obj.dimensio();
  }

  public String toString() {
   String s="{";
   for (int i=0; i<t.length; i++)
    if (s.equals("{")) s = s + t[i];
    else s = s + ", " + t[i];
   s = s + "}";
   return s;
  }

  public static void main (String args[]) {
   Integer ti[] = {1, 2, null, 3, 4, null, 5};
   Double td[] = {1.1, 2.2, 3.3, null, 4.4, 5.5, 6.6};
   String ts[] = { "Cad1", "Cad2" };
   TaulaValorsNumerics<Integer> tvn1= new TaulaValorsNumerics<Integer> (ti);
   TaulaValorsNumerics<Double> tvn2= new TaulaValorsNumerics<Double> (td);

// Les següents instruccions comentades no són compilables, per ser tipus genè
    ric T restringit
// TaulaValorsNumerics tvn3 = new TaulaValorsNumerics (ts);
// TaulaValorsNumerics<String> tvn3 = new TaulaValorsNumerics<String> (ts);

   System.out.println("tvn1: "+tvn1);
   System.out.println("Mitja: "+tvn1.mitja());
   System.out.println("Dimensió: "+tvn1.dimensio());
   System.out.println("tvn2: "+tvn2);
   System.out.println("Mitja: "+tvn2.mitja());
   System.out.println("Dimensió: "+tvn2.dimensio());
   System.out.println("tvn1 i tvn2 tenen la mateixa mitja?" + tvn1.mateixaMitja
       (tvn2));
   System.out.println("tvn1 i tvn2 tenen la mateixa dimensió?" + tvn1.
       mateixaDimensio(tvn2));
  }
}
```

L'execució del mètode "main()" mostra:

```
tvn1: {1, 2, null, 3, 4, null, 5}
   Mitja: 3.0
   Dimensió: 7
tvn2: {1.1, 2.2, 3.3, null, 4.4, 5.5, 6.6}
   Mitja: 3.85
   Dimensió: 7
tvn1 i tvn2 tenen la mateixa mitja?false
tvn1 i tvn2 tenen la mateixa dimensió?true
```

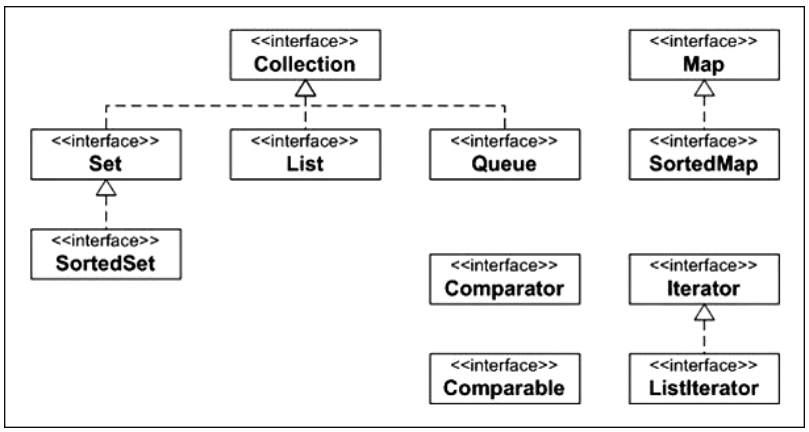
1.2.2 Interfícies

Les interfícies són tipus abstractes de dades (TAD) que defineixen la funcionalitat de les col·leccions i funcionalitats de suport. En el *framework* de col·leccions cal distingir-ne dos tipus:

- Interfícies que constitueixen el cor del *framework* i defineixen els diferents tipus de col·leccions: `Collection`, `Set`, `List`, `Queue`, `SortedSet`, `Map` i `SortedMap`, conegudes normalment com *interfícies del JCF*. Aquest conjunt d'interfícies, al seu torn, està organitzat en dues jerarquies: una, que agrupa les col·leccions amb accés per posició (seqüencial i directe) i que té la interfície Collection per arrel, i l'altra, que defineix les col·leccions amb accés per clau i que té la interfície Map per arrel.
- Interfícies de suport: `Iterator`, `ListIterator`, `Comparable` i `Comparator`.

La figura 1.3 mostra les interfícies del *framework* de col·leccions de Java amb la jerarquia que hi ha entre elles.

FIGURA 1.3. Interfícies del framework de col·leccions de Java



L'explicació detallada de cadascuna de les interfícies del *framework* de col·leccions cal cercar-la en la documentació del llenguatge Java.

Interfície "Collection"

La interfície `Collection<E>` és la interfície pare de la jerarquia de col·leccions amb accés per posició (seqüencial i directe) i comprèn col·leccions de diversos tipus:

- Col·leccions que permeten elements duplicats i col·leccions que no els permeten.
- Col·leccions ordenades i col·leccions desordenades.

- Col·leccions que permeten el valor `null` i col·leccions que no el permeten.

La seva definició, extreta de la documentació de Java, és força autoexplicativa atesos els noms que utilitzen:

```
public interface Collection<E> extends Iterable<E> {
// Basic operations
int size();
boolean isEmpty();
boolean contains(Object element);
boolean add(E element);//optional
boolean remove(Object element);//optional
Iterator<E> iterator();

// Bulk operations
boolean containsAll(Collection<?> c);
boolean addAll(Collection<? extends E> c);//optional
boolean removeAll(Collection<?> c);//optional
boolean retainAll(Collection<?> c);//optional
void clear(); //optional

// Array operations
Object[] toArray();
<T> T[] toArray(T[] a);
}
```

El comentari *optional* acompanyant un mètode d'una interfície indica que el mètode pot no estar disponible en alguna de les implementacions de la interfície. Això pot passar si un mètode de la interfície no té sentit en una implementació concreta. La implementació ha de definir el mètode, però en ser cridat provoca una excepció *UnsupportedOperationException*.

El llenguatge Java no proporciona cap classe que implementi directament aquesta interfície, sinó que implementa interfícies derivades d'aquesta. Això no és un impediment per implementar directament, quan convingui, aquesta interfície, però la majoria de vegades n'hi ha prou d'implementar les interfícies derivades o, fins i tot, derivar directament de les implementacions que Java proporciona de les diverses interfícies.

El recorregut pels elements d'una col·lecció es pot efectuar, en principi, gràcies al mètode `iterator()`, que retorna una referència a la interfície `Iterator`.

Interfícies "Iterator" i "ListIterator"

Les interfícies de suport `Iterator` i `ListIterator` utilitzades per diversos mètodes de les classes que implementen la interfície `Collection` o les seves subinterfícies permeten recórrer els elements d'una col·lecció.

La definició de la interfície `Iterator` és:

```
public interface Iterator<E>{
  boolean hasNext();
  E next();
  void remove();// optional
}
```

Així, per exemple, la referència que retorna el mètode `iterator()` de la interfície `Collection` permet recórrer la col·lecció amb els mètodes `next()` i `hasNext()`, i també permet eliminar l'element actual amb el mètode remove() sempre que la col·lecció permeti l'eliminació dels seus elements.

La interfície `ListIterator` permet recórrer els objectes que implementen la interfície `List` (subinterfície de `Collection`) en ambdues direccions (endavant i endarrere) i efectuar algunes modificacions mentre s'efectua el recorregut. Els mètodes que defineix la interfície són:

```
1  public interface ListIterator<E> extends Iterator<E>{
2    boolean hasNext();
3    E next();
4    boolean hasPrevious();
5    E previous();
6    int nextIndex();
7    int previousIndex();
8    void remove();
9    void set(E e);
10   void add(E e);
11 }
```

El recorregut pels elements de la col·lecció (llista) s'efectua amb els mètodes `next()` i `previous()`. En una llista amb n elements, els elements es numeren de 0 a n-1, però els valors vàlids per a l'índex iterador són de 0 a n, de manera que l'índex x es troba entre els elements x-1 i x i, per tant, el mètode `previousIndex()` retorna x-1 i el mètode `nextIndex()` retorna x; si l'índex és 0, `previousIndex()` retorna -1, si l'índex és n, `nextIndex()` retorna el resultat del mètode `size()`.

Interfícies "Comparable" i "Comparator"

Les interfícies de suport `Comparable` i `Comparator` estan orientades a mantenir una relació d'ordre en les classes del *framework* de col·leccions que implementen interfícies que faciliten ordenació (`List`, `SortedSet` i `SortedMap`).

La interfície Comparable consta d'un únic mètode:

```
1  public interface Comparable<T> {
2    int compareTo(T o);
3  }
```

El mètode `compareTo()` compara l'objecte sobre el qual s'aplica el mètode amb l'objecte rebut per paràmetre i retorna un enter negatiu, zero o positiu en funció de si l'objecte sobre el qual s'aplica el mètode és menor, igual o major que l'objecte rebut per paràmetre. Si els dos objectes no són comparables, el mètode ha de generar una excepció `ClassCastException`.

El llenguatge Java proporciona un munt de classes que implementen aquesta interfície (`String, Character, Date, Byte, Short, Integer,Long, Float, Double, BigDecimal, BigInteger...`). Per tant, totes aquestes classes proporcionen una implementació del mètode `compareTo()`.

Tota implementació del mètode `compareTo()` hauria de ser compatible amb el mètode `equals()`, de manera que `compareTo()` retorni zero únicament si

`equals()` retorna true i, a més, hauria de verificar la propietat transitiva, com en qualsevol relació d'ordre.

De les classes que implementen la interfície Comparable es diu que tenen un ordre natural.

Les col·leccions de classes que implementen la interfície Comparable es poden ordenar amb els mètodes `static Collections.sort()` i `Arrays.sort()`.

Exemple d'utilització de la interfície Comparable en la classe Persona

Considerem la classe `Persona`. Hi tenim definit un mètode `equals()` que informa sobre la igualtat de dues persones a partir del dni. Ara ens interessa ampliar aquest concepte amb una relació d'ordre, de manera que puguem comparar persones i, per tant, puguem, per exemple, ordenar una taula de persones.

Per aconseguir-ho, cal fer que la classe Persona implementi la interfície Comparable i cal programar-hi el mètode compareTo():

```
1  public class Persona implements Comparable {
2    private String dni;
3
4     ...
5
6     public final int compareTo (Object obj) {
7        if (obj instanceof Persona) return dni.compareTo(((Persona)
             obj).dni);
8        throw new ClassCastException();  // Instrucció que genera
             una excepció
9     }
10
11    ...
12 }
```

Amb aquesta versió de la classe `Persona`, podem utilitzar el mètode `Arrays.sort()` per ordenar una taula de persones (bé, d'objectes de classes derivades perquè la classe Persona és abstracta). El programa següent ens ho demostraria:

```
1  //Fitxer ProvaPersona.java
2
3  import java.util.Arrays;
4
5  public class ProvaPersona {
6     public static void main(String args[]) {
7        Persona t[] = new Persona[6];
8        t[0] = new Alumne("99999999","Anna",20);
9        t[1] = new Alumne("00000000","Pep",33,'m');
10       t[2] = new Alumne("22222222","Maria",40,'s');
11       t[3] = new Alumne("66666666","Àngel",22);
12       t[4] = new Alumne("11111111","Joanna",25,'M');
13       t[5] = new Alumne("55555555","Teresa",30,'S');
14
15       System.out.println("Contingut inicial de la taula:");
16       for (int i=0; i<t.length; i++) System.out.println("   "+t[i
             ]);
17       Arrays.sort(t);
18       System.out.println("Contingut de la taula després d'haver
             estat ordenada:");
19       for (int i=0; i<t.length; i++) System.out.println("   "+t[i
             ]);
20    }
21 }
```

La seva execució dóna el resultat:

```
Contingut inicial de la taula:
   Dni: 99999999 – Nom: Anna – Edat: 20 – Nivell: ???
   Dni: 00000000 – Nom: Pep – Edat: 33 – Nivell: Cicle F. Mitjà
   Dni: 22222222 – Nom: Maria – Edat: 40 – Nivell: Cicle F.
       Superior
   Dni: 66666666 – Nom: Àngel – Edat: 22 – Nivell: ???
   Dni: 11111111 – Nom: Joanna – Edat: 25 – Nivell: Cicle F. Mitj
       à
   Dni: 55555555 – Nom: Teresa – Edat: 30 – Nivell: Cicle F.
       Superior
Contingut de la taula després d'haver estat ordenada:
   Dni: 00000000 – Nom: Pep – Edat: 33 – Nivell: Cicle F. Mitjà
   Dni: 11111111 – Nom: Joanna – Edat: 25 – Nivell: Cicle F. Mitj
       à
   Dni: 22222222 – Nom: Maria – Edat: 40 – Nivell: Cicle F.
       Superior
   Dni: 55555555 – Nom: Teresa – Edat: 30 – Nivell: Cicle F.
       Superior
   Dni: 66666666 – Nom: Àngel – Edat: 22 – Nivell: ???
   Dni: 99999999 – Nom: Anna – Edat: 20 – Nivell: ???
```

La interfície `Comparator` permet ordenar conjunts d'objectes que pertanyen a classes diferents. Per establir un ordre en aquests casos, el programador ha de subministrar un objecte d'una classe que implementi aquesta interfície:

```
public interface Comparator<T> {
  int compare(T o1, T o2);
  boolean equals(Object obj);
}
```

L'objectiu del mètode `equals()` és comparar objectes `Comparator`. En canvi, l'objectiu del mètode `compare()` és comparar dos objectes de diferents classes i obtenir un enter negatiu, zero o positiu, en funció de si l'objecte rebut per primer paràmetre és menor, igual o major que l'objecte rebut per segon paràmetre.

Exemple d'utilització de la interfície Comparator

La situació que presentem aquí no és gens lògica, però ens serveix per introduir un exemple d'utilització clara de la interfície `Comparator`. Suposem que tenim una taula que conté objectes de les classes `Persona`, `Date`, `Double` i `Short` i la volem ordenar. Evidentment, entre aquests elements no hi ha cap ordre natural.

Suposem que es decideix que cal ordenar una tal taula deixant, en primer lloc, els elements Date, seguits dels elements `Double`, després els elements `Persona` i, finalment, els elements `Short`. A més, entre el grup d'elements d'un mateix tipus, es demana que actuï l'ordre natural definit en aquest tipus (definit per la implementació del mètode `compareTo()` de la interfície `Comparable`).

El programa següent mostra com es pot dissenyar una classe xComparator que implementa la interfície Comparator i permet definir un objecte que, passat al mètode `Arrays.sort()`, permet ordenar una taula com la indicada en l'ordre demanat.

```
//Fitxer ProvaComparator.java

import java.util.*;

public class ProvaComparator {
   public static void main (String args[]) {
      Object t[] = new Object[6];
      t[0] = new Alumne("99999999","Anna",20);
      t[1] = new Date();
      t[2] = new Alumne("22222222","Maria",40,'s');
      t[3] = new Double(33.33);
```

```
12       t[4] = new Short((short)22);
13       t[5] = new Date(109,0,1);
14       System.out.println("Contingut inicial de la taula:");
15       for (int i=0; i<t.length; i++)
16          System.out.println("   "+t[i].getClass().getName()+" - "
                +t[i]);
17       Arrays.sort(t,new MyComparator());
18       System.out.println("Contingut de la taula després d'haver
             estat ordenada:");
19       for (int i=0; i<t.length; i++)
20          System.out.println("   "+t[i].getClass().getName()+" - "
                +t[i]);
21    }
22 }
```

```
1  //Fitxer MyComparator.java
2
3  public class MyComparator implements Comparator <Object> {
4     public int compare (Object o1, Object o2) {
5        if (o1 instanceof Date)
6           if (o2 instanceof Date) return ((Date)o1).compareTo((
                Date)o2);
7           else return -1;
8        else if (o1 instanceof Double)
9           if (o2 instanceof Date) return 1;
10          else if (o2 instanceof Double) return ((Double)o1).
                compareTo((Double)o2);
11               else return -1;
12       else if (o1 instanceof Persona)
13          if (o2 instanceof Date || o2 instanceof Double) return
                1;
14          else if (o2 instanceof Persona) return ((Persona)o1).
                compareTo((Persona)o2);
15               else return -1;
16       else if (o1 instanceof Short)
17          if (o2 instanceof Short) return ((Short)o1).compareTo((
                Short)o2);
18          else return 1;
19       else throw new ClassCastException();  // Instrucció que
             genera una excepció
20    }
21 }
```

L'execució del programa dóna el resultat:

```
1  Contingut inicial de la taula:
2     Alumne - Dni: 99999999 - Nom: Anna - Edat: 20 - Nivell: ???
3     java.util.Date - Sun Apr 05 18:26:00 CEST 2009
4     Alumne - Dni: 22222222 - Nom: Maria - Edat: 40 - Nivell: Cicle
          F. Superior
5     java.lang.Short - 22
6     java.lang.Double - 33.33
7     java.util.Date - Thu Jan 01 00:00:00 CET 2009
8  Contingut de la taula després d'haver estat ordenada:
9     java.util.Date - Thu Jan 01 00:00:00 CET 2009
10    java.util.Date - Sun Apr 05 18:26:00 CEST 2009
11    java.lang.Double - 33.33
12    Alumne - Dni: 22222222 - Nom: Maria - Edat: 40 - Nivell: Cicle
          F. Superior
13    Alumne - Dni: 99999999 - Nom: Anna - Edat: 20 - Nivell: ???
14    java.lang.Short - 22
```

Interfícies "Set" i "SortedSet"

La interfície `Set<E>` és destinada a col·leccions que no mantenen cap ordre d'inserció i que no poden tenir dos o més objectes iguals. Correspon al concepte matemàtic de conjunt.

El conjunt de mètodes que defineix aquesta interfície és idèntic al conjunt de mètodes definits per la interfície `Collection<E>`.

Les implementacions de la interfície `Set<E>` necessiten el mètode `equals()` per veure si dos objectes del tipus de la col·lecció són iguals i, per tant, no permetre'n la coexistència dins la col·lecció.

La crida `set.equals(Object obj)` retorna `true` si "obj" també és una instància que implementi la interfície Set, els dos objectes ("set" i "obj") tenen el mateix nombre d'elements i tots els elements d'"obj" estan continguts en set. Per tant, el resultat pot ser `true` malgrat que set i "obj" siguin de classes diferents però que implementen la interfície Set.

Els mètodes de la interfície Set permeten realitzar les operacions algebraiques unió, intersecció i diferència:

- `s1.containsAll(s2)` permet saber si "s2" està contingut en "s1".
- `s1.addAll(s2)` permet convertir s1 en la unió d'"s1" i "s2".
- `s1.retainAll(s2)` permet convertir "s1" en la intersecció d'"s1" i "s2".
- `s1.removeAll(s2)` permet convertir "s1" en la diferència d'"s1" i "s2".

La interfície `SortedSet` derivada de Set afegeix mètodes per permetre gestionar conjunts ordenats. La seva definició és:

```
1  public interface SortedSet<E> extends Set<E> {
2    // Range-view
3    SortedSet<E> subSet(E fromElement, E toElement);
4    SortedSet<E> headSet(E toElement);
5    SortedSet<E> tailSet(E fromElement);
6
7    // Endpoints
8    E first();
9    E last();
10
11   // Comparator access
12   Comparator<? super E> comparator();
13 }
```

La relació d'ordre a aplicar sobre els elements d'un objecte col·lecció que implementi la interfície `SortedSet` es defineix en el moment de la seva construcció, indicant una referència a un objecte que implementi la interfície `Comparator`. En cas de no indicar cap referència, els elements de l'objecte es comparen amb l'ordre natural.

El mètode `comparator()` retorna una referència a l'objecte `Comparator` que defineix l'ordre dels elements del mètode, i retorna `null` si es tracta de l'ordre natural.

En un objecte `SortedSet`, els mètodes `iterator()`, `toArray()` i `toString()` gestionen els elements segons l'ordre establert en l'objecte.

Interfície "List"

La interfície `List<E>` es destina a col·leccions que mantenen l'ordre d'inserció i que poden tenir elements repetits. Per aquest motiu, aquesta interfície declara mètodes addicionals (als definits en la interfície `Collection`) que tenen a veure amb l'ordre i l'accés a elements o interval d'elements:

```
1  public interface List<E> extends Collection<E> {
2    // Positional access
3    E get(int index);
4    E set(int index, E element);    //optional
5    void add(int index, E element); //optional
6    E remove(int index);            //optional
7    boolean addAll(int index, Collection<? extends E> c); //optional
8
9    // Search
10   int indexOf(Object o);
11   int lastIndexOf(Object o);
12
13   // Iteration
14   ListIterator<E> listIterator();
15   ListIterator<E> listIterator(int index);
16
17   // Range-view
18   List<E> subList(int from, int to);
19 }
```

La crida `list.equals(Object obj)` retorna `true` si "obj" també és una instància que implementa la interfície `List`, i els dos objectes tenen el mateix nombre d'elements i contenen elements iguals i en el mateix ordre. Per tant, el resultat pot ser `true` malgrat que "list" i "obj" siguin de classes diferents però que implementen la interfície `List`.

El mètode `add(E o)` definit en la interfície `Collection` afegeix l'element "o" pel final de la llista, i el mètode `remove(Object o)` definit en la interfície `Collection` elimina la primera aparició de l'objecte indicat.

Interfície "Queue"

La interfície `Queue<E>` es destina a gestionar col·leccions que guarden múltiples elements abans de ser processats i, per aquest motiu, afegeix els mètodes següents als definits en la interfície `Collection`:

```
1  public interface Queue<E> extends Collection<E> {
2    E element();
3    boolean offer(E e);
4    E peek();
5    E poll();
6    E remove();
7  }
```

En informàtica, quan es parla de cues, es pensa sempre en una gestió dels elements amb un algorisme FIFO (*first input, first output* -primer en entrar, primer en sortir-), però això no és obligatòriament així en les classes que implementen la

interfície Queue. Un exemple són les cues amb prioritat, en què els elements s'ordenen segons un valor per a cada element. Per tant, les implementacions d'aquesta interfície han de definir l'ordre dels seus elements si no es tracta d'implementacions FIFO.

Els mètodes típics en la gestió de cues (encuar, desencuar i inici) prenen, en la interfície `Queue`, dues formes segons la reacció davant una fallada en l'operació: mètodes que retornen una excepció i mètodes que retornen un valor especial (`null` o `false`, segons l'operació). La taula 1.1 ens mostra la classificació dels mètodes segons aquests criteris.

TAULA 1.1. Classificació dels mètodes de gestió de cues

Operació	Mètode que provoca excepció	Mètode que retorna un valor especial
Encuar	boolean add (E e)	boolean offer(E e)
Desencuar	E remove()	E poll()
Inici	E element()	E peek()

Interfícies "Map" i "SortedMap"

La interfície `Map<E>` es destina a gestionar agrupacions d'elements als quals s'accedeix mitjançant una clau, la qual ha de ser única per als diferents elements de l'agrupació. La definició és:

```
1  public interface Map<K,V> {
2
3    // Basic operations
4    V put(K key, V value);   //optional
5    V get(Object key);
6    V remove(Object key);    //optional
7    boolean containsKey(Object key);
8    boolean containsValue(Object value);
9    int size();
10   boolean isEmpty();
11
12   // Bulk operations
13   void putAll(Map<? extends K, ? extends V> m);  // optional
14   void clear();                                  // optional
15
16   // Collection Views
17   public Set<K> keySet();
18   public Collection<V> values();
19   public Set<Map.Entry<K,V>> entrySet();
20
21   // Interface for entrySet elements
22   public interface Entry {
23     K getKey();
24     V getValue();
25     V setValue(V value);
26   }
27 }
```

Molts d'aquests mètodes tenen un significat evident, però altres no tant.

El mètode `entrySet()` retorna una visió del `Map` com a `Set`. Els elements d'aquest Set són referències a la interfície `Map.Entry` que és una interfície interna

de `Map` que permet modificar i eliminar elements del `Map`, però no afegir-hi nous elements. El mètode `get(key)` permet obtenir l'element a partir de la clau. El mètode `keySet()` retorna una visió de les claus com a `Set`. El mètode `values()` retorna una visió dels elements del `Map` com a `Collection` (perquè hi pot haver elements repetits i com a `Set` això no seria factible). El mètode `put()` permet afegir una parella clau/element mentre que `putAll(map)` permet afegir-hi totes les parelles d'un `Map` passat per paràmetre (les parelles amb clau nova s'afegeixen i, en les parelles amb clau ja existent en el Map, els elements nous substitueixen els elements existents). El mètode `remove(key)` elimina una parella clau/element a partir de la clau.

La crida `map.equals(Object obj)` retorna `true` si "obj" també és una instància que implementa la interfície `Map` i els dos objectes representen el mateix mapatge o, dit amb altres paraules, si l'expressió:

```
map.entrySet().equals( ( (Map) obj ).entrySet() )
```

retorna `true`. Per tant, el resultat pot ser `true` malgrat que "map" i "obj" siguin de classes diferents però que implementen la interfície `Map`.

La interfície `SortedMap` és una interfície `Map` que permet mantenir ordenades les seves parelles en ordre ascendent segons el valor de la clau, seguint l'ordre natural o la relació d'ordre proporcionada per un objecte que implementi la interfície `Comparator` proporcionada en el moment de creació de la instància.

La seva definició, a partir de la interfície Map, és similar a la definició de la interfície `SortedSet` a partir de la interfície `Set`:

```
1  public interface SortedMap<K, V> extends Map<K, V> {
2    // Range-view
3    SortedMap<K, V> subMap(K fromKey, K toKey);
4    SortedMap<K, V> headMap(K toKey);
5    SortedMap<K, V> tailMap(K fromKey);
6    // Endpoints
7    K firstKey();
8    K lastKey();
9    // Comparator access
10   Comparator<? super K> comparator();
11 }
```

1.2.3 Implementacions

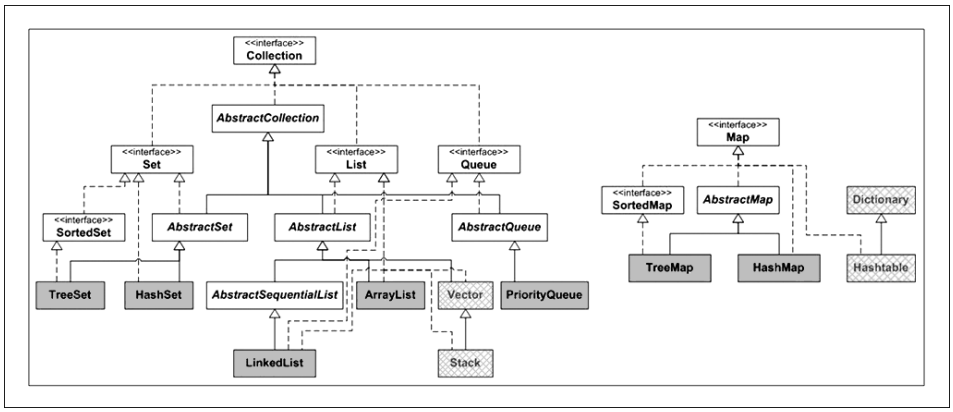
El *framework* de col·leccions de Java proporciona un ampli conjunt d'implementacions de les diverses interfícies per ser utilitzades directament en el desenvolupament d'aplicacions o per crear classes derivades.

La figura 1.4 mostra les implementacions més conegudes i utilitzades de les interfícies que deriven de les interfícies `Collection` i `Map`.

El *framework* de col·leccions, per facilitar la feina als programadors que vulguin crear classes implementant les interfícies, proporciona un conjunt de classes

abstractes que tenen parcialment o totalment implementats els mètodes de les interfícies corresponents, de manera que els programadors poden derivar les seves classes directament d'elles amb un mínim esforç de programació. La figura 1.4 presenta les classes no abstractes amb un fons no blanc per identificar-les de manera ràpida.

FIGURA 1.4. Jerarquia d'interfícies i implementacions més usuals a partir de les interfícies Collection iMap



Abans que aparegués el *framework*de col·leccions en la versió 1.2 de Java, ja hi havia unes classes pensades per a la gestió d'agrupacions d'objectes: `Vector`, `Stack` i `Hashtable`. Amb l'aparició del *framework*de col·leccions, aquestes classes històriques es van mantenir i incorporar al *fra- mework*, de manera que mantenen els mètodes originals i, a més, implementen les interfícies `List` (classes `Vector` i `Stack`) i Map (classe `Hashtable`). La figura 1.4 mostra les classes històriques amb un fons tramat grisós i incorpora la classe `Dictionary` perquè és classe base de la classe `Hashtable`.

Classe històrica "Vector"

La classe `Vector<E>`, a banda de derivar de la classe `AbstractList`, implementa la interfície `Cloneable` per poder clonar objectes Vector amb el mètode clone(), i la interfície `Serializable` per poder convertir objectes `Vector` en cadenes de caràcters.

Com el seu nom indica, `Vector<E>` representa una taula de referències a objectes de tipus "E" que, a diferència de les taules clàssiques de Java (*arrays*), pot créixer i reduir el nombre d'elements. També permet l'accés als seus elements amb un índex, encara que no permet la utilització dels claudàtors "[]" a diferència de les taules clàssiques de Java. Hi ha molts mètodes, entre els quals cal destacar:

- **El mètode capacity()**, que retorna la grandària o el nombre d'elements que pot tenir el vector. És el mètode equivalent a la propietatlength de les taules clàssiques de Java.
- **El mètode size()**, que retorna el nombre d'elements que realment conté el vector. Per tant, a diferència de les taules clàssiques de Java, no cal mantenir una variable entera com a comptador del nombre d'elements que conté el vector.

- **El mètode get(int n)** retorna la referència que hi ha en la posició indicada per n. Les posicions s'enumeren a partir de zero.
- **El mètode add(E obj)** permet afegir la referència obj després del darrer element que hi ha en el vector, i amplia automàticament la grandària del vector si aquest era ple.
- **El mètode add (int n, E obj)** permet inserir la referència obj a la posició indicada per n sempre que n>=0 i n<=`size()`.

Per a un coneixement profund de tots els mètodes, cal fer una ullada a la documentació que acompanya el llenguatge Java. Com a dades membres protegides d'aquesta classe cal conèixer:

- **capacityIncrement**, que indica l'increment que patirà el vector cada vegada que necessiti créixer.
- **elementCount**, que conté el nombre de components vàlids del vector.
- **elementData[] ,** que és la taula de referències Object en què realment es desen els elements de l'objecte Vector.

En implementar la interfície `List`, la classe Vector hereta el mètode `iterator()`, que retorna una referència a la interfície `Iterator`, la qual permet fer un recorregut pels diferents elements de l'objecte Vector. Però la interfície `Iterator` va néixer juntament amb el *framework* de col·leccions quan la classe Vector ja existia. Per això, la classe Vector té el mètode `elements()`, que retorna una referència a la interfície `Enumeration` existent des de la versió 1.0 de Java, amb funcionalitat similar a la de la interfície `'Iterator`.

Exemple d'utilització de la classe Vector

Aquest exemple mostra el disseny de la classe `VectorValorsNumerics` pensada per definir vectors que desin valors numèrics (valors que implementin la classe `Number`) de manera que, a més de proporcionar els mètodes típics de la classe `Vector`, proporciona mètodes com el càlcul de la mitjana dels valors continguts, comparació de mitjana, capacitat i nombre d'elements vers altres vectors i implementació del mètode `toString()`.

```
1  //Fitxer VectorValorsNumerics.java
2  import java.util.Vector;
3  class VectorValorsNumerics extends Vector<Number>
4  {
5    public VectorValorsNumerics(int capacitatInicial, int increment
       )
6    {
7     super(capacitatInicial, increment);
8    }
9    public double mitja ()
10   {
11    double suma=0;
12    int valorsNoNuls=0;
13    for (int i=0; i<size(); i++)
14    {
15     suma = suma + get(i).doubleValue();
16     valorsNoNuls++;
17    }
18    if (valorsNoNuls==) return 0;
```

```
   else return suma / valorsNoNuls;
  }
  public boolean mateixaMitja (VectorValorsNumerics obj)
  {
   return mitja() == obj.mitja();
  }
  public boolean mateixNombreElements (VectorValorsNumerics obj)
  {
   return size() == obj.size();
  }
  public boolean mateixaCapacitat (VectorValorsNumerics obj)
  {
   return capacity() == obj.capacity();
  }
  public String toString()
  {
   String s="{";
   for (int i=0; i<size(); i++)
    if (s.equals("{")) s = s + get(i);
    else s = s + ", " + get(i);
   s = s + "}";
   return s;
  }
  public static void main (String args[])
  {
   Integer ti[]={1,2,3,4,5};
   Double td[]={1.1,2.2,3.3,4.4,5.5,6.6,7.7,8.8,9.9,10.10};
   VectorValorsNumerics vvn1 = new VectorValorsNumerics (5,2);
   VectorValorsNumerics vvn2 = new VectorValorsNumerics (5,3);
   int i;
   for (i=0; i<ti.length && i<td.length; i++)
   {  vvn1.add(ti[i]); vvn1.add(td[i]); vvn2.add(ti[i]); vvn2.add
       (td[i]);}
   for (; i<ti.length; i++)
   {  vvn1.add(ti[i]); vvn2.add(ti[i]);}
   for (; i<td.length; i++)
   {  vvn1.add(td[i]); vvn2.add(td[i]);}
   System.out.println("vvn1: "+vvn1);
   System.out.println("   Mitja       : "+vvn1.mitja());
   System.out.println("   Nre.Elements: "+vvn1.size());
   System.out.println("   Capacitat...: "+vvn1.capacity());
   System.out.println("vvn2: "+vvn2);
   System.out.println("   Mitja       : "+vvn2.mitja());
   System.out.println("   Nre.Elements: "+vvn2.size());
   System.out.println("   Capacitat...: "+vvn2.capacity());
   System.out.println("vvn1 i vvn2 tenen la mateixa mitja? " +
       vvn1.mateixaMitja(vvn2));
   System.out.println("vvn1 i vvn2 tenen el mateix nombre d'
       elements? " + vvn1.mateixNombreElements(vvn2));
   System.out.println("vvn1 i vvn2 tenen la mateixa capacitat? "
       + vvn1.mateixaCapacitat(vvn2));
  }
}
```

L'execució del mètode `main()` contingut en la classe dóna el resultat:

```
vvn1: {1, 1.1, 2, 2.2, 3, 3.3, 4, 4.4, 5, 5.5, 6.6, 7.7, 8.8,
    9.9, 10.1}
   Mitja  : 4.973333333333334
   Nre.Elements: 15
   Capacitat...: 15
vvn2: {1, 1.1, 2, 2.2, 3, 3.3, 4, 4.4, 5, 5.5, 6.6, 7.7, 8.8,
    9.9, 10.1}
   Mitja  : 4.973333333333334
   Nre.Elements: 15
   Capacitat...: 17
vvn1 i vvn2 tenen la mateixa mitja? true
```

```
vvn1 i vvn2 tenen el mateix nombre d'elements? true
vvn1 i vvn2 tenen la mateixa capacitat? false
```

Fixem-nos que, tot i que els dos vectors "vvn1" i "vvn2" s'han creat amb la mateixa capacitat inicial i s'han emplenat amb els mateixos valors, al final la capacitat de cadascun és diferent perquè s'han creat amb un increment diferent de capacitat.

De la classe històrica `Vector` es deriva una altra classe històrica, la classe `Stack`, pensada per a la gestió de piles. És recomanable fer-hi una ullada.

Classe històrica "Hashtable"

La classe `Hashtable<K, V>` deriva de la classe abstracta `Dictionary` i implementa la interfície `Map`, la interfície `Cloneable` per poder clonar objectes `Hashtable` amb el mètode `clone()`, i la interfície `Serializable` per poder convertir objectes `Hashtable` en cadenes de caràcters.

Un objecte `Hashtable` és una taula que relaciona una clau amb un element, utilitzant tècniques *Hash*, en què qualsevol objecte no `null` pot ser clau i/o element. La classe a què pertanyen les claus ha d'implementar els mètodes `hashCode()` i `equals()` (heretats de la classe Object) per tal de poder fer cerques i comparacions. El mètode `hashCode()` ha de retornar un enter únic i diferent per cada clau, que sempre és el mateix dins una execució del programa però que pot canviar en diferents execucions. A més, per dues claus que resultin iguals segons el mètode `equals()`, el mètode `hashCode()` ha de retornar el mateix valor enter.

Els objectes `Hashtable` estan dissenyats per mantenir una grup de parelles clau/element, de manera que permeten la inserció i la cerca

d'una manera molt eficient i sense cap tipus d'ordenació. Cada objecte `Hashtable` té dues dades membre: "capacity" i "loadFactor" (entre 0.0 i 1.0). Quan el nombre d'elements de l'objecte `Hashtable` supera el producte "capacity*loadFactor", l'objecte `Hashtable` creix cridant el mètode `rehash()`. Un "loadFactor" més gran apura més la memòria però és menys eficient en les cerques. És convenient partir d'una `Hashtable` suficientment gran per no estar ampliant contínuament.

Exemple d'utilització de la classe Hashtable

Aquest exemple mostra el disseny de la classe `HashtablePersona` per definir taules `Hashtable` de persones (segons el disseny de persones obtingut fins ara) considerant que el dni de la persona sigui la clau que permetrà accedir a les persones.

```
//Fitxer HashtablePersones.java

import java.util.Hashtable;
import java.util.Enumeration;

class HashtablePersones extends Hashtable<String,Persona>
{
   public static void main (String args[])
   {
      HashtablePersones htp = new HashtablePersones();
      Alumne a;
      Enumeration<Persona> ep;
      Enumeration<String> es;
      String dni;
```

```
15
16        a = new Alumne("99999999","Anna",20);
17        htp.put(a.getDni(), a);
18        a = new Alumne("00000000","Pep",33,'m');
19        htp.put(a.getDni(), a);
20        a = new Alumne("55555555","Teresa",30,'S');
21        htp.put(a.getDni(), a);
22        a = new Alumne("22222222","Maria",40,'s');
23        htp.put(a.getDni(), a);
24        a = new Alumne("66666666","Àngel",22);
25        htp.put(a.getDni(), a);
26        a = new Alumne("11111111","Joanna",25,'M');
27        htp.put(a.getDni(), a);
28        a = new Alumne("55555555","Maria Teresa",30,'S');
29        htp.put(a.getDni(), a);
30
31        System.out.println("Contingut de la Hashtable via
              enumeracions:");
32        ep = htp.elements();
33        es = htp.keys();
34        while (es.hasMoreElements())
35           System.out.println (es.nextElement() + " >>> " + ep.
                 nextElement());
36
37        System.out.println();
38        System.out.println("Contingut de la Hashtable cridant el mè
              tode toString:");
39        System.out.println(htp);
40
41        System.out.println();
42        System.out.println("Efectuem alguna recerca de persones per
               la clau \"dni\":");
43        dni = "66666666";
44        System.out.println("Dni : " + dni + " >>> " + htp.get(dni))
              ;
45        dni = "6666";
46        System.out.println("Dni : " + dni + " >>> " + htp.get(dni))
              ;
47     }
48  }
```

L'execució del mètode `main()` dóna el resultat:

```
1  Contingut de la Hashtable via enumeracions:
2  66666666 >>> Dni: 66666666 – Nom: Àngel – Edat: 22 – Nivell: ???
3  00000000 >>> Dni: 00000000 – Nom: Pep – Edat: 33 – Nivell: Cicle
       F. Mitjà
4  11111111 >>> Dni: 11111111 – Nom: Joanna – Edat: 25 – Nivell:
       Cicle F. Mitjà
5  22222222 >>> Dni: 22222222 – Nom: Maria – Edat: 40 – Nivell:
       Cicle F. Superior
6  99999999 >>> Dni: 99999999 – Nom: Anna – Edat: 20 – Nivell: ???
7  55555555 >>> Dni: 55555555 – Nom: Maria Teresa – Edat: 30 –
       Nivell: Cicle F. Superior
8
9  Contingut de la Hashtable cridant el mètode toString:
10 {66666666=Dni: 66666666 – Nom: Àngel – Edat: 22 – Nivell: ???,
       00000000=Dni: 00000000 – Nom: Pep
11 – Edat: 33 – Nivell: Cicle F. Mitjà, 11111111=Dni: 11111111 – Nom
       : Joanna – Edat: 25 – Nivell: Ci
12 cle F. Mitjà, 22222222=Dni: 22222222 – Nom: Maria – Edat: 40 –
       Nivell: Cicle F. Superior, 9999999
13 9=Dni: 99999999 – Nom: Anna – Edat: 20 – Nivell: ???, 55555555=
       Dni: 55555555 – Nom: Maria Teresa
14 – Edat: 30 – Nivell: Cicle F. Superior}
15
16 Efectuem alguna recerca de persones per la clau "dni":
```

```
17 Dni : 66666666 >>> Dni: 66666666 - Nom: Àngel - Edat: 22 - Nivell
       : ???
18 Dni : 6666 >>> null
```

En aquest exemple veiem el funcionament de diversos mètodes de la classe `Hashtable`:

- **El mètode put()** permet afegir diversos elements a la taula i els elements queden ordenats segons l'ordre en què el mètode put () s'executa. Fixem-nos que en el moment en què s'intenta afegir una parella clau/element per a la qual ja hi ha la clau en la taula, la nova parella substitueix la parella existent i passa a ocupar el darrer lloc de la taula. Aquesta situació té lloc en efectuar la inserció d'una segona persona amb dni "55555555", de manera que en les visualitzacions posteriors del contingut de la taula trobarem la persona anomenada Maria Teresa en la darrera posició.
- **El mètode elements()** proporciona una enumeració per recórrer els elements emmagatzemats en la taula, i el mètode keys()proporciona una enumeració per recórrer les claus emmagatzemades en la taula.
- **El mètode toString()**, sense necessitat d'implementar-lo, mostra el contingut de la taula en el format { clau=element, clau=element, ... }.
- **El mètode get(clau)** cerca l'element corresponent a la clau i retorna valor null si no el troba.

1.2.4 Algorismes

El *framework* de col·leccions de Java proporciona les classes `Collections` (acabada en *s*) i `Arrays`, que són una mica especials: no són abstractes, però no proporcionen cap constructor públic per crear objectes.

La classe `Arrays` és una classe que conté mètodes `static` destinats a gestionar (ordenar, omplir, realitzar cerques i comparar) les taules clàssiques de Java. També permet veure les taules clàssiques de Java com a objectes que implementen la interfície `List`.

La classe `Collections` és una classe que conté mètodes i constants `static` destinats a gestionar els objectes de les classes que implementen les interfícies `Collection` i `Map` i les subinterfícies corresponents.

La gran quantitat de mètodes que hi ha en ambdues classes (105 en la classe `Arrays` i 169 en la classe `Collections`) en fa impossible la simple enumeració en aquest material. És aconsellable fer una ullada a la documentació corresponent de Java.

La taula 1.2 presenta un resum d'alguns dels mètodes que ens podem trobar en les classes `Arrays` i `Collections` tenint en compte que, per als diferents mètodes, hi pot haver diverses versions (polimorfisme).

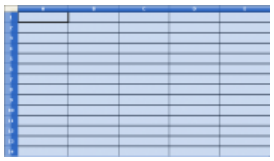
TAULA 1.2. Resum de mètodes d' "Arrays" i "Collections".

Mètode	Objectiu
Arrays.sort(T[])	Ordenar taules clàssiques de Java
Arrays.toString(T[])	Obtenir una representació `String` dels continguts d'una taula
Arrays.fill(T[],...)	Emplenar totes les cel·les d'una taula amb valors determinats
Arrays.equals(T[], T[],...)	Comparar dues taules
Arrays.binarySearch(T[], T)	Efectuar una cerca dicotòmica en una taula ordenada
Collections.sort(List<T>)	Ordenar objectes `List`
Collections.binaySearch(List<T>, Object)	Efectuar una cerca dicotòmica en un objecte `List` ordenat
Collections.reverse(List<T>)	Invertir l'ordre dels elements en un objecte `List`
Collections.max(Collection<E>)	Trobar el màxim valor d'un objecte `Collection`
Collections.min(Collection<E>)	Trobar el mínim valor d'un objecte `Collection`

1.3 "Arrays" multidimensionals

L'ordre dels índexs és important.

Quan es declara un *array*, hi ha una propietat especial, a part del seu identificador de tipus i de mida. Es tracta de la seva **dimensió**, el nombre d'índexs que cal usar per establir la posició que es vol tractar. Res no impedeix que el nombre d'índexs sigui més gran que 1.



Un "array" de dimensió 2: una taula

Un **array multidimensional** és aquell en què per accedir a una posició concreta, en lloc d'usar un sol valor com a índex, s'usa una seqüència de diversos valors. Cada índex serveix com a coordenada per a una dimensió diferent.

Entre els *arrays* multidimensionals, el cas més usat és el dels *arrays* bidimensionals, de dues dimensions. Aquest és un dels més fàcils de visualitzar i és, per tant, un bon punt de partida. Com que es tracta d'un tipus d'*array* en què calen dos índexs per establir una posició, es pot considerar que el seu comportament és com el d'una matriu, taula o full de càlcul. El primer índex indicaria la fila, mentre que el segon indicaria la columna on es troba la posició a què es vol accedir.

Per exemple, suposeu que un programa per tractar notes d'estudiants ara ha de gestionar les notes individuals de cada estudiant per a un seguit d'exercicis, de manera que es poden fer càlculs addicionals: calcular la nota final segons els resultats individuals, veure l'evolució de cada estudiant al llarg del curs... En aquest cas, tota aquesta informació es pot resumir de manera eficient en una taula o en un full de càlcul, en què els estudiants es poden organitzar per files, on cada columna correspon al valor d'una prova, i la darrera columna pot ser la nota final.

1.3.1 Declaració d'"arrays" bidimensionals

La sintaxi per declarar i inicialitzar un *array* de dues dimensions és semblant a la dels unidimensionals, si bé ara cal especificar que hi ha un índex addicional. Això es fa usant un parell de claus extra, `[]`, sempre que calgui especificar un nou índex.

Per declarar-ne un d'inicialitzat amb totes les posicions dels seus valors per defecte, caldria usar la sintaxi:

```
1 paraulaClauTipus[] identificadorVariable = new paraulaClauTipus[nombreFiles][
      nombreColumnes];
```

Com passava amb els *arrays* unidireccionals, el valor dels índexs, de les files i de les columnes, no ha de ser necessàriament un de literal. Pot ser definit d'acord amb el contingut d'una variable, diferent per a cada execució del programa. El valor del nombre de files i de columnes tampoc no ha de ser necessàriament el mateix.

La inicialització mitjançant valors concrets implica indicar tants valors com el nombre de files per columnes. Per fer-ho, primer s'enumeren els valors individuals que hi ha a cada fila de la mateixa manera que s'inicialitza un *array* unidimensional: una seqüència de valors separats per comes i envoltats per claudàtors, `{...}`. Un cop enumerades les files d'aquesta manera, aquestes s'enumeren al seu torn separades per comes i envoltades per claudàtors. Conceptualment, és com declarar un *array* de files, en què cada posició té un altre *array* de valors. Això es veu més clar amb la sintaxi, que és la següent:

```
1 paraulaClauTipus[][] identificadorVariable = {
2                         {Fila0valor1, Fila0valor2, ... , Fila0valorN},
3                         {Fila1valor1, Fila1valor2, ... , Fila1valorN},
4                         ...,
5                         {FilaNvalor1, FilaNvalor2, ... , FilaNvalorN}
6                         };
```

Fixeu-vos com a l'inici i al final hi ha els claudàtors extra que tanquen la seqüència de files i després de cada fila hi ha una coma, per separar-les entre si. Per fer més clara la lectura, s'han usat diferents línies de text per indicar els valors emmagatzemats a cada fila, però res no impedeix escriure-ho tot en una sola línia molt llarga. De tota manera, és recomanable usar aquest format i intentar deixar espais on pertoqui de manera que els valors quedin alineats verticalment, ja que això facilita la comprensió del codi font.

Per exemple, la manera de declarar i d'inicialitzar amb valors concrets un *array* bidimensional de tres files per cinc columnes de valors de tipus enter seria:

```
1 int[][] arrayBidi = {
2                     {1 ,2 ,3 ,4 ,5 },
3                     {6 ,7 ,8 ,9 ,10},
4                     {11,12,13,14,15}
5                     };
```

Des del punt de vista d'una taula o matriu, la primera fila seria {1, 2, 3, 4, 5}, la segona {6 ,7 ,8 ,9 ,10} i la tercera {11,12,13,14,15}, tal com mostra la taula 1.3.

TAULA 1.3. Resultat d'inicialitzar amb valors concrets un array bidimensional de tres files per cinc columnes.

1	2	3	4	5
6	7	8	9	10
11	12	13	14	15

En usar aquest tipus d'inicialització cal assegurar-se que les mides de totes les files siguin exactament iguals (que tinguin el mateix nombre de valors separats entre comes), ja que cal garantir que totes les files tenen el mateix nombre de columnes.

1.3.2 Esquemes d'ús d'"arrays" bidimensionals

L'accés a les posicions d'un *array* bidimensional és pràcticament idèntic a l'unidireccional, però tenint en compte que ara hi ha dos índexs per preveure referents a les dues coordenades.

```
1 identificadorVariable[índexFila][índexColumna]
```

La taula 1.4 fa un resum de com s'accediria a les diferents posicions d'un *array* de 4 * 5 posicions. En qualsevol cas, en accedir-hi, sempre cal mantenir present que cada índex mai no pot ser igual o superior al nombre de files o de columnes, segons correspongui. En cas contrari, es produeix un error.

TAULA 1.4. Array bidimensional de 4 * 5 posicions.

a[0][0]	a[0][1]	a[0][2]	a[0][3]	a[0][4]
a[1][0]	a[1][1]	a[1][2]	a[1][3]	a[1][4]
a[2][0]	a[2][1]	a[2][2]	a[2][3]	a[2][4]
a[3][0]	a[3][1]	a[3][2]	a[3][3]	a[3][4]

Assegureu-vos d'usar variables diferents per indicar cada índex.

Com que ara hi ha dos índexs, tots els esquemes fonamentals per treballar amb *arrays* bidimensionals es basen en dues estructures de repetició imbricades, una per tractar cada fila i l'altra per tractar cada valor individual dins la fila.

En el cas d'*arrays* bidimensionals, l'atribut `length` té un comportament una mica especial, ja que ara hi ha dos índexs. Per tal entendre'l, cal recordar que per inicialitzar l'*array* el que es fa és enumerar una llista de files, dins les quals hi ha els valors emmagatzemats realment. Per tant, el que diu l'atribut és el nombre de files que hi ha. Per saber el nombre de valors d'una fila heu de fer:

```
1 identificadorVariable[índexFila].length
```

Per exemple, si la taula 1.7 correspon a una variable anomenada `arrayBidi`, `arrayBidi.length` avalua a 4 (hi ha quatre files) i `arrayBidi[0].length`, `arrayBidi[1].length`, etc. tots avaluen a 5 (hi ha 5 columnes).

Inicialització procedural

Novament, es pot donar el cas en què vulgueu assignar a cada posició valors que cal calcular prèviament i que, per tant, calgui anar assignant un per un a totes les posicions de l'*array* bidimensional.

A mode d'exemple, compileu i executeu el codi font següent, que emmagatzema a cada posició la suma dels índexs d'un *array* bidimensional de dimensions arbitràries i després es visualitza el resultat per pantalla, ordenat per files.

```
1  import java.util.Scanner;
2
3  //Un programa que inicialitza un array bidimensional.
4  public class InicialitzacioBidi {
5
6    public static void main (String[]  args) {
7
8      Scanner lector = new Scanner(System.in);
9
10     //Llegeix les files.
11     int nombreFiles = 0;
12     while (nombreFiles <= 0 ) {
13       System.out.print("Quantes files tindrà la taula? " );
14       if (lector.hasNextInt() ) {
15         nombreFiles = lector.nextInt();
16       } else {
17         lector.next();
18         System.out.print("Aquest valor no és correcte. " );
19       }
20     }
21     lector.nextLine();
22
23     //Llegeix les columnes.
24     int nombreColumnes = 0;
25     while (nombreColumnes <= 0) {
26       System.out.print("Quantes files tindrà la taula? " );
27       if (lector.hasNextInt() ) {
28         nombreColumnes = lector.nextInt();
29       } else {
30         lector.next();
31         System.out.print("Aquest valor no és correcte. ");
32       }
33     }
34     lector.nextLine();
35
36     //Inicialització amb valors per defecte.
37     int[][] arrayBidi = new int[nombreFiles][nombreColumnes];
38
39     //Observeu l'ús de l'atribut "length".
40     System.out.println("Hi ha " + arrayBidi.length + " files." );
41     System.out.println("Hi ha " + arrayBidi[0].length + "columnes." );
42
43     //Bucle per recórrer cada fila.
44     //"i" indica el número de fila.
45     for(int i = 0; i < nombreFiles; i++) {
46
47       //Bucle per recórrer cada posició dins la fila (columnes de la fila).
48       //"j" indica el número de fila.
49       for (int j = 0; j < nombreColumnes; j++)  {
50
51         //Valor assignat a la posició: suma dels índex de fila i columna.
52         arrayBidi[i][j] = i + j;
53       }
54     }
55     //Es visualitza el resultat, també calen dos bucles.
56     for(int i = 0; i < nombreFiles; i++) {
57       //Inici de fila, obrim claudàtors.
```

```
      System.out.print("Fila " + i + " { ");
      for (int j = 0; j < nombreColumnes; j++ )  {
        System.out.print(arrayBidi[i][j] + " ");
      }
      //Al final de cada fila es tanquen claudàtors i es fa un salt de línia.
      System.out.println("}" );
    }

  }
}
```

Recorregut

En el cas d'*arrays* bidimensionals, també pot passar que sigui necessari efectuar càlculs fent un recorregut per tots els valors continguts. Novament, cal anar fila per fila, mirant totes les posicions de cadascuna.

Per exemple, suposeu que hi ha un programa en què es desa a cada fila el valor de les notes dels estudiants d'un curs. Es demana calcular la nota final de cada estudiant i emmagatzemar-la a la darrera columna, i també saber la mitjana de totes les notes finals. El codi es mostra tot seguit; analitzeu-lo i proveu que funciona. En aquest cas, per facilitar-ne l'execució, els valors de les notes estan inicialitzats per a valors concrets. En el cas de la nota final de cada estudiant, inicialment es deixa a 0 i serà el programa qui la calculi.

Ara sí: observeu atentament l'ús de l'atribut `length` per controlar els índexs en recórrer l'*array* i saber quin és el nombre de valors en una fila o el nombre de files.

```
//Un programa que calcula notes mitjanes en un array bidimensional.
public class RecorregutBidi {
  public static void main (String[] args) {
    //Dades de les notes.
    float[][] arrayBidiNotes = {
                                { 4.5f, 6f  , 5f  , 8f  , 0f  },
                                { 10f , 8f  , 7.5f, 9.5f, 0f  },
                                { 3f  , 2.5f, 4.5f, 6f  , 0f  },
                                { 6f  , 8.5f, 6f  , 4f  , 0f  },
                                { 9f  , 7.5f, 7f  , 8f  , 0f  }
                                };
    //Mitjana aritmètica del curs per a tots els estudiants.
    float sumaFinals = 0f;

//Es fa tractant fila per fila, indicada per "i". Cada fila és un estudiant.
    //"arrayBidiNotes.length" avalua el nombre de files.
    for(int i = 0; i < arrayBidiNotes.length; i++) {
      //Aquí s'acumulen les notes de l'estudiant tractat.
      float sumaNotes = 0f;

      //Tractem cada fila (cada estudiant). Cada nota la indexa "j".
      //"arrayBidiNotes[i].length avalua el nombre de posicions a la fila.
      for(int j = 0; j < arrayBidiNotes[i].length; j++) {
          //Estem a la darrera posició de la fila?
        if(j != (arrayBidiNotes[i].length - 1) ) {
          //Si no és la darrera posició, anem acumulant valors.
          sumaNotes = sumaNotes + arrayBidiNotes[i][j];
        } else {
          //Si ho és, cal escriure la mitjana.
          //Hi ha tantes notes com la mida d'una fila - 1.
          float notaFinal =  sumaNotes/(arrayBidiNotes[i].length - 1);
          arrayBidiNotes[i][j] = notaFinal;
          System.out.println("L'estudiant " + i + " ha tret " + notaFinal + "."
              );
```

```
34            //S'actualitza la suma de mitjanes de tots els estudiants.
35            sumaFinals = sumaFinals + notaFinal;
36          }
37        }
38        //Fi del tractament d'una fila.
39
40      }
41      //Fi del tractament de totes les files.
42      //Es calcula la mitjana: suma de notes finals dividit entre nombre d'
            estudiants.
43      float mitjanaFinal = sumaFinals / arrayBidiNotes.length;
44      System.out.println("La nota mitjana del curs és " + mitjanaFinal);
45    }
46  }
```

Cerca

En *arrays* bidimensionals la cerca també implica haver de gestionar dos índexs per anar avançant per files i per columnes. Ara bé, aquest cas és especialment interessant, ja que en assolir l'objectiu cal sortir de les dues estructures de repetició imbricades. Per tant, no es pot usar simplement una sentència `break`, ja que només serveix per sortir d'un únic bucle. Cal tenir un semàfor que s'avaluï en tots dos bucles.

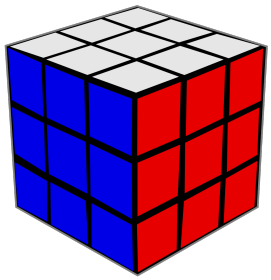
Per exemple, suposeu que el programa de gestió de notes vol cercar si algun estudiant ha tret un 0 en algun dels exercicis al llarg del curs. Caldrà anar mirant fila per fila totes les notes de cada estudiant, i quan es trobi un 0, acabar immediatament la cerca. El codi per fer-ho, basant-se exclusivament en un semàfor que diu si ja s'ha trobat el valor, seria el següent. Proveu-lo al vostre entorn de treball.

En un cas com aquest, cal anar amb molt de compte a inicialitzar i incrementar correctament l'índex per recórrer la posició de cada fila (és a dir, que mira cada columna), ja que en usar una sentència `while` en lloc de `for` no es fa automàticament. Si us despisteu i no ho feu, el valor serà incorrecte per a successives iteracions del bucle que recorre les files i el programa no actuarà correctament.

```
1   //Un programa que cerca un 0 entre els valors d'un array bidimensional.
2   public class CercaBidi {
3     public static void main (String[] args) {
4       //Dades de les notes.
5       float[][] arrayBidiNotes = {
6                                  { 4.5f, 6f  , 5f  , 8f  },
7                                  { 10f , 8f  , 7.5f, 9.5f},
8                                  { 3f  , 2.5f, 0f  , 6f  },
9                                  { 6f  , 8.5f, 6f  , 4f  },
10                                 { 9f  , 7.5f, 7f  , 8f  }
11                                 };
12
13      //Mirarem quin estudiant ha tret el 0.
14      //Inicialitzem a un valor invàlid (de moment, cap estudiant té un 0)
15      //Aquest valor fa de semàfor. Si pren un valor diferent, cal acabar la
            cerca.
16      int estudiant = -1;
17
18      //"i indica la fila.
19      int i =0;
20
21      //Es va fila per fila.
```

```
22      //S'acaba si s'ha trobat un 0 o si ja s'ha cercat a totes les files.
23      while ((estudiant == -1)&&(i < arrayBidiNotes.length) ){
24        //"j indica la "columna".
25        int j =0;
26
27        //Se cerca en una fila.
28        //S'acaba si s'ha trobat un 0 o si ja s'ha cercat a totes les posicions.
29        while ((estudiant == -1)&&(j < arrayBidiNotes[i].length) ){
30          //Aquesta nota és un 0?
31          if  (arrayBidiNotes[i][j] == 0f) {
32            //L'índex que diu l'estudiant és el de la fila.
33            estudiant = i;
34          }
35          //S'avança a la posició següent dins de la fila.
36          j++;
37        }
38        //Fi del tractament d'una fila.
39        //S'avança a la fila següent.
40        i++;
41      }
42      //Fi del tractament de totes les files.
43
44  if (estudiant == -1) {
45        System.out.println("Cap estudiant no té un 0.");
46      } else {
47        System.out.println("L'estudiant " + estudiant + " té un 0.");
48      }
49    }
50  }
```

1.3.3 "Arrays" de més de dues dimensions

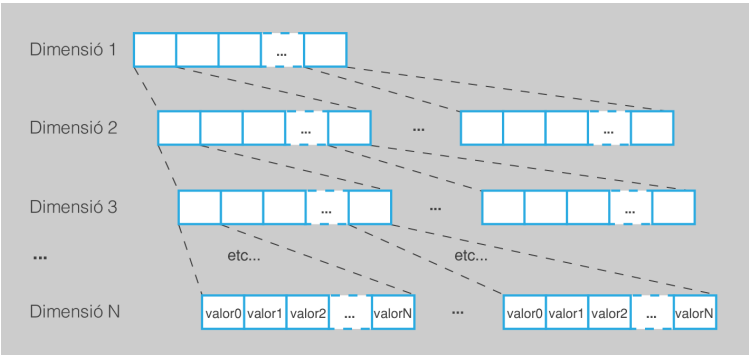


Un "array" amb tres dimensions: el cub de Rubik

El cas dels *arrays* bidimensionals és el més comú, però res no impedeix que el nombre d'índexs necessaris per situar una posició sigui també més gran de dos, d'un valor arbitrari. Un *array* de tres dimensions encara es pot visualitzar com una estructura tridimensional en forma de cub, però dimensions superiors ja requereixen més grau d'abstracció, i per això només es presentaran, sense entrar en més detall.

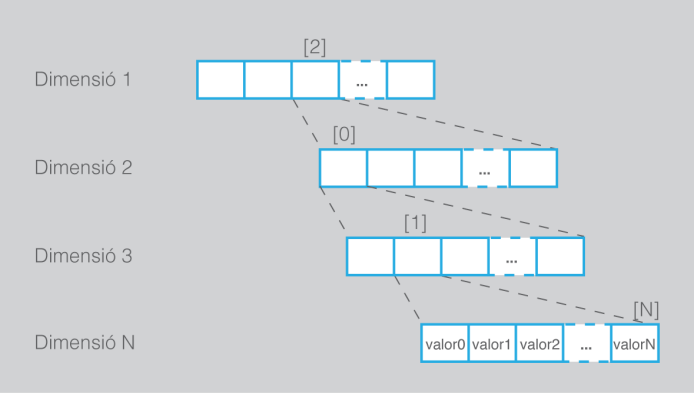
Conceptualment, es pot considerar que defineixen una estructura d'*arrays* imbricats a diferents nivells com el que es mostra a la figura 1.5. Cada nivell correspon a una dimensió diferent.

FIGURA 1.5. Un "array" multidimensional: "arrays" dins "arrays" dins "arrays"...



Cada índex individual identifica una casella de l'*array* triat d'una dimensió determinada, de manera que el conjunt d'índexs serveix com a coordenada per establir el valor final (que ja no serà un altre *array*) al qual es vol accedir. La figura 1.6 en mostra un exemple de quatre dimensions.

FIGURA 1.6. Accés a un "array" de quatre dimensions (calen 4 índexs)



Tot i que aquesta estructura sembla força complexa, no és gaire diferent d'una estructura de carpetes i de fitxers, en què les carpetes prenen el paper dels *arrays* i els fitxers, el de les dades que es volen desar. Podeu tenir una única carpeta per emmagatzemar totes les fotos, que seria el cas d'un únic nivell d'*array* (o sigui, els *arrays* tal com s'han vist fins ara). Però també les podríeu organitzar en una carpeta que al seu torn conté altres carpetes, dins les quals es desen diferents fotos. Això seria equivalent a disposar de dos nivells d'*arrays*. I així podríeu anar creant una estructura de carpetes amb tants nivells com volguéssiu. Per triar un fitxer n'hi ha prou a saber en quin ordre cal anar obrint les carpetes fins arribar a la carpeta final, on ja només us cal l'ordre del fitxer.

Estenem les restriccions que compleixen els *arrays* bidireccionals al cas dels *arrays* multidimensionals:

- En una mateixa dimensió, tots els *arrays* tindran **exactament** la mateixa mida.
- El tipus de dada que es desa a les posicions de la darrera dimensió serà el mateix per a tots. Per exemple, no es poden tenir reals i enters barrejats.

En llenguatge Java, per afegir noves dimensions per sobre de la segona, cal anar afegint claus `[]` addicionals a la declaració, una per a cada dimensió.

```
1 paraulaClauTipus[]...[] identificadorVariable = new paraulaClauTipus[midaDim1
      ]...[midaDimN];
```

Per exemple, per declarar amb valors per defecte un *array* de quatre dimensions, la primera de mida 5, la segona de 10, la tercera de 4 i la quarta de 20, caldria la instrucció:

```
1 ...
2   int[][][][] arrayQuatreDimensions = new int[5][10][4][20];
3   ...
```

L'accés seria anàleg als *arrays* bidimensionals, però usant quatre índexs envoltats per parells de claus.

```
...
  int valor = arrayQuatreDimensions[2][0][1][20];
  ...
```

En qualsevol cas, es tracta d'estructures complexes que no se solen usar, excepte en casos molt específics.

1.4 Tractament de fitxers XML

Una de les problemàtiques més importants amb què haureu de tractar quan utilitzeu fitxers per desar o recuperar les dades dels vostres programes és escollir de quina manera s'estructuren dins el fitxer. En quin format i en quin ordre estan escrites les dades, de manera que sempre que es llegeixin o s'escriguin és imprescindible seguir exactament aquest format. En cas contrari, en el millor dels casos, es produirà un error, i en el pitjor, les dades llegides o escrites seran totalment incorrectes. Això és un problema ja que, a menys que es disposi de documentació detallada, és molt difícil que, només amb un fitxer donat, es pugui esbrinar amb detall quina mena de dades conté. Per intentar pal·liar aquest problema, va sorgir el llenguatge XML.

Aquests materials han estan generats usant un llenguatge de marcat.

XML són les inicials en anglès d'**eXtensible Markup Language** o llenguatge de marques extensible.

Un llenguatge de marques és un conjunt de paraules i de símbols per descriure la identitat o la funció dels components d'un document (per exemple, "aquest és un paràgraf", "aquesta és una llista", "això és el títol d'aquesta figura"...). Aquestes marques permeten facilitar el processament de les dades a tractar dins el fitxer. Per exemple, transformar el document a diferents formats de la sortida de pantalla, mitjançant un full d'estil.

Alguns llenguatges de marques (especialment els utilitzats en processadors de text) només descriuen les aparences al seu lloc ("això és cursiva", "això és negreta"...), de manera que aquests sistemes només es poden utilitzar per a la visualització, i per poca cosa més. Un dels llenguatges més populars de marcat, d'acord a aquesta definició, és HTML, el llenguatge emprat per codificar pàgines web. XML permet dur aquesta idea més enllà de la visualització i facilitar la comprensió de qualsevol document de dades en els vostres programes, independentment de la seva funció, no només per indicar com s'han de representar visualment.

Per veure clarament aquest fet, el millor és mostrar ja directament un exemple d'un mateix conjunt de dades relatives a un objecte estructurades sense XML i amb XML, sense ni tan sols haver presentat abans com s'estructura un document XML. D'aquesta manera, sense cap coneixement previ, encara queden més patents els

seus objectius i quins avantatges presenta. En el primer cas, s'ha triat representar els valors de l'objecte com a text separat per comes:

```
Sense XML:
  Producte 1, 1, 10.0, 4, true
Amb XML:
<Producte id="1" aLaVenda="true">
  <Nom>Producte 1</Nom>
  <Preu>10.0</Preu>
  <Estoc>4</Estoc>
</Producte>
```

Atès el primer cas, sense tenir cap documentació prèvia, ¿és possible esbrinar a què es refereixen exactament els valors "4" o "1"? Quin és el nom de la classe a què pertany l'objecte? La veritat és que caldria tenir molta imaginació, per no dir directament que és impossible. XML es basa en la idea que a cada dada dins el document se li afegeix una informació extra (les marques) que, si s'ha triat estratègicament, permet saber exactament el seu propòsit. El document generat és autoexplicatiu.

Com es pot veure de l'exemple anterior, XML i HTML són molt semblants. Si mai heu vist el codi font d'una pàgina web, XML us pot resultar familiar, encara que sigui el primer cop que veieu un document XML. Això no és casualitat, ja que ambdós són variacions d'un llenguatge de marcat més genèric anomenat SGML (*Standard Generalized Markup Language*, o llenguatge de marques estàndard generalitzat). Ara bé, entre HTML i XML hi ha algunes diferències importants. Entre les més destacades hi ha les següents:

- HTML serveix únicament per indicar com representar visualment unes dades (normalment a navegadors web). XML serveix per a qualsevol tipus de processament de dades.
- En HTML la llista de marques possibles ja està prefixada. En XML no; vosaltres us podeu inventar les que vulgueu.
- HTML sol tenir una sintaxi més laxa, on alguns errors en el format de les marques no afecten el seu processament. XML és molt més estricte.

1.4.1 Estructura d'un document XML

Un document XML emmagatzema un conjunt de dades d'acord a un seguit de marques que permeten identificar de manera senzilla el propòsit d'aquestes dades. Ara bé, les marques són definides pel seu creador, d'acord a la mena d'informació que vol emmagatzemar. Cada aplicació usarà documents XML amb marques diferents segons el que li resulti més convenient. Per tant, serà vostra la tasca de definir-les, usant noms adients que permetin identificar amb claredat la funció de les dades emmagatzemades.

Tot i que documents d'aplicacions diferents, com és d'esperar, tindran un format diferent, la seva estructura serà semblant, ja que tots han de seguir l'estructura i les regles bàsiques que marca el llenguatge XML. Des d'un punt de vista genèric,

tot document XML es correspon a una estructura jeràrquica (en forma d'arbre), on es parteix d'un node arrel i d'aquest sorgeixen nodes fills que es van propagant fins arribar a nodes fulla, o els darrers dins la jerarquia. Els tipus de nodes més importants són els següents:

- De text, que contenen la informació final a emmagatzemar. La majoria de nodes fulla solen ser d'aquest tipus. Es representen dins el document directament usant una cadena de text que representa les dades en qüestió.
- Elements, que defineixen agrupacions lògiques de continguts caracteritzades per un nom. Es representen usant una parella de marques d'inici i de fi, amb format `<NomElement>...</NomElement>`.
- Atributs, parelles de nom i de valor, sempre associades a un element concret. S'escriuen dins la marca d'inici de l'element, usant el format `nom=''valor''`. Si, en un element, n'hi ha més d'un, se separen amb un espai en blanc.
- Sovint, a l'inici del document, s'hi afegeix pròleg, en realitat opcional, que dóna certa informació sobre el format del fitxer (versió, codificació...).

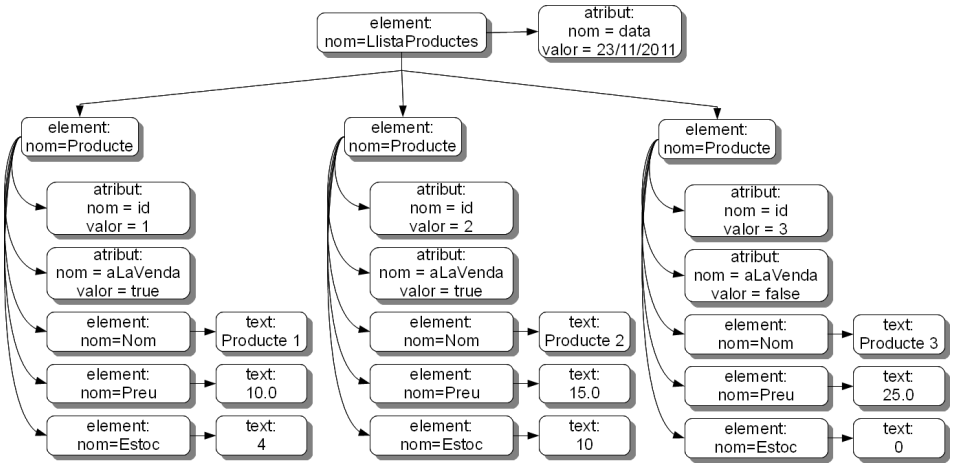
Els nodes d'un document XML han de seguir unes normes en la seva estructura perquè es consideri que el document és correcte (o ben format). Primer de tot, només hi pot haver un únic element arrel. D'altra banda, atès qualsevol element, dins seu, entre el símbol d'inici i final, només hi poden haver, o altres elements complets, ja sigui un o molts, o text. De fet, aquesta inclusió és el que defineix la relació jeràrquica entre elements (pare = qui inclou, fills = els que estan inclosos). Finalment, les marques d'inici i de final dels diferents elements sempre han d'estar correctament niuades. Per exemple, el següent cas seria incorrecte d'acord a aquesta darrera norma:

```
<Producte><Nom></Producte></Nom>
```

Bàsicament, les normes es poden resumir així: que tots els elements han de conformar sempre una estructura jeràrquica correcta. La figura 1.7 mostra la representació jeràrquica del document XML que es mostra tot seguit.

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<LlistaProductes data="23/11/2011">
<Producte id="1" aLaVenda="true">
  <Nom>Producte 1</Nom>
  <Preu>10.0</Preu>
  <Estoc>4</Estoc>
</Producte>
<Producte id="2" aLaVenda="true">
  <Nom>Producte 2</Nom>
  <Preu>15.0</Preu>
  <Estoc>10</Estoc>
</Producte>
<Producte id="3" aLaVenda="false">
  <Nom>Producte 3</Nom>
  <Preu>25.0</Preu>
  <Estoc>0</Estoc>
</Producte>
</LlistaProductes>
```

FIGURA 1.7. Estructura jeràrquica dels nodes d'un document XML



Algú una mica observador pot adonar-se que, tot plegat, és molt semblant a un mapa d'objectes. I és que XML i orientació a objectes són dos temes que, tot i que en realitat de manera casual, semblen fets l'un per l'altre.

1.4.2 Lectura de fitxers XML

Atès que l'estructura i la sintaxi d'un document XML és sempre la mateixa exactament, només varia el nom i els valors dels nodes, és possible dissenyar mecanismes genèrics per llegir-lo. El codi no ha de ser *ad hoc* depenent de cada tipus d'aplicació. Això és un gran avantatge a l'hora de fer programes.

Existeixen diferents mètodes per fer el tractament de documents XML. Aquest apartat se centra només en l'anomenat DOM (*Document Object Model*, o model d'objectes de document), que sol ser el més popular i senzill d'usar quan la quantitat de dades contingudes al document és moderada. Per tractar documents molt grans és preferible usar altres sistemes, com per exemple l'anomenat SAX (*Simple API for XML*, o API simple per a XML).

La senzillesa del DOM recau en el fet que es basa a llegir qualsevol document XML i processar-lo per convertir-lo en una estructura d'objectes a memòria, el model DOM, idèntica a l'estructura del document original, exactament tal com s'ha presentat. Diferents objectes de tipus `Node` estan entrellaçats entre si, contenint un seguit de mètodes que permeten consultar els seus noms, valors o altres nodes dins l'estructura, de manera que és possible fer-hi recorreguts. El procés de creació d'aquest model sempre és igual independentment del contingut del fitxer XML, pel que es pot generalitzar. Ara bé, un cop es disposa del model, ja és tasca vostra fer recorreguts i indicar què cal cercar, ja que això dependrà de cada aplicació concreta.

Creació i invocació del parser DOM Java

Dins les darreres versions, Java ja incorpora un seguit de classes auxiliars que permeten processar un document XML automàticament usant el model DOM. No és necessari que vosaltres tracteu el text que hi ha dins un fitxer XML. El resultat d'aquest procés proporciona una estructura d'objectes d'acord al model DOM, que sí que haureu de recórrer i analitzar per dur a terme la tasca encomanada. Normalment, extreure'n certa informació.

Per poder començar a treballar, per tant, primer cal llegir el fitxer i processar-lo, mitjançant un processador XML. Dins el *package* `javax.xml.parsers` es poden trobar les classes que permeten dur a terme aquesta tasca. Per al cas del model DOM, la classe que us interessa és `DocumentBuilder` (constructor de documents). Aquesta classe té la particularitat, però, que no es pot instanciar directament, sinó que cal emprar una classe auxiliar anomenada `DocumentBuilderFactory` per obtenir-ne objectes.

Un cop conegudes les classes que hi intervenen, el codi per poder obtenir un processador i llegir un fitxer XML sempre ve a ser el mateix, que es mostra tot seguit:

```
1  import javax.xml.parsers.*;
2  import org.w3c.dom.*;
3
4    //...
5
6    DocumentBuilderFactory builderFactory = DocumentBuilderFactory.newInstance();
7    Document document = null;
8    try {
9      DocumentBuilder  builder = builderFactory.newDocumentBuilder();
10     File f = new File ("elMeuFitxer.xml");
11     document = builder.parse(f);
12   } catch (Exception ex) {
13     System.out.println("Error en la lectura del document: " + ex);
14   }
```

Estrictament, `Document` és una interfície. La instància creada pertany a la classe `Node`.

A l'hora de llegir un document XML, és molt important controlar excepcions, ja que a part dels possibles errors que poden succeir en accedir al sistema de fitxers (el fitxer no existeix, hi ha un error al sistema d'entrada/sortida, el fitxer està protegit...), també cal afegir aquells vinculats a la sintaxi del mateix document. Si el document no és un fitxer XML o no en segueix de manera estricta la sintaxi, també es produeix una excepció i se n'interromp el processament.

Si no ha succeït cap error, a la variable "document" es disposa d'un objecte de la classe `org.w3c.dom.Document`, a partir del qual es pot accedir a tota l'estructura d'objectes (nodes i atributs) que conformen les diferents parts d'un document XML.

Objectes dins el model DOM Java

L'objecte `Document` representa un document complet XML. Ara bé, per analitzar el contingut del fitxer, s'ha d'obtenir el contingut dels seus nodes. Cal tenir en compte que tot objecte DOM pot contenir una gran quantitat de diferents nodes

connectats en una estructura d'arbre. De totes maneres, el primer pas sempre és l'obtenció de l'element arrel, a partir del qual es poden iniciar els recorreguts:

```
1 Element arrel = document.getDocumentElement();
```

A la documentació oficial de Java podeu trobar la llista detallada de tots els mètodes disponibles als elements DOM.

Fixeu-vos que aquesta crida retorna un objecte de tipus `Element` (un element del document XML). Sobre un element es poden cridar diversos mètodes. Els més típics són:

- `String getTagName()`, consulta el nom de l'element.
- `String getAttribute(String name)`, donat un nom d'un atribut, si aquest existeix a l'element, retorna el seu valor.
- `NodeList getElementsByTagName(String name)`, proporciona una llista de tots els elements descendents (fills, néts...), partint des d'aquest element, donat nom concret. Es pot usar "*" com a comodí per cercar-los tots. Si no n'hi ha cap, la llista és buida.

La llista completa de constants amb tipus de node la trobareu a la documentació de la classe `Node`

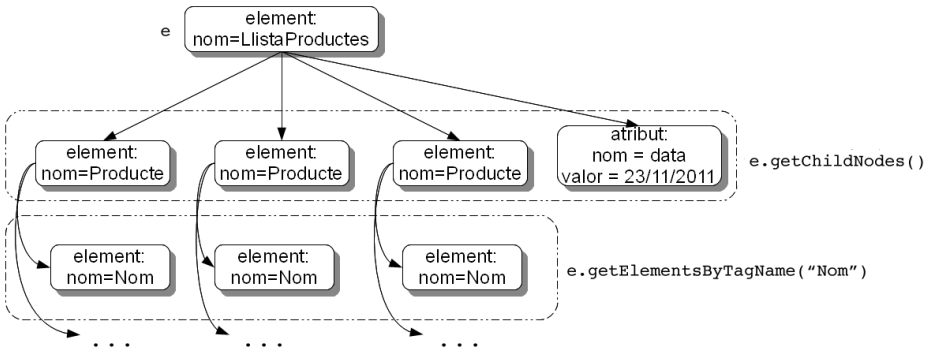
A l'hora de treballar amb un model DOM és important anar amb compte amb la jerarquia d'herència de les diferents classes que el conformen, ja que si bé tots els nodes de l'arbre són objectes `Node` (superclasse), només alguns són `Element` (subclasse). Els nodes que són atributs o text no pertanyen a aquesta classe.

En el cas de la classe `Node`, alguns mètodes interessants per manipular-los són:

- `getNodeType()`, retorna el tipus de node (element, text, atribut...). La classe `Node` disposa de tot un seguit de constants amb les quals es pot comparar el resultat d'invocar aquest mètode per discriminar un node per tipus. Per exemple, `Node.ATTRIBUTE_NODE` indica que és un node del tipus atribut.
- `NodeList getChildNodes(String name)`, proporciona una llista de nodes fill d'aquest element amb un nom concret. Si no n'hi ha cap, la llista és buida. Cal anar amb molt de compte amb aquest mètode, ja que la llista és de nodes, no d'elements, per la qual cosa també pot incloure atributs i nodes de text.

Recordeu que `Element` hereta de `Node` i, per tant, tots aquests mètodes també es poden invocar des d'un objecte d'aquest tipus. També val la pena fer èmfasi en una diferència important entre `getChildNodes` i `getElementsByTagName` que sovint es passa per alt. El primer només consulta els nodes fills immediatament inferiors, mentre que el segon recorre tot el subarbre cercant elements amb un nom concret, no només els fills. Aquest comportament s'esquematitza a la figura 1.8, on es mostra el resultat d'invocar els dos mètodes sobre l'element arrel del document.

FIGURA 1.8. Diferències d'invocar getChildNodes i getElementsByTagName.



A mode d'exemple, el següent codi permet recórrer un fitxer XML amb una llista de productes i comptar quants n'hi ha a la venda. Per veure les particularitats dels mètodes `getElementsByTagName` i `getChildNodes`, es mostra com fer-ho amb cadascun dels mètodes. Fixeu-vos com, en el primer cas, es pot garantir que els nodes recuperats a la llista són sempre objectes `Element`, mentre que en el segon cas no es pot donar per segur, ja que l'element LlistaProductes també té un atribut, i cal discriminar.

```
Element e = document.getDocumentElement();
  NodeList l = e.getElementsByTagName("Producte");
  for (int i = 0; i < l.getLength(); i++) {
    Element elem = (Element)l.item(i);
    String venda = elem.getAttribute("aLaVenda");
    if ("true".equals( venda) ) {
      nreArticles++;
    }
  }
  System.out.println("Articles a la venda: " + nreArticles);
```

```
Element e = document.getDocumentElement();
  int nreArticles = 0;
  NodeList l = e.getChildNodes();
  for (int i = 0; i < l.getLength(); i++) {
    Node n = l.item(i);
    if (n.getNodeType() == Node.ELEMENT_NODE) {
      Element elem = (Element)n;
      String venda = elem.getAttribute("aLaVenda");
      if ("true".equals( venda) ) {
        nreArticles++;
      }
    }
  }
  System.out.println("Articles a la venda: " + nreArticles);
```

Extracció de text d'un element DOM

De fet, la manera com s'estructuren els diferents tipus de nodes en un model DOM fa que el mecanisme per obtenir el text que hi ha dins un element DOM sigui una mica més complicat del que pot semblar a simple vista, o si més no del que ens agradaria que fos. Val la pena estudiar aquest fet amb detall. El punt més important és que un text dins un element pot estar dispers en diversos nodes de tipus textos dins el model DOM. Per tant, per obtenir tot el text, cal obtenir tots els nodes fill

d'un element (usant `getChildNodes`), veure quins són de text, i concatenar el contingut de cadascun d'ells.

Dins la classe `Node`, les constants que identifiquen nodes que contenen text són `CDATA_NODE` i `TEXT_NODE`.

Vegem-ne un exemple, en què es llisten els noms dels productes que hi ha al document.

```
Element e = document.getDocumentElement();
  //Obtenir tots els nodes del document anomenats "nom"
  NodeList llistaElems = e.getElementsByTagName("Nom");
  //Recorregut d'elements anomenats "Nom"
  for (int i = 0; i < llistaElems.getLength(); i++) {
    Element elem = (Element)llistaElems.item(i);
    NodeList llistaFills = elem.getChildNodes();
    //Recorregut de nodes fill d'un element (text, atributs, etc.)
    String text = "";
    for (int j = 0; j < llistaFills.getLength(); j++) {
      Node n = llistaFills.item(j);
      //Mirar si els nodes són de text
      if ( (n.getNodeType() == Node.TEXT_NODE)||
          (n.getNodeType() == Node.CDATA_SECTION_NODE) ) {

        text += n.getNodeValue();
      }
      System.out.println(text);
    }
  }
```

1.4.3 Generació de fitxers XML

D'entrada, atès que un fitxer XML no és més que text, sempre es pot generar usant qualsevol mecanisme ofert per Java que permeti escriure cadenes de text en un fitxer. Ara bé, si es vol garantir la seva correctesa abans d'escriure'l, el més convenient és fer exactament el pas invers que en el procediment de lectura. Primer es genera un model DOM, usant les classes corresponents, i tot seguit aquest model s'escriu sobre un fitxer.

Creació del model DOM Java

La creació del model DOM parteix de la generació de l'objecte que representa el document, mitjançant la classe `DocumentBuilder`, i després usa crides als mètodes adients per anar-hi afegint els nodes que correspongui. Primer caldrà afegir l'element arrel al document (només un), i a partir d'aquí caldrà anar afegint tots els elements fills que calgui, amb els seus atributs i el text corresponents, fins a tenir l'estructura finalitzada. La classe Element proporciona tres mètodes amb els quals afegir atributs, text i crear relacions pare-fill entre dos elements:

- `void setAttribute(String name, String value)`.
- `void setTextContent(String text)`.
- `void appendChild(Node n)`.

Un fet important al llarg d'aquest procés és que la classe `Element` tampoc no es pot instanciar directament amb una sentència `new`. Per crear un element nou cal cridar el mètode `createElement(String nom)`, que ofereix l'objecte document generat. Això es deu al fet que, internament, un document ha de controlar tots els nodes que conté, i per tant, ha de controlar el procés de generació de tot node que hi pertanyi.

El codi següent serveix d'exemple de tot aquest procés. En aquest codi, es genera una part del document XML on s'enumeren productes. Només hi ha un producte, però afegir-ne més consistiria, simplement, a repetir el mateix els cops que fes falta.

```
1  import javax.xml.parsers.*;
2  import org.w3c.dom.*;
3
4    //...
5
6    //Creació del document
7    DocumentBuilderFactory builderFactory = DocumentBuilderFactory.newInstance();
8    DocumentBuilder  builder = builderFactory.newDocumentBuilder();
9    Document doc = builder.newDocument();
10
11   //Element arrel
12   Element arrel = doc.createElement("LlistaProductes");
13   doc.appendChild(arrel);
14
15   //Element fill de l'arrel
16   Element prod = doc.createElement("Producte");
17   prod.setAttribute("id", "1");
18   prod.setAttribute("aLaVenda", "true");
19
20   //Elements fill de l'anterior
21   Element fill = doc.createElement("Nom");
22   fill.setTextContent("Producte 1");
23   prod.appendChild(fill);
24
25   fill = doc.createElement("Preu");
26   fill.setTextContent("10.0");
27   prod.appendChild(fill);
28
29   fill = doc.createElement("Estoc");
30   fill.setTextContent("4");
31   prod.appendChild(fill);
32
33   arrel.appendChild(prod);
```

Escriptura del model a fitxer

Java ofereix un mecanisme genèric per enviar un document generat usant un model DOM al sistema de sortida de l'aplicació. Mitjançant aquest mecanisme és possible desar el document a fitxer. Ara bé, al tractar-se d'un mecanisme genèric també és possible triar altres destinacions per a les dades, com la sortida estàndard, de manera que també es pot imprimir el resultat per pantalla.

La classe que s'encarrega de gestionar aquesta tasca principalment és l'anomenada `Transformer`, dins el *package* `javax.xml.transform`. Com en el cas de la construcció de documents, aquesta classe no es pot instanciar directament, sinó que cal l'ajut d'una classe auxiliar anomenada `TransformerFactory`.

El codi per desar el document a fitxer és sempre el mateix, i és el següent:

```
import javax.xml.transform.*;
import javax.xml.transform.dom.DOMSource;
import javax.xml.transform.stream.StreamResult;

...

try {
  TransformerFactory tf = TransformerFactory.newInstance();
  Transformer transformer = tf.newTransformer();
  DOMSource source = new DOMSource(doc);
  File f = new File("nouDoc.xml");
  StreamResult result = new StreamResult(f);
  transformer.transform(source, result);

} catch (Exception ex) {
  System.out.println("Error desant fitxer: " + ex);
}
```

Per enviar el document a la sortida estàndard, n'hi ha prou de canviar la instanciació de l'objecte `StreamResult`, i, en lloc d'usar un fitxer, la variable "f" en aquest cas, escriure-hi "System.out" (sense les cometes).

Si proveu aquest codi i obriu el document resultant, veureu un fet que potser us resulta curiós: tot està escrit en una sola línia de text i l'ordre d'alguns atributs no es correspon amb l'ordre d'invocació dels mètodes durant la creació del document.

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?><LlistaProductes><
    Producte aLaVenda="true" id="1"><Nom>Producte 1</Nom><Preu>10.0</Preu><
    Estoc>4</Estoc></Producte></LlistaProductes>
```

Aquest fet no us ha d'alarmar, ja que una de les propietats del processament de dades en XML és que els salts de línia o sagnats en el text no tenen cap influència. Només se solen posar per tal de millorar la llegibilitat, però des del punt de vista de discriminar els nodes dins l'estructura, són irrellevants. El processador els ignora. Per això, en escriure el document, ja directament no es tenen en compte. A més a més, cal considerar que en un document XML l'ordre d'aparició d'atributs o elements no modifica la semàntica de les dades (quan se cerqui una informació, estarà allà igualment), i per tant, tampoc no és un problema.

1.5 Expressions regulars

En moltes aplicacions, una bona part de les dades que cal processar pot prendre la forma de cadenes de text, ja que les persones utilitzem paraules per identificar-nos o comunicar informació, ja siguin estrictament amb lletres (noms, adreces...) o amb xifres. En el cas d'aquestes últimes, s'emmagatzemen també com cadenes de text, ja que no es vol realment fer cap operació matemàtica sobre elles (telèfons, DNI, codis identificadors, ISBN...). Atès aquest fet, una tasca no gaire infreqüent és haver de fer cerques sobre aquestes dades, cercant un element que conté una cadena de text en concret.

ISBN són les inicials de "International Standard Book Number", o estàndard internacional de la numeració de llibres.

En Java, les cadenes de text es representen usant objectes de la classe `String`. Dins aquesta classe hi ha un seguit de mètodes que es poden usar per veure si

un text s'avé a cert criteri, i per tant fer cerques sobre un conjunt de cadenes de text. El mètode més simple i directe seria `equals`, que mira si dues cadenes de text són idèntiques caràcter a caràcter, mentre que entre els mètodes que permeten dur a terme accions més imaginatives es podrien trobar `startsWith`, per veure si comença amb una cadena de text concreta, o `indexOf`, per veure si conté una altra cadena de text. Si es volen fer cerques d'acord a criteris més complicats dels que permeten els mètodes de classe `String`, cal fer un bocí de codi que els combini per assolir la fita. Ara bé, com que totes les comparacions són exactament lletra per lletra, sovint hi ha casos especials on, si es volen tenir en compte totes les possibilitats el codi es pot arribar a fer molt feixuc, com és el cas dels accents (per exemple, fer cerques independentment d'accents o no).

Afortunadament, Java ofereix un mecanisme genèric per veure si una cadena de text compleix un seguit de normes de complexitat arbitrària, anomenades expressions regulars, de manera que és més fàcil fer cerques complexes sobre una gran quantitat de cadenes de text.

Una **expressió regular** és un conjunt de regles de coincidència de patrons codificats com una cadena de text, d'acord a determinades regles de sintaxi de certa complexitat.

1.5.1 Format de les expressions regulars

Podeu trobar la llista completa de tots els caràcters especials a la documentació de la classe `Pattern` a l'API del Java.

El concepte d'expressió regular genèric s'usa en molts llenguatges de programació (com Perl), no està vinculat exclusivament a Java. Per tant, abans de poder entrar en detall en quines classes permeten utilitzar-les, cal tenir clar com generar una expressió regular d'acord als criteris que us interessi.

Una expressió regular es codifica com una cadena de text d'acord a certes normes de format: es va comparant caràcter a caràcter, des de l'inici fins al final, amb la cadena de text que es vol avaluar. Si tots els caràcters coincideixen, l'avaluació retorna cert, mentre que si algun és diferent, retorna fals. La particularitat està en el fet que, dins les expressions regulars, es poden incloure caràcters especials amb diferents significats que van més enllà de comparar només un caràcter concret. El nombre de caràcters especials és molt gran, per la qual cosa ens centrarem només en una part petita.

L'expressió regular més simple és un literal, o sigui un text directament, com "abc" o "Hola". En aquest cas, avaluar l'expressió regular seria exactament igual que comparar-la amb la cadena de text, com es faria invocant el mètode `equals`. Dins d'aquest literal, un punt (".") equival a un comodí per dir "qualsevol lletra". Per tant, l'expressió ".ola" avalua a cert tant per "Hola" com per "hola".

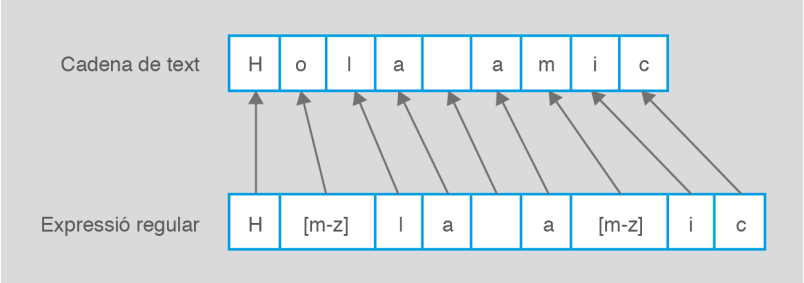
A la posició d'un caràcter dins l'expressió regular també s'hi pot posar, en lloc de només un caràcter, un conjunt de caràcters entre claus ([...]), anomenats símbols de coincidència (*matching symbols*). En aquest cas, el caràcter que es troba en

aquella posició a la cadena a avaluar es compara amb tots els caràcters entre les claus seguint certes condicions.

- `[...]`, Una llista de caràcters consecutius. Es mira si el caràcter a avaluar és algun de la llista.
- `[^...]`, Una llista de caràcters consecutius iniciats amb "^". Es mira si el caràcter a avaluar NO és algun de la llista.
- `[.-.]`, Dos caràcters separats amb un guió. Es mira si el caràcter a avaluar està entre els dos caràcters.
- `[.-.-[...]]`, Es mira si el caràcter a avaluar està entre els dos caràcters separats pel guió, però no entre els que marca el segon bloc `[...]`, que pot usar qualsevol de les regles anteriors.

Aquestes regles es poden combinar indefinidament per fer regles més complexes. Cal fer èmfasi, però, en el fet que aquest conjunt de caràcters entre claus es compara només amb un únic caràcter de la cadena a avaluar, seguint ordenadament ambdues cadenes, tal com mostra la figura 1.9.

Figura 1.9. Avaluació d'una expressió regular amb símbols de coincidència.



Com sempre, vegem-ho més clar amb els exemples de la taula 1.5.

Taula 1.5. Exemples d'expressions regulars.

Símbol	Resultat
[abc]	Es mira si el caràcter és a, b o c.
[^abc]	Negació. Es mira si el caràcter no és ni a, ni b, ni c.
[a-z]	Es mira si és un caràcter entre a-z.
[a-z-[bc]]	Es mira si el caràcter està entre a-z però no compleix "...".
[a-zA-Z]	Unió dels casos 1 i 3. Es mira si és un caràcter entre a-z o A-Z.
[a-z-[m-p]]	Unió dels casos 4 i 3. Es mira si el caràcter està entre a-z però no entre m i p.
[a-z-[^def]]	Unió dels casos 4 i 2. És d, e o f.

Donada aquesta sintaxi, les expressions regulars de la taula 1.6 avaluarien a cert o fals per a la cadena de text "Hola amic".

TAULA 1.6. Exemples d'avaluació d'expressions regulars.

Expressió	Resultat
Hola ami	fals, falta la c final.
Ho[bfl]a amic	cert, el 3er caràcter és b, f o l.
Ho[bfl]a ami[^cdf]	fals, el darrer és c (i no hauria de ser ni c, ni d, ni f).
H[m-z]la a[m-z]ic	cert, en els dos casos el caràcter està entre m i z.
H[m-z-[opq]]la a[m-z]ic	fals, el 2on caràcter hauria d'estar entre m i z, però no pot ser ni o, ni p, ni q.
[a-mA-M]ola [a-mA-M]mic	cert, en els dos casos el caràcter està entre a i m, o A i M.

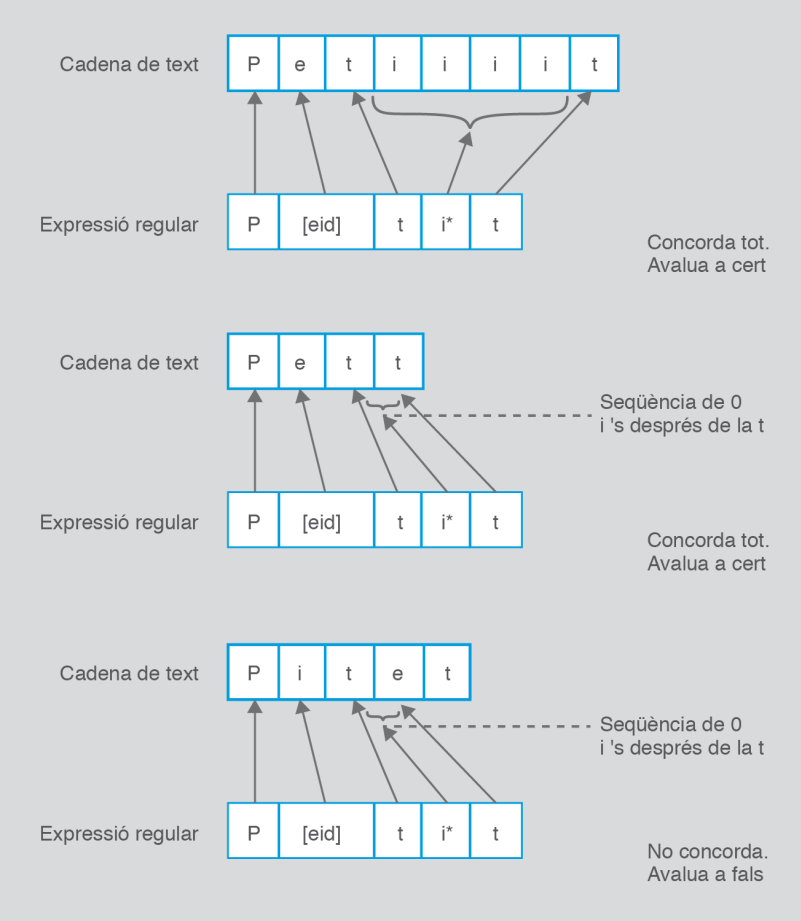
Un altre conjunt de símbols amb una semàntica especial dins una expressió regular són aquells que permeten controlar repeticions de caràcters. Aquests s'escriuen immediatament després de qualsevol símbol (ja sigui un caràcter normal o un símbol de coincidència) i actuen associats a ell (vegeu la taula 1.7).

TAULA 1.7. Llistat de símbols amb significats especials.

Símbol	Resultat
*	El símbol associat es repeteix de 0 a N vegades (N no fitada).
+	El símbol associat es repeteix de 1 a N vegades (N no fitada).
?	El símbol associat apareix de 0 o 1 vegada.
{X}	El símbol associat apareix X vegades.
{X,Y}	El símbol associat apareix entre X i Y vegades.

La seva principal particularitat és que, quan s'apliquen sobre la cadena a avaluar, aquests símbols actuen sobre més d'un caràcter, de manera que ja no hi ha una correspondència 1 a 1 entre els caràcters de la cadena i de l'expressió, com s'ha vist fins ara. Per seguir l'avaluació d'una expressió regular d'aquest tipus cal considerar que aquesta és qui porta la batuta del procés, i cada símbol dins seu es consumeix cada cop que es troba una coincidència dins el text a avaluar, sigui del nombre de caràcters que sigui (que pot ser 0, pel cas "*"). El procés avaluarà a cert només si, en acabar el procés, tots els caràcters de la cadena de text han concordat amb algun símbol de l'expressió, i no en queda cap sense aparellar. Aquest fet s'esquematitza a l'exemple de la figura 1.10.

FIGURA 1.10. Tractament de l'expressió regular amb símbols per controlar repeticions.



A la taula 1.8 es mostren alguns exemples d'ús d'aquests símbols.

TAULA 1.8. Exemples d'ús de símbols especials a expressions regulars.

Cadena a avaluar	Expressió	Resultat
Una miiiica	Una mi*ca	cert, a partir de la 6a posició hi ha 0 o més caràcters 'i', i al final un "ca".
Una mca	Una mi*ca	cert, a partir de la 6a posició hi ha 0 o més caràcters 'i'.
Una miiiica	Una mi+ca	cert, a partir de la 6a posició hi ha 1 o més caràcters 'i'.
Una mca	Una mi+ca	fals, a partir de la 6a posició no hi ha 1 o més caràcters 'i'.
Una miiiica	Una mi?ca	fals, a partir de la 6a posició hi ha més de 0 o 1 caràcter 'i'.
Una mica	Una m[a-f]+ca	fals, a partir de la 6a posició hi ha 1 o més caràcters entre a i f (hi ha una i).
Una mica	Una m[a-f]*ca	fals, ara bé, la condició "[a-f]*" sí que es compleix, ja que a partir de la posició 6 no hi ha cap caràcter entre a i f (per tant, n'hi ha entre 0 i N), però la 'i' de la cadena de text no encaixa amb cap part de l'expressió regular (que acaba en "ca").

1.5.2 Aplicació d'expressions regulars

Un cop s'és capaç de generar cadenes de text que codifiquen expressions regulars correctes, Java ofereix dues vies diferents per fer-ne ús dins els vostres programes, usant uns pocs mètodes que aporta la classe `String`, per a tasques simples, o mitjançant classes específicament creades per fer aquesta mena de tasques.

Aplicació directa sobre objectes String

La manera més directa d'aplicar expressions regulars sobre cadenes de text en Java és mitjançant el mètode `matches` que ofereix la classe `String`. El seu paràmetre és una cadena de text que codifica una expressió regular que segueixi correctament les regles de sintaxi. La dificultat a l'hora d'usar-lo recau molt més en el fet de generar una expressió regular correcta que no pas en la seva invocació, ja que és molt senzill d'usar. Per exemple, suposem que es vol comprovar si un BIF està en format correcte (7 xifres i una lletra majúscula):

```
1 String text = "1234567B";
2 String regex = "[0-9]{7}[A-Z]";
3 System.out.println(text.matches(regex) );
```

Si reflexioneu sobre quin seria el codi usant altres mètodes de la classe `String` que no es basen en expressions regulars, de ben segur que us adonareu que resultaria molt més complex. Només per veure si al final hi ha una lletra, caldria comparar el caràcter amb totes les lletres de l'abecedari, una a una. Mitjançant `matches`, s'ha fet amb una sola línia.

La classe `String` proporciona dos mètodes addicionals que operen partint d'expressions regulars, si bé el seu objectiu no és veure si aquesta avalua a cert o fals. El seu comportament és una mica diferent, ja que no intenta encaixar l'expressió amb tota la cadena de text, sinó que cerca aparicions de l'expressió dins la cadena i opera sobre aquestes subcadenes trobades.

Per una banda, `replaceAll (String expressió, String substitució)` reemplaça totes les coincidències d'expressions regulars dins la cadena de text que invoca el mètode amb la cadena de text indicada amb el paràmetre "substitució". Qualsevol part de la cadena que coincideix amb l'expressió se substitueix. El mètode retorna la nova cadena resultant (ja que els objectes `String` en Java són immutables). Cada grup de lletres sencer que coincideix amb l'expressió és reemplaçat i, si passa més d'un cop, hi haurà diversos canvis dins la cadena original.

Per exemple, en el codi següent, aquest mètode s'usa per reemplaçar tots els grups de lletres `t` i `s` precedides per 4 lletres qualsevol per una `x`. El resultat final és "Ha x molt poc x", ja que els grups de lletres que compleixen aquesta condició són "estat" i "temps".

```
1 String text = "Ha estat molt poc temps";
2 String regex = "[a-z]{4}[ts]";
3 String res = text.replaceAll(regex, "x");
```

D'altra banda, `String[] split (String expressió)` divideix la cadena en parts d'acord amb la coindicència de l'expressió regular. El mètode retorna un *array* de cadenes on cada element és una part de l'original. Les parts coincidents amb l'expressió no s'inclouen enlloc. Per exemple, seguint l'exemple anterior, el següent codi retorna l'*array* {"Ha ", " molt poc "}.

```
1 String text = "Ha estat molt poc temps";
2 String regex = "[a–z]{4}[ts]";
3 String[] res = text.split(regex);
```

Aplicació mitjançant classes dedicades

Una manera més sofisticada de treballar amb expressions regulars és usant les classes incloses dins el *package* `java.util.regex`, `Pattern` i `Matcher`, que han estat creades amb aquest propòsit específic. La primera serveix per establir una expressió regular (concretament per compilar-la), mentre que la segona és l'encarregada de processar-la i indicar si troba coincidències. La principal particularitat d'aquestes classes és que permeten cercar coincidències ja siguin parcials o totals dins la cadena de text, així com identificar exactament on han succeït. Això atorga un nivell de detall molt superior al que proporcionen els mètodes de la classe `String`.

Les opcions d'operació possibles s'enumeren en forma de constants a la classe `Pattern`. Podeu consultar-ne la documentació.

El procés per usar aquestes dues classes sempre és el següent:

1. Instanciar un objecte `Pattern` usant el seu mètode estàtic `compile(String regex)`. No es pot fer usant `new`.
2. Obtenir un objecte `Matcher` associat al text a processar, invocant `matcher(CharSequence input)`. Com a paràmetre, es pot usar un `String` també.
3. A partir de l'objecte `Matcher`, es pot processar el text de diverses maneres invocant els diferents mètodes que ofereix aquesta classe.

La classe `Matcher` permet anar cercant coincidències dins la cadena de text, una a una, a poc a poc (en lloc de totes de cop), de manera semblant, fins a cert punt, a com s'aniria recorrent una llista on els elements són els blocs de text que coincideixen amb l'expressió regular. Els seus mètodes més rellevants són:

- `boolean find()`, cerca si hi ha alguna coincidència dins la cadena de text a processar. Successives invocacions van retornant cert mentre es trobi alguna coincidència. Un cop s'han esgotat totes les coincidències possibles, retorna fals.
- `int start()`, indica quin és l'índex del caràcter on s'inicia la darrera coincidència trobada (el darrer cop que s'ha cridat `find` i ha retornat cert).
- `int end()`, indica quin és l'índex del primer caràcter **posterior** a la darrera coincidència.

- `String group()`, indica quina és la cadena de text exacta que ha provocat la darrera coincidència.
- `Matcher reset()`, reinicialitza l'objecte, de manera que es torna a començar des de zero el procés de detecció de coincidències (crides al mètode `find`).

Novament, la millor manera de veure com funciona aquesta classe és mitjançant un exemple. El codi següent va cercant dins una cadena de text tots aquells casos on hi ha una lletra "t" o "s" precedida de grups de 3 a 6 lletres minúscula qualsevol. Si analitzeu aquesta condició vosaltres mateixos, es pot veure que hi ha quatre casos on això succeeix: "estat" (t precedida de 4 lletres), "molt" (t precedida de 3 lletres), "emps" (s precedida de 4 lletres, la "T" no entra ja que és majúscula) i "dedicat" (t precedida de 6 lletres). El cas de la cadena "has" no coincideix, per exemple, ja que està precedida només per dues lletres. Per tant, es poden fer 4 successives invocacions a `find` que s'avaluïn a cert.

```
1  import java.util.regex.*;
2
3  public class Cercador {
4    public static void main(String[] args) {
5
6      String text = "Ha estat molt poc Temps el que hi has dedicat.";
7      String regex = "[a-z]{3,6}[ts]";
8
9      Pattern p = Pattern.compile(regex);
10     Matcher m = p.matcher(text);
11     while (m.find() == true) {
12       System.out.print("Cadena: " + m.group() );
13       System.out.print(" (Inici: " + m.start() );
14       System.out.println(", Fi: " + m.end() + ")");
15     }
16   }
17 }
```

El resultat de la seva execució és la següent. Fixeu-vos en els valors d'inici i de final obtinguts a partir de les crides a `start` i `end`. Per exemple, la cadena "estat" comença a la posició 3 del text complet, i el caràcter que hi ha a continuació (l'espai en blanc entre "estat" i "poc") es troba a la posició 8.

```
1 Cadena: estat (Inici: 3, Fi: 8)
2 Cadena: molt (Inici: 9, Fi: 13)
3 Cadena: emps (Inici: 19, Fi: 23)
4 Cadena: dedicat (Inici: 35, Fi: 42)
```

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.

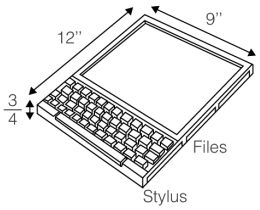
1. Interfícies gràfiques d'usuari

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

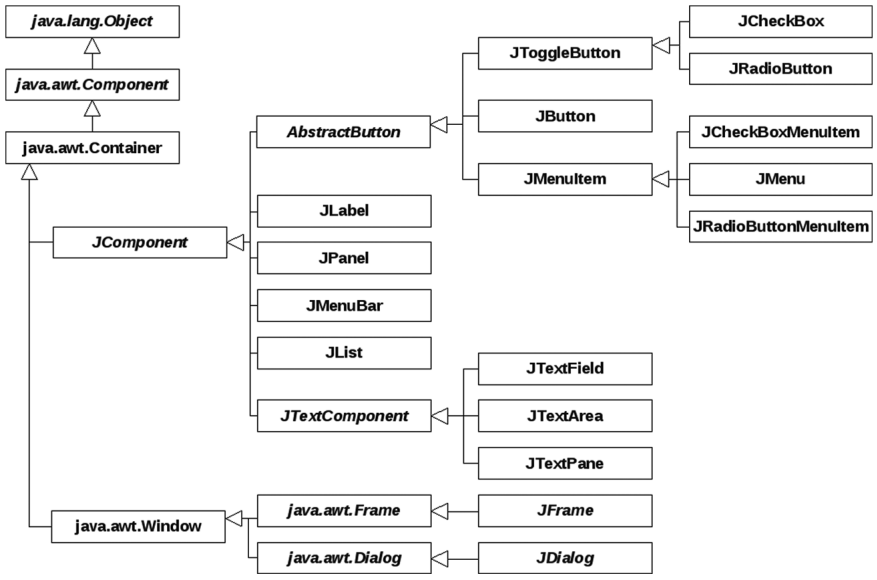
Una jerarquia de classes és la manera com es classifiquen diferents classes amb relacions d'herència entre elles.

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.

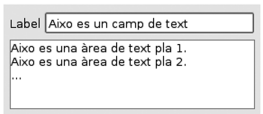
Les classes amb noms que comencen per *J* en la figura són les incloses a Swing, mentre que la resta pertanyen a AWT o a la biblioteca estàndard del Java. Com es pot veure, una part de la biblioteca original AWT continua present dins Swing en forma de les superclasses `java.awt.Component` i `java.awt.Container` (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.

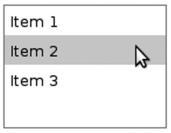
JCheckBox i JRadioButton

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.

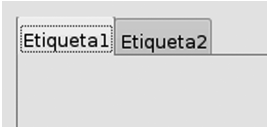
1.1.2 Agregació de components

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.

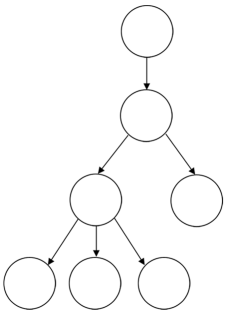
Exemples de contenidors poden ser JFrame, JDialog, JApplet, JPanel, JScrollPane, JTabbedPane.



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".

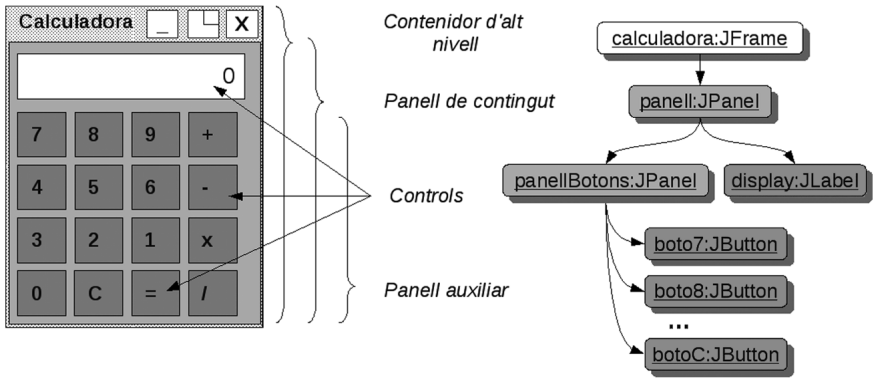
> Per incloure qualsevol **component** dins un contenidor, cal cridar el mètode, `add(Component comp)` sobre el contenidor, passant com a paràmetre el component a afegir-hi. Aquest es troba definit en la classe `java.awt.Container` i totes les classes de la biblioteca Swing l'hereten.

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.

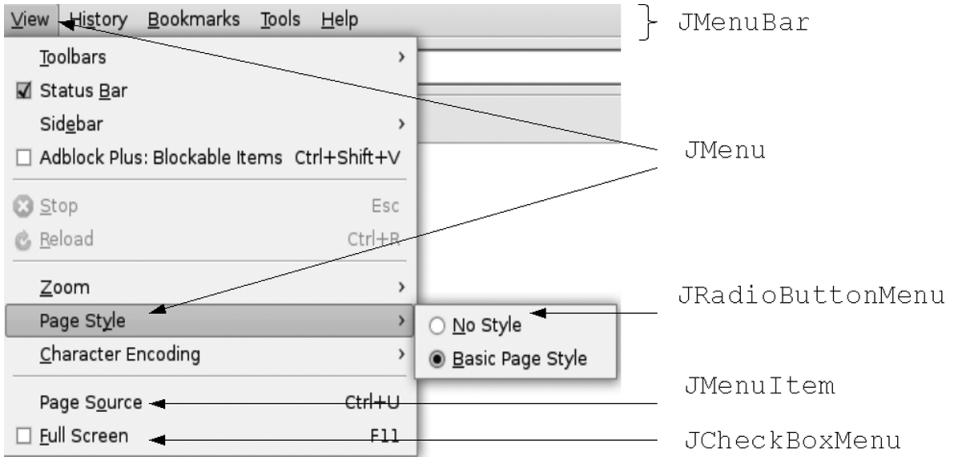
1.1.3 La barra de menús

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.

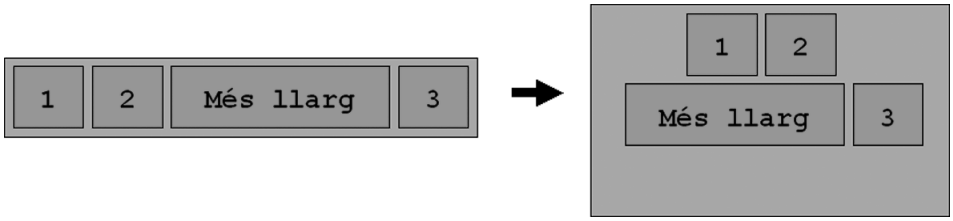
FlowLayout

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)
```

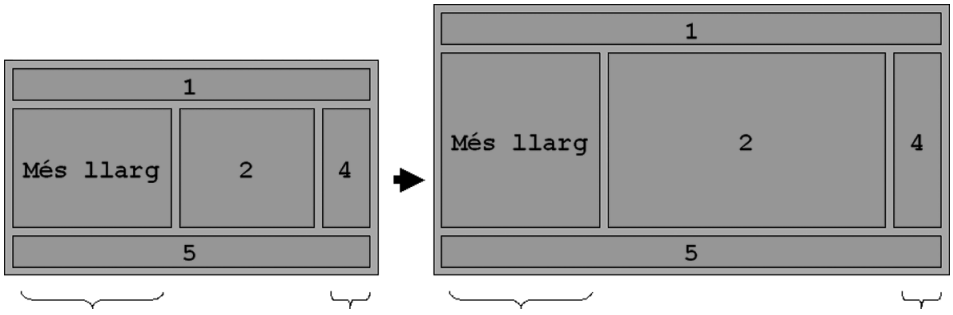
La classe BorderLayout defineix cinc constants que serveixen per indicar cada zona en el paràmetre `constraints`:

- `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

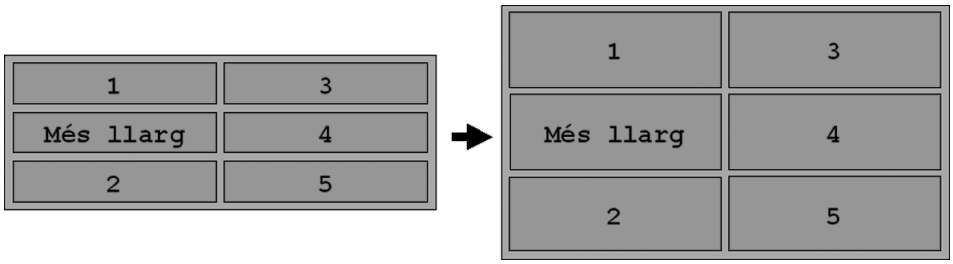
El GridLayout organitza el component com una matriu amb cel·les de mida idèntica. En cada cel·la només hi pot haver un component, que ocupa tot l'espai disponible independentment de quina en seria la mida per defecte. Els components s'ubiquen per files, d'esquerra a dreta en el ma-add teix ordre en què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

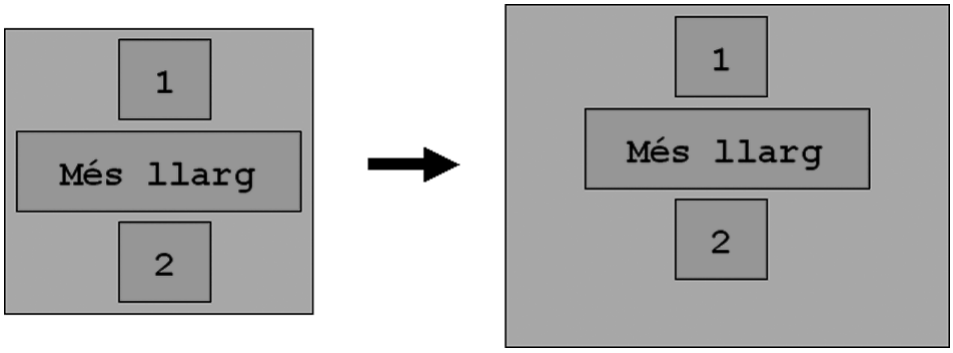
El BoxLayout s'acostuma a aplicar sobre un contenidor específic, el Box, que ja porta incorporat aquest *layout* per defecte en ser instanciat i no es pot canviar. Els components que conté mantenen la mida per defecte i s'ubiquen en una sola línia horitzontal o vertical, centrada.

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



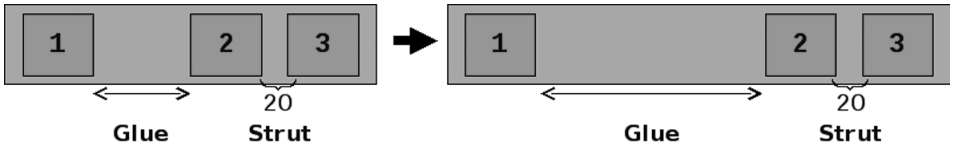
Una altra de les particularitats dels contenidors Box és la seva capacitat d'incloure, usant el mètode add, dos components especials que cap altre tipus de *layout* pot usar: les *Struts* i les *Glues*, verticals i horitzontals. Donada una instància de Box, només es poden afegir *Glues* o *Struts* de 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.

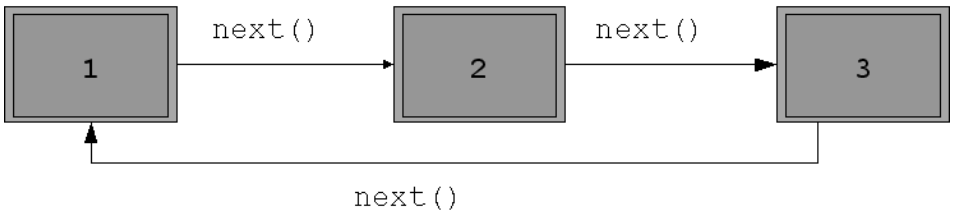
El seu constructor és `public CardLayout()`.

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)
```

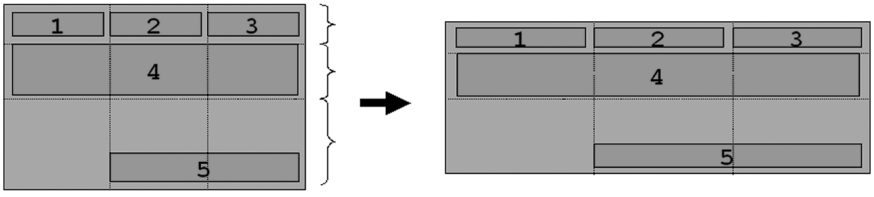
La classe `GridBagConstraints` té un conjunt de propietats amb les quals definim com s'ubica l'element afegit dins la graella i quantes caselles ocupa per cada eix. Algunes de les més significatives són:

- **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 constantGridBag-Constraints.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()`

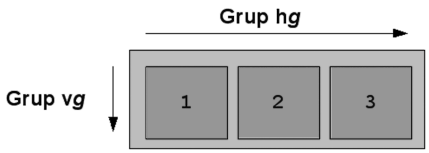
La clau d'aquest *layout* està en el fet que almenys hi ha d'haver un grup associat a l'eix vertical i un altre a l'horitzontal, i tots els components han de pertànyer a dos grups, un de l'eix vertical i un de l'horitzontal. A partir del grup on estan associats, se'n pot calcular la ubicació.

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");
```

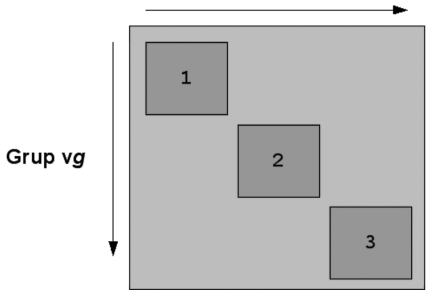
```
8  hg.addComponent(l1);
9  hg.addComponent(l2);
10 hg.addComponent(l3);
11 vg.addComponent(l1);
12 vg.addComponent(l2);
13 vg.addComponent(l3);
14 layout.setHorizontalGroup(hg);
```

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).



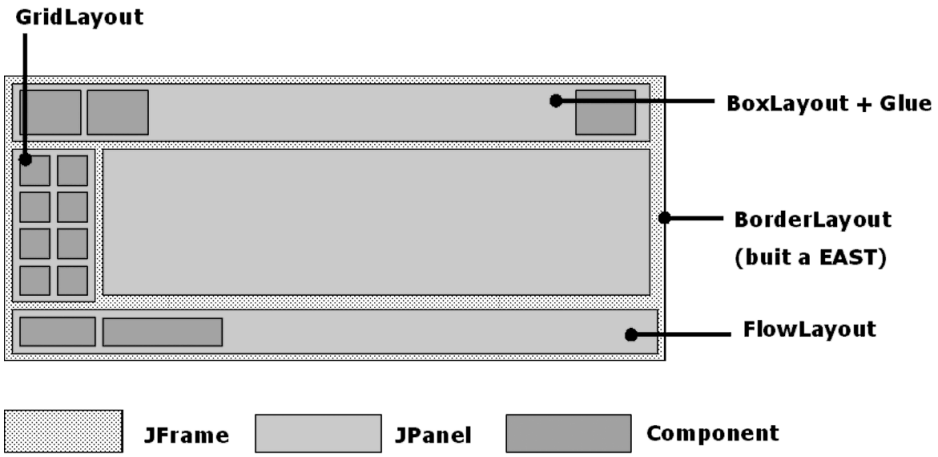
Una de les particularitats més importants del GroupLayout és que permet crear composicions d'elements sense haver de dependre de subpanells auxiliars. Els grups ja fan aquest paper.

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.

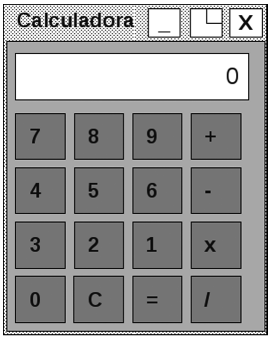
FIGURA 1.13. Combinació de layouts per generar una interfície gràfica complexa.



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`.

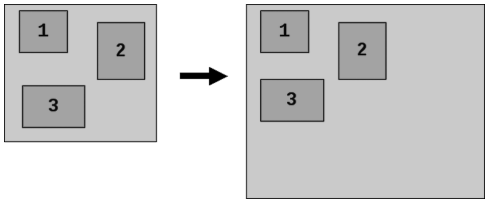
Afortunadament, a l'hora de generar *interfaces* amb composicions complexes, en què cal ubicar els components de manera molt especial, hi ha una alternativa molt útil: els *layouts* de posicionament absolut (o, simplement, *layouts* absoluts). Algunes biblioteques externes a les estàndard del Java proporcionen 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.

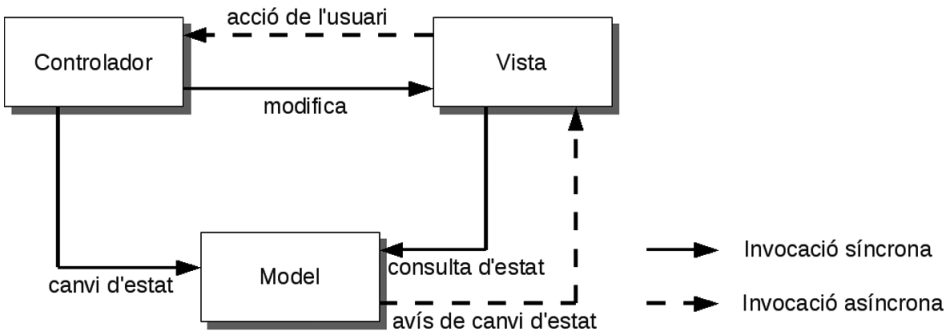
S'entén per fer doble clic pitjar un botó del ratolí dues vegades molt ràpidament.

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.

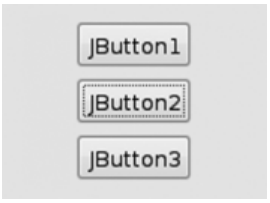
Reflexionant una mica, el concepte d'esdeveniment asíncron amb un codi associat no és un aspecte totalment nou, ja que conceptualment no és gens diferent de l'usat en el Java per al control d'errors mitjançant excepcions.

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.

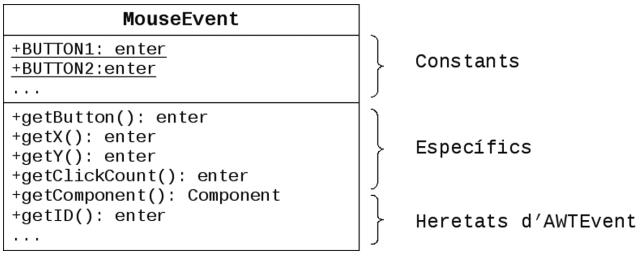
- **MouseEvent**: Generat davant qualsevol acció vinculada exclusivament al ratolí. Per exemple, en pitjar qualsevol botó del ratolí, moure l'apuntador dins una àrea concreta, etc.
- **KeyEvent**: Associat a accions exclusivament relatives al teclat. Per exemple, pitjar una tecla o deixar-la anar.
- **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

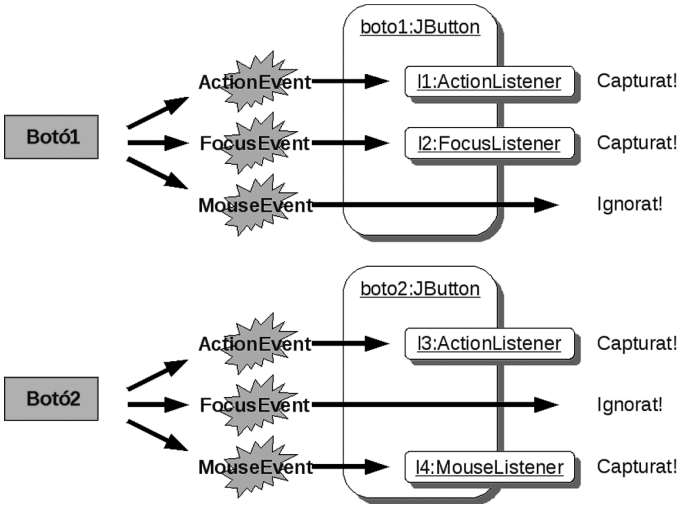
Donat un tipus d'esdeveniment "xxxEvent", normalment, el seu "Listener" associat té el nom "xxxListener". El mètode per assignar-lo a un component es diu "addxxxListener".

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.

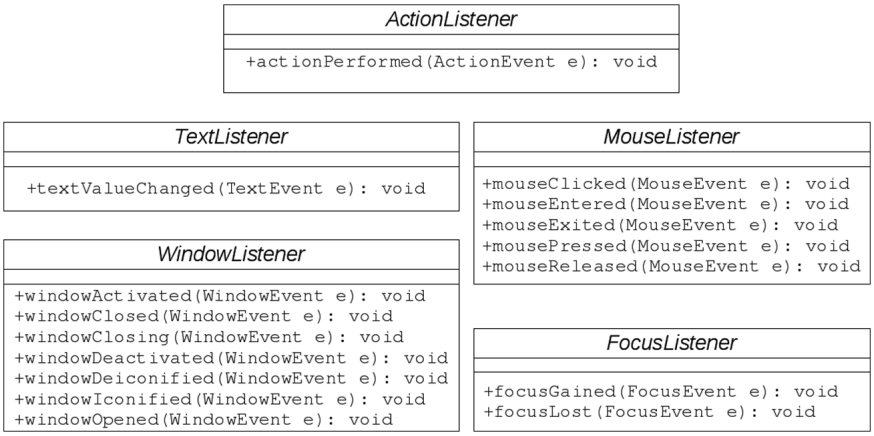
FIGURA 1.17. Exemple de registre de Listeners i captura d'Events.



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.

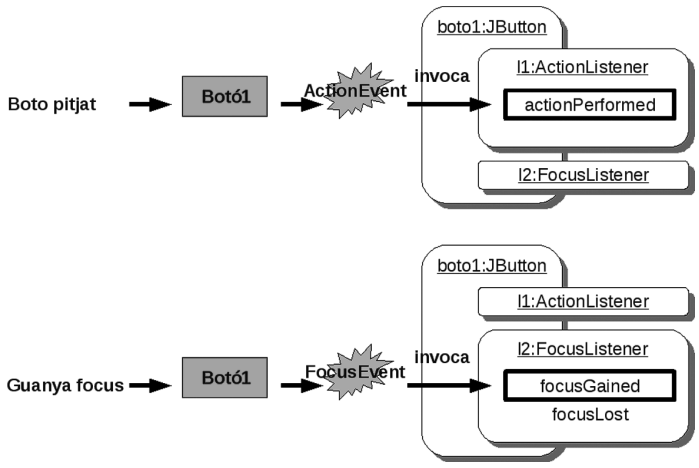
La figura 1.18 mostra els *Listeners* més usats d'entre els definits pel Java, amb tots els seus mètodes.

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.

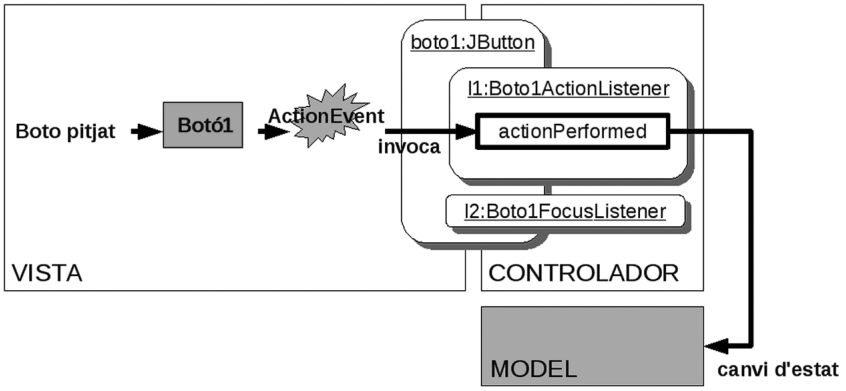
Donats els rols dels elements del patró MVC, és justament dins els mètodes sobreescrits a aquests *Listeners* personalitzats des d'on cal fer les crides cap a objectes del Model. La figura 1.20 mostra una visió general de tot el sistema, associant ja cada part implicada al patró MVC. Recordem que, pel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

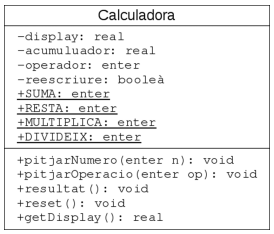
FIGURA 1.20. Assignació de codi als esdeveniments usant mètodes polimórfics i usant el patró MVC.

```
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 internes. 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

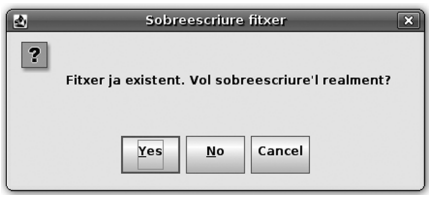
La biblioteca gràfica Swing és molt extensa i ofereix un conjunt de funcionalitats que, si són conegudes pel desenvolupador, poden estalviar-li molta feina. Swing permet realitzar tasques habituals dins la generació d'una *interface* gràfica de manera flexible i amb poc esforç. Per veure com funcionen 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.

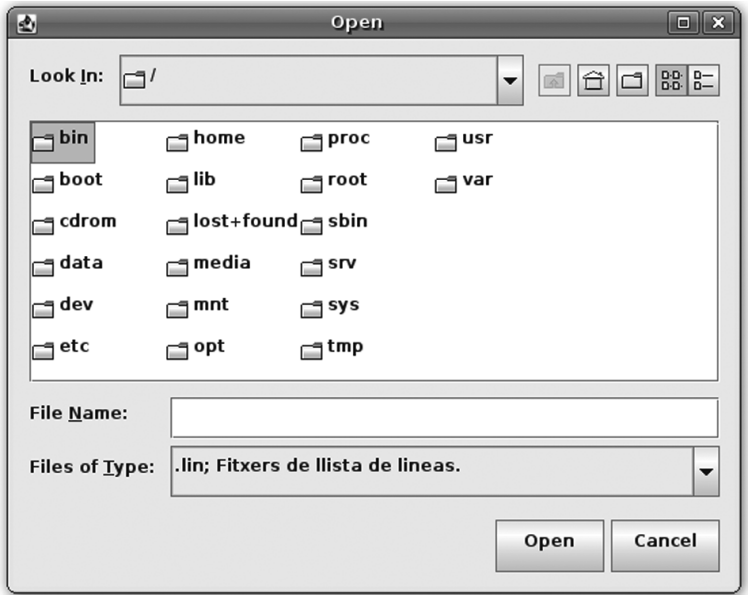
```
1  JFrame finestra = new JFrame();
2  ...
3  int resposta = JOptionPane.showConfirmDialog(finestra,
4    "Fitxer ja existent. Vol sobreescriure'l realment?",
5    "Sobreescriure fitxer",
6    JOptionPane.YES_NO_CANCEL_OPTION,
7    JOptionPane. QUESTION_MESSAGE);
8
9  switch(resposta) {
10   case 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.

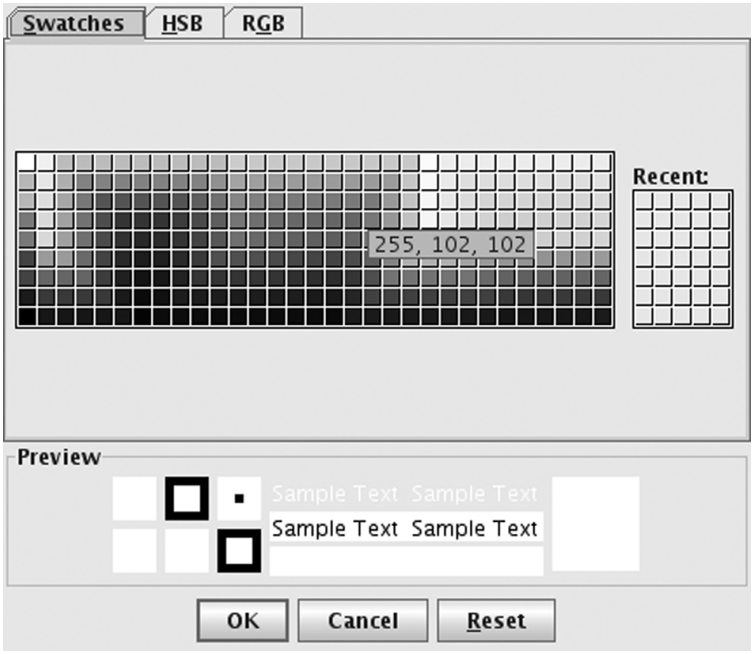
A continuació es mostra el codi que genera el selector de fitxers representat en la figura 1.22.

```
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

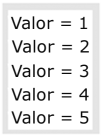
La classe `JList` correspon a una llista d'elements, de la qual se'n pot seleccionar un o més. Si es mira la documentació, es pot apreciar que no té cap mètode amb el qual afegir aquests elements, ja que cal usar el seu Model per fer-ho. El Model definit per interactuar-hi és la *interface* ListModel. 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

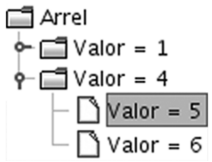
En aquest cas, també hi ha implementacions parcials, definides juntament amb el Model dins el paquet `javax.swing.table`. Aquestes són la classe `AbstractTableModel`, que proporciona una implementació parcial, i la classe `DefaultTableModel`, que en proporciona una de completa amb un comportament per 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.

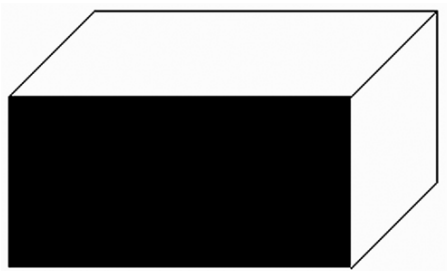
- **drawRoundRect(int x, int y, int width, int height, int arcWidth, int arcHeight)**. Dibuixa un rectangle de cantonades arrodonides, partint de les coordenades (x,y), amb una amplada i alçària de width i heightpíxels. El grau d'arrodoniment de les cantonades està determinat pels valors arcWidthi arcHeightde 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*.

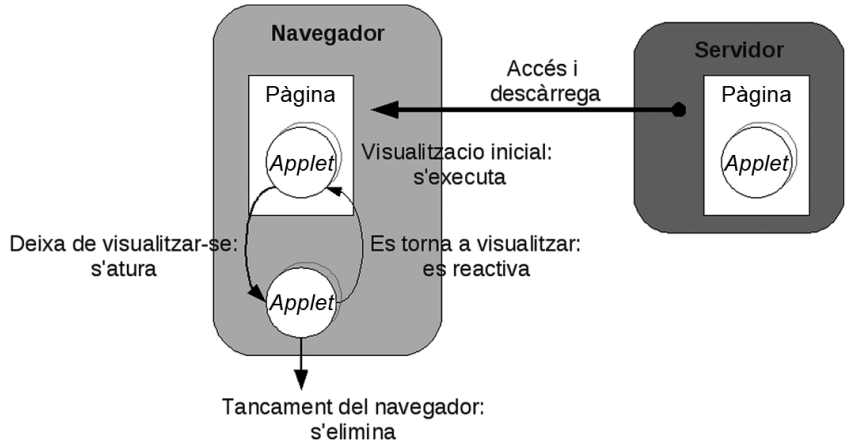
Provant applets

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.

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*.

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.

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.*

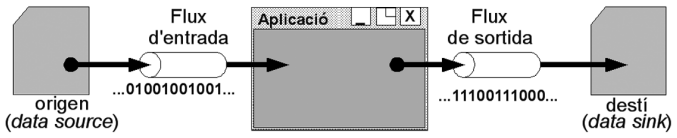
2. Fluxos i fitxers

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.

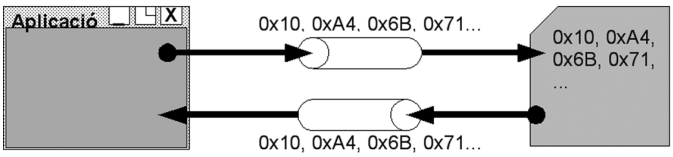
FIGURA 2.1. Fluxos d'entrada i de sortida



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

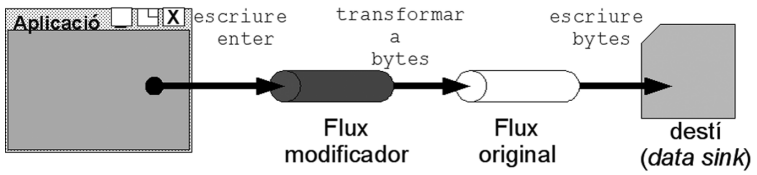
Hi ha situacions en les quals haver de processar qualsevol informació binària directament a un nivell tan baix com de byte o de caràcter pot ser una feina pesada per al desenvolupador. Suposem que es vol emmagatzemar un valor enter, un tipus primitiu `int`. Amb el funcionament per defecte dels fluxos 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.

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`.

- **void println(...)**. Escriu la representació en forma de cadena de text del paràmetre d'entrada i al final fa un salt de línia.

`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:

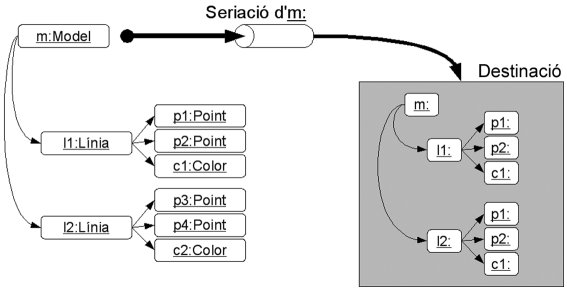
```
1 public class Model {...}
```

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:

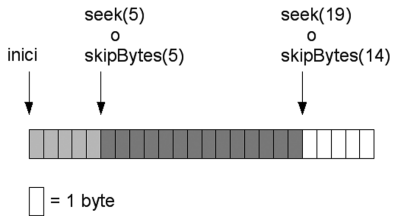
- r **:** Mode escriptura. Qualsevol intent d'escriure en el fitxer, incloent-hi el fet que no existeixi, causarà una excepció.
- rw: Mode escriptura-lectura. Si el fitxer no existeix, se'n crearà un de nou, buit.
- rws: Igual que el cas rw, però, addicionalment, es força l'actualització al sistema de 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.

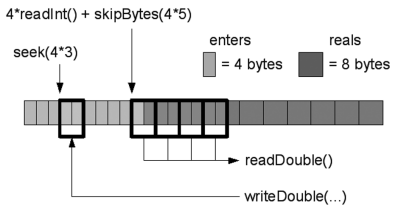
De la mateixa manera, si es fes:

```
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:

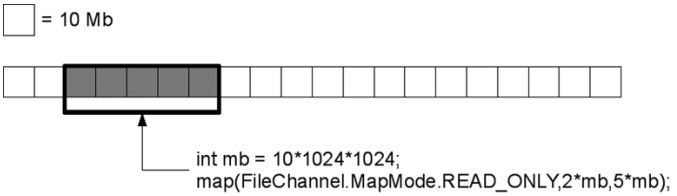
```
1 MappedByteBuffer map(FileChannel.MapMode mode, long position, long size).
```

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).` Łvoid `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.

POO i gestors de base de dades

Joan Arnedo Moreno

Índex

Introducció

El conjunt d'objectes existents a memòria en un moment donat de l'execució del programa indiquen l'estat intern de l'aplicació. Aquest va canviant d'acord a les interaccions de l'usuari. En tractar-se d'objectes emmagatzemats a la memòria de l'ordinador, quan l'aplicació deixa d'executar-se, totes les dades associades desapareixen per sempre, no hi ha persistència. A més a més, aquestes dades només són accessibles per l'aplicació que ha instanciat els objectes.

Sovint, és necessari assolir la persistència de dades, de manera que sigui possible conservar certa informació entre diferents execucions de l'aplicació, o compartir dades entre diferents aplicacions. La manera més simple de fer-ho és desant totes les dades dins d'un fitxer, però aquesta solució és poc eficient per a aplicacions complexes, o simplement no és factible si cal compartir informació entre aplicacions que no s'executen sobre el mateix ordinador. Les aplicacions modernes normalment s'inclinen per usar alternatives millors, com les bases de dades.

En aquesta unitat se segueix amb l'estudi de les classes accessibles a través de l'API del Java, ja que, igual que existeix un conjunt de classes accessibles mitjançant l'API del Java per gestionar errors o interfícies d'usuari, també hi ha una part que gestiona l'accés a bases de dades i que, per tant, val la pena estudiar amb més detall. Ara bé, per a alguns casos de bases de dades l'API no és suficient i no disposa de cap classe que resolgui la tasca que cal dur a terme. Quan això succeeix, cal anar més enllà i estudiar com usar llibreries creades per altres programadors diferents dels creadors del Java i incorporar-les als vostres programes.

Abans de poder establir com fer millor ús d'una base de dades, però, cal tenir ben clar de quina manera s'estructurarà tota la informació dins el vostre programa. Quins objectes hi ha i quines dades contenen. Per assolir aquesta fita, una eina molt útil és el llenguatge UML, que permet representar el disseny d'aplicacions orientades a objectes de manera relativament intuïtiva.

L'apartat "Introducció al diagrama estàtic UML" dóna una visió sobre com dur a terme el disseny d'aplicacions utilitzant aquest llenguatge d'especificació d'aplicacions orientades a objectes. D'aquesta manera, abans de ni tan sols començar a escriure codi font, el programador ja pot definir clarament sobre quina informació ha de tractar l'aplicació i com s'estructura a memòria, de la mateixa manera que un arquitecte no comença a construir un gratacels sense haver fet uns plànols abans.

L'apartat "Aplicacions amb BD no orientades a objectes" presenta els aspectes bàsics per poder assolir la persistència de les dades mitjançant el mecanisme més popular per desplegar aplicacions a gran escala: una base de dades relacionals. Aquest sistema es basa en taules i no està vinculat explícitament a les aplicacions

orientades a objectes, sinó a qualsevol tipus d'aplicació. Per tant, per fer-ne ús cal un pas previ de traducció del model orientat a objectes a un model relacional.

L'apartat "Aplicacions amb BD orientades a objectes" introdueix un nou tipus de bases de dades, relativament recents, on la informació emmagatzemada sí que s'estructura d'una manera semblant a com es fa amb els objectes a memòria dins una aplicació orientada a objectes. Per tant, això permet estalviar-se el pas de la traducció prèvia i accedir a les seves dades d'una manera més natural.

Un aspecte molt important al llarg de tota la unitat és ser conscient de la impossibilitat de descriure fins la darrera funcionalitat de totes les classes implicades tant en una interfície gràfica com en la gestió de l'entrada/sortida per assolir persistència. Les llibreries de Java són molt extenses i complexes. Per tant, és inevitable haver d'acudir a la documentació oficial (l'anomenada API de Java) 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. Gestiona informació emmagatzemada en bases de dades relacionals mantenint la integritat i consistència de les dades.

- Identifica les característiques i mètodes d'accés a sistemes gestors de bases de dades relacionals.
- Programa connexions amb bases de dades.
- Escriu codi per emmagatzemar informació en bases de dades.
- Crea programes per recuperar i mostrar informació emmagatzemada en bases de dades.
- Efectua esborrats i modificacions sobre la informació emmagatzemada.
- Crea aplicacions que executin consultes sobre bases de dades.
- Crea aplicacions per a possibilitar la gestió d'informació present en bases de dades relacionals.

2. Gestiona informació emmagatzemada en bases de dades objecte- relacionals mantenint la integritat i consistència de les dades.

- Identifica les característiques de les bases de dades objecte-relacionals.
- Analitza la seva aplicació en el desenvolupament d'aplicacions mitjançant llenguatges orientats a objectes.
- Classifica i analitza els diferents mètodes que suporten els sistemes gestors de bases de dades per a la gdestió de la informació emmagatzemada de forma objecte-relacional.
- Programa aplicacions que emmagatzemen objectes en bases de dades objecte-relacionals.
- Realitza programes per recuperar, actualitzar i eliminar objectes de les bases de dades objecte-relacionals.
- Realitza programes per emmagatzemar i gestionar tipus de dades estructurades, compostos i relacionats.

3. Utilitza bases de dades orientades a objectes, analitzant les seves característiques i aplicant tècniques per mantenir la persistència de la informació

- Identifica les característiques de les bases de dades orientades a objectes.

- Analitza la seva aplicació en el desenvolupament d'aplicacions mitjançant llenguatges orientats a objectes.
- Defineix les estructures de dades necessàries per a l'emmagatzematge d'objectes.
- Classifica i analitza els diferents mètodes suporten els sistemes gestors per a la gestió de la informació emmagatzemada.
- Programa aplicacions que emmagatzemin objectes en les bases de dades orientades a objectes.
- Realitza programes per recuperar, actualitzar i eliminar objectes de les bases de dades orientades a objectes.
- Realitza programes per emmagatzemar i gestionar tipus de dades estructurades, compostos i relacionats.

1. Introducció al diagrama estàtic UML

A l'hora de resoldre un problema, no sol ser una bona idea començar a anar per feina sense aturar-se un moment a pensar què es vol fer i, en el cas de problemes complexos, cal idear sobre el paper un petit esquema o croquis de què cal fer. Llavors, sobre el paper, es pot reflexionar si la solució sembla tenir sentit i es pot anar refinant, ja sigui un mateix o amb l'ajut d'altres entesos. Fins que no s'arriba a una solució satisfactòria, no es comença la feina. En el cas d'un problema d'una disciplina complexa, evidentment, cal anar una mica més enllà i no n'hi ha prou només amb un croquis *ad hoc*. Els plànols d'un arquitecte per fer un gratacels no són simples esbossos. Cal un mecanisme formal per descriure el problema i la solució, de manera que tota persona implicada pugui plasmar les seves idees de manera que tothom l'entengui. El desenvolupament d'aplicacions no és una excepció.

Un dels avantatges més importants de la metodologia de l'orientació a objectes, i el principal motiu que l'ha fet rellevant, és que ha permès establir un llenguatge unificat per representar dissenys: el llenguatge UML (*unified modelling language*). D'aquesta manera, el desenvolupament del programari es pot equiparar a la resta de disciplines tècniques, en les quals és possible generar un esquema d'allò que es vol crear, que pot ser interpretat per qui s'encarregarà de fabricar-ho, sense necessitat que el constructor hagi format part del procés de disseny. A més a més, també és possible que altres experts externs avaluïn l'esquema abans de la fabricació, de manera que es puguin detectar errades abans de consumir cap recurs. Aquesta possibilitat és molt valuosa quan es tracta de crear sistemes complexos.

En el cas concret del desenvolupament del programari, quan parlem de constructor ens referim a un programador.

L'**UML** és un llenguatge estàndard que permet especificar amb notació gràfica programari orientat a objectes.

Els orígens de l'UML es remunten a l'any 1994, quan Grady Booch i Jim Rumbaugh van començar a unificar diferents tècniques de modelització orientada a objectes, en un intent de trobar una solució satisfactòria als problemes que els dissenyadors tenien a l'hora d'especificar el programari. Al llarg d'aquest procés, s'hi va unir Ivar Jacobson. Cadascú va aportar el propi mètode: el mètode Boosch, OMT (*object modelling technique*, 'tècnica de modelització d'objectes') i OOSE (*object oriented software engineering*, 'enginyeria de programari orientada a objectes').

L'UML no està limitat al programari, sinó que també permet especificar altres sistemes com, per exemple, models de negoci.

Els motius principals que els van dur a unir esforços van ser tres:

- Evitar que cada mètode evolucionés independentment, cosa que portaria a una situació amb molts mètodes heterogenis que seria perjudicial per als dissenyadors.

- Establir una notació única que aportés certa estabilitat al camp de l'orientació a objectes. Això era molt important per tal que el diferent programari de suport a la metodologia fos compatible entre si.
- Trobar millores mitjançant la col·laboració entre ells, partint de l'experiència individual en la creació de cada mètode.

L'OMG és una organització sense ànim de lucre amb l'objectiu d'impulsar les tecnologies basades en l'orientació a objectes.

L'any 1996 van fer la seva proposta d'UML 0.9, subjecta a l'escrutini i els comentaris de la comunitat, la qual es va convertir en objectiu estratègic de diverses organitzacions, com Digital, Hewlett-Packard, Oracle o Microsoft. Els esforços d'aquestes organitzacions van ser canalitzats per l'Object Management Grup (OMG, Grup de Gestió d'Objectes), que va donar lloc a l'UML 1.0. En els anys següents, l'UML ha seguit evolucionant i se n'han fet noves versions, que es continuen utilitzant dins el procés de disseny d'aplicacions.

1.1 Esquema general de resolució de problemes mitjançant orientació a objectes

A nivell general, els passos ordenats per resoldre un problema usant orientació a objectes es poden resumir en:

1. Plantejar l'escenari descriptivament, amb llenguatge humà. Com més detallada sigui la descripció, més fàcil serà la feina.
2. Localitzar, dins la descripció de l'escenari, els elements que es consideren més importants: els que realment interactuen amb vista a resoldre el problema. Aquests seran els objectes del programa. Normalment, solen ser substantius dins la descripció.
3. Considerar quins elements són d'una certa complexitat. Redefinir-los com a agrupacions d'objectes més simples. Una bona estratègia és partir del fet que tot l'escenari en si és un objecte complex (igual que una màquina també és un objecte complex) i anar-lo descomponent en parts més petites.
4. Agrupar els diferents objectes segons el tipus: quins objectes veiem que tenen propietats o comportaments idèntics. Cada tipus d'objecte serà una classe que caldrà especificar.
5. Identificar i enumerar les característiques dels objectes de cada classe: quines en són les propietats (els atributs) i el comportament (les operacions que ofereixen). N'hi ha prou amb una llista general, escrita en llenguatge humà però suficientment entenedora.
6. Establir les relacions que hi ha entre els objectes de les diferents classes a partir del paper que interpreten en el problema general. Els objectes no es generen en un buit, sinó que estan relacionats entre si, de la mateixa manera que les peces d'una màquina o els elements d'un edifici no floten en l'aire, sinó que estan connectats per formar un tot. De la mateixa manera,

un cop identificats els objectes que conformen el problema a resoldre (el programa que es vol fer, en aquest cas), cal identificar com es relacionen entre ells. Normalment, aquesta mena d'enllaços es pot identificar com "un objecte d'aquest tipus conté objectes d'aquesta altra classe" o "una instància d'aquesta classe gestiona instàncies d'aquesta altra classe". A mode d'ajut, es pot generar un **mapa d'objectes**.

7. Per a cada classe, especificar-ne formalment els atributs i les operacions, extrets a partir de la llista de propietats dels seus objectes dels punts 5 i 6. Normalment, especificar-ne els atributs és un procés més immediat que l'especificació de les operacions.

UML ofereix una notació formal per poder plasmar tots els elements resultants de cadascun dels passos: classes, atributs, mètodes, relacions entre objectes... El resultat final d'aquest procés seguint la notació UML és un esquema anomenat *diagrama estàtic UML.*

> Un **diagrama estàtic UML** mostra el conjunt de classes i objectes importants que formen part d'un sistema, juntament amb les relacions existents entre aquestes classes i objectes. Mostra d'una manera estàtica l'estructura d'informació del sistema i la visibilitat que té cadascuna de les classes, donada per les seves relacions amb les altres en el model.

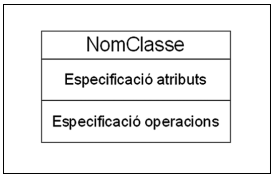
Mitjançant UML també és possible definir formalment molts altres aspectes del comportament de l'aplicació (comportament dinàmic, relacions entre crides a mètodes...), però aquí ens centrarem en el diagrama estàtic, que és la base de tot.

1.2 Definició de classes mitjançant UML

La peça fonamental d'un diagrama estàtic són les classes, ja que, de fet, el codi d'un programa orientat a objectes es compon de classes, on es defineixen els atributs i els mètodes de què disposaran les seves instàncies (els objectes) quan el programa s'executi. Per tant, abans de poder crear un diagrama complex, el primer pas és identificar-les i saber com representar-les en UML.

En UML, una classe es representa en format complet mitjançant una caixa dividida horitzontalment en tres parts. La part superior compleix exactament la mateixa funció i té el mateix format que en el format simplificat, i és on consta el nom de la classe. En la part del mig es defineixen els atributs que tindran les seves instàncies. Finalment, en la part inferior, es defineixen les operacions que es poden cridar sobre qualsevol de les seves instàncies. L'aspecte és el que es mostra en l'exemple de la figura 1.1.

FIGURA 1.1. Notació formal d'una classe en UML.



1.2.1 Definició d'atributs

En definir una classe, els atributs s'especifiquen segons la sintaxi següent. El camp de valor inicial es correspon al valor que pren l'atribut en el moment d'instanciar un objecte d'aquesta classe. Concretar-lo en l'especificació dels atributs és opcional. Com veieu, el format és semblant a la declaració d'una variable qualsevol en Java.

```
1 visibilitat nomAtribut: tipus [= valor inicial]
```

Nomenclatura

Per a atributs s'usen paraules concatenades, en què la primera inicial està amb minúscula i la resta amb majúscula. Per exemple: `elMeuAtribut`.

Els dos aspectes més importants de la definició d'un atribut en representar una classe en UML són la seva visibilitat i el seu tipus de dades.

Pel que fa a la seva visibilitat, aquesta indica si l'atribut és accessible directament des d'altres classes. Hi ha diferents tipus de visibilitat, si bé es destacaran els dos més utilitzats: la visibilitat pública i la privada. Tot i que és possible triar entre diverses, normalment, els atributs es defineixen amb visibilitat privada.

L'UML no indica explícitament quin és el significat real de cada tipus de visibilitat, i deixa aquesta tasca a cada llenguatge de programació. El motiu és que aquest terme es refereix a l'accessibilitat a un objecte en l'àmbit del codi. Tot i així, es descriurà quina sol ser la seva interpretació en la majoria de llenguatges de programació orientats a objectes. Cada tipus de visibilitat s'identifica en la definició de l'atribut amb un símbol especial:

- Un **atribut públic** s'identifica amb el símbol "+". En aquest cas, si una instància `a:` està enllaçada amb una instància `b:`, `a:` pot accedir lliurement als valors emmagatzemats en els atributs de `b:`.
- Un **atribut privat** s'identifica amb el símbol "-". No es pot accedir a aquest atribut des d'altres objectes, independentment del fet que existeixi un enllaç o no. A efectes pràctics, és com si no existís fora de l'especificació de la classe i, en conseqüència, només es pot utilitzar en les operacions dins la mateixa classe en què s'ha definit.

Quant al tipus de dades, donat que UML és un mecanisme de notació genèric, no vinculat a cap llenguatge específic de programació, els tipus de dades que es poden usar no són exactament els mateixos que en Java, si es vol ser estricte. Se sol usar el llenguatge natural, en anglès, per indicar el tipus, enlloc d'una paraula clau concreta depenent del llenguatge. Tot seguit s'enumeren els més típics si es fa en català:

TAULA 1.1. Tipus bàsics dels atributs.

Tipus	Significat	Exemple
Enter	Un nombre sense decimals	1, 56, 128, 15487
Real	Un nombre amb decimals	1,34, 3,2415, 267,14, 41,0
Caràcter	Una lletra	A, a, b, g, -, ?, ç
Booleà	Cert/fals	Cert, fals
Byte	Un byte	0x30, 0xA2, 0xFF
Matriu de... (...[ ])	Un conjunt d'elements... [1, 2, 3], [a, b, c, f, g], [1,2, 3,0]	

En el darrer cas, els tipus múltiples es poden especificar de dues maneres diferents, segons la interpretació que es vol expressar:

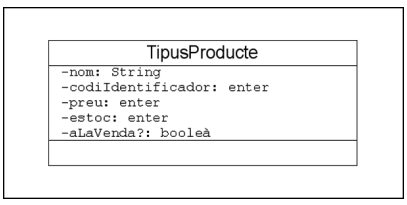
- enter[5] indica que hi ha exactament cinc enters.
- enter[0..5] indica que hi pot haver entre zero i cinc enters.

De totes maneres, també es pot considerar que ja hi ha predefinits un conjunt de tipus de propòsit general, que pràcticament tots els llenguatges suporten d'una manera o d'una altra:

- `String`, per especificar tipus de dades que corresponen a cadenes de caràcters, així s'evita haver d'operar amb caràcters.
- `List<nomTipus>`, per especificar seqüències d'elements de tipus "nomTipus", sense cap fita predeterminada. En el camp "nomTipus" s'indica el tipus d'element que conté la llista, per exemple:List<enter>, List<Cita>, List<String> ...

Així, doncs, els atributs d'una classe anomenada `TipusProducte` es poden definir tal com mostra la figura 1.2.

FIGURA 1.2. Definició d'atributs de la classe TipusProducte



Els atributs de classe són especialment útils per definir constants.

A part dels atributs que defineixen les propietats de cada instància d'una classe, hi ha un tipus especial d'atributs, anomenats **atributs de classe** o estàtics. A l'hora de definir-los, es diferencien amb un subratllat i, si són constants, normalment estan escrits amb el nom en majúscules. A part d'això, la sintaxi és idèntica als atributs genèrics.

```
1 visibilitat nomAtributClasse: tipus [= valor inicial]
```

Per exemple:

```
+PI: real
```

1.2.2 Definició de mètodes

La convenció de nomenclatura d'una operació és idèntica a la dels atributs.

Dins la definició d'una classe, les operacions disponibles s'especifiquen en UML de la manera següent:

```
visibilitat nomOperació (llistaParàmetres): tipusRetorn
```

El camp "llistaParàmetres" té el format següent:

```
nomParàmetre1: tipus, ... , nomParàmetreN: tipus
```

Recordeu que el "+" abans del nom indica que les operacions tenen visibilitat pública.

Totes les explicacions donades per als tipus o la visibilitat en el cas dels atributs també són aplicables a les operacions. En aquest cas, normalment les operacions solen ser públiques, i es deixen com a privades operacions que es consideren auxiliars, o que simplement faciliten la comprensió de les tasques que duran a terme les instàncies de les classes, però que no es poden invocar des de cap altra part del futur codi.

Alguns exemples d'especificacions d'operacions poden ser:

```
+afegirMèdia (m: Mèdia)
+ajustarVolum (v: enter)
+pausa/reanuda ()
+mèdiaSegüent(): Mèdia
```

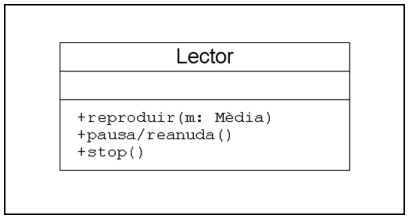
Addicionalment, hi ha un conjunt d'operacions que no sempre cal especificar, ja que se suposen en dissenyar una classe: les operacions accessores. S'acostumen a considerar operacions o mètodes accessors els que donen accés de lectura o d'escriptura als atributs d'una classe. La nomenclatura estàndard per a l'accessor d'escriptura (per modificar el valor de l'atribut) i de lectura (per consultar-lo) és, respectivament:

- setNomAtribut (valor: tipus).
- getNomAtribut(): tipus.

Per tant, en l'especificació d'una classe no cal explicitar tots els accessors entre les seves operacions. Per a cada atribut especificat ja es dóna per entès que sempre hi ha les operacions *set* i *get* associades, a menys que es digui el contrari.

La figura 1.3 mostra una representació UML de les operacions d'una classe anomenada `Lector`.

FIGURA 1.3. Especificació d'operacions de la classe Lector.



1.3 Relacions entre classes

Un cop identificades les classes dels objectes que componen el problema a resoldre, el pas següent és establir quines relacions hi ha entre elles i representar-les dins el diagrama estàtic UML. De fet, les relacions són el punt especialment rellevant d'un diagrama estàtic UML, ja que, en cas contrari, les classes queden representades en un buit, disperses. Només es disposa d'una llista de classes i prou, sense cap idea de com col·laboren entre elles o quin sentit tenen dins l'aplicació futura. En el diagrama estàtic, cada relació indica que hi ha una connexió entre els objectes d'una classe i els d'una altra. Fent un símil amb una màquina, l'existència d'una relació és equivalent al fet que dues de les seves peces estiguin connectades entre elles.

1.3.1 Associacions

El tipus de relació més freqüent és l'associació.

> Es considera que hi ha una **associació** entre dues classes quan es vol indicar que els objectes d'una classe poden cridar operacions sobre els objectes d'una altra.

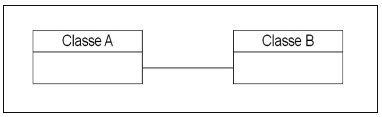
Per tant, perquè dos objectes puguin interactuar hi ha d'haver una associació entre les seves classes al diagrama estàtic UML, de manera que es considera que estan **enllaçats**. En cas contrari, la crida d'operacions no és possible. Quan per mitjà d'un enllaç un objecte **objecteA:** crida una operació sobre un objecte **objecteB:**, les transformacions fetes per l'operació únicament afectaran l'objecte **objecteB:**. No n'afectaran cap de la resta d'instàncies que hi hagi en aquell moment que pertanyin a la mateixa classe que l'**objecteB:**.

Eines CASE

Normalment, els diagrames estàtics UML es generen mitjançant l'ajut d'eines CASE (*computer aided software engineering*, ,enginyeria de programari assistida per ordinador').

El diagrama estàtic UML estarà format en la seva totalitat per la representació gràfica de totes les classes identificades i les relacions que hi ha entre totes elles. En la figura 1.4 es representen les relacions amb una línia que uneix les classes implicades.

FIGURA 1.4. Associació entre classes.



Tot i que les associacions indiquen que hi ha enllaços entre objectes, es representen gràficament en el diagrama respecte a les seves classes. Quan dues classes es representen relacionades, les seves instàncies poden estar enllaçades en algun moment de l'execució de l'aplicació.

Els enllaços són dinàmics. Poden variar al llarg de l'execució de l'aplicació.

Donada una associació, hi ha un conjunt de descriptors addicionals que es poden especificar:

- En el centre, entre les dues classes implicades, s'especifica el **nom de l'associació**. Aquest nom indica què representa l'associació a nivell conceptual.
- En cada extrem de l'associació s'especifica quines són les funcions de les classes implicades. El nom indica, de manera entenedora, el paper que tenen els objectes de cada classe en la relació.
- Amb una fletxa se n'especifica la **navegabilitat**. Partint del nom de l'associació i les funcions de les classes, s'ha de poder establir quina és la classe origen i quina la destinació. A efectes pràctics, la navegabilitat indica el sentit de les interaccions entre objectes: a quina classe pertanyen els objectes que poden cridar operacions i a quina els objectes que reben aquestes crides.
- Juntament amb la funció, s'especifica la **cardinalitat**, o multiplicitat, de la relació per a cada extrem. La cardinalitat especifica amb quantes instàncies d'una de les classes pot estar enllaçada una instància de l'altra classe.

Si cal, res no impedeix que una associació sigui **navegable** en ambdós sentits, tot i que no és el cas més freqüent. En aquest cas, no cal posar cap fletxa.

La **cardinalitat** defineix quantes instàncies diferents d'una classe ClasseA es poden associar amb una instància de la classe ClasseB en un moment determinat de l'execució del programa.

Les cardinalitats més usades solen ser les 1, 1..* i *.

El nombre d'instàncies per a cada cas s'indica amb una llista de nombres enters, ubicada en l'extrem oposat de l'associació. Per exemple, la cardinalitat que indica quantes instàncies de la ClasseB pot enllaçar una instància de la ClasseA s'ubica en l'extrem de l'associació en què està representada la classe ClasseB. En cas que es vulgui indicar un nombre indeterminat, sense fita superior en el nombre d'enllaços, s'utilitza el símbol *. Per establir rangs de valors possibles, s'usen les fites inferior i superior separades amb tres punts.

Diferents cardinalitats i el seu significat

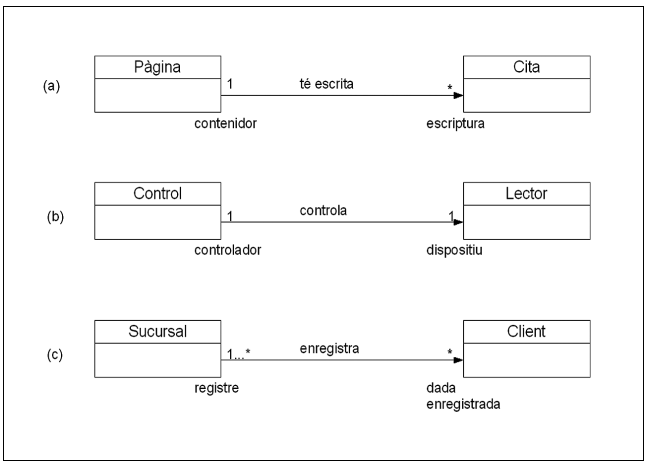
- 1: Només un enllaç.
- 0...1: Cap o un enllaç.
- 2,4: Dos o quatre enllaços.
- 1...5: Entre un i cinc enllaços (un, dos, tres, quatre o cinc).
- 1...*: Entre un i un nombre indeterminat. És a dir, més d'un.
- *: Un nombre indeterminat. És equivalent a 0...*.

Perquè el programa es consideri correcte a l'hora d'implementar el disseny, s'ha de fer el necessari perquè les condicions que expressen les cardinalitats sempre siguin certes. Si una cardinalitat és, per exemple, 1, això vol dir que tot objecte d'aquesta classe sempre està enllaçat amb un, i només un, objecte de l'altra classe. Mai no hi pot haver dins el programa en execució un cas en què no es compleixi aquesta condició.

De tots aquests descriptors, només la navegabilitat i la cardinalitat són imprescindibles (i de les dues, la cardinalitat és la més important), ja que la decisió que es prengui en aquests aspectes dins l'etapa de disseny tindrà implicacions directes sobre la implementació. Les funcions i el nom són opcionals, si bé molt útils amb vista a la comprensió del diagrama estàtic i com a mètode de reflexió per al dissenyador.

En la figura 1.5 es mostra una representació completa d'una associació, amb tota la informació que cal especificar.

FIGURA 1.5. Representació gràfica d'una associació entre classes.



A partir del que expressa la figura, aquestes són algunes de les conclusions a les quals arribaria un observador aliè al procés de disseny:

- **A partir d'(a)**. Una instància qualsevol de la classe `Pàgina` pot enllaçar fins a un nombre indeterminat d'objectes diferents de la classe `Cita`. Això inclou no tenir-ne cap (una pàgina en blanc). Així, doncs, hi ha pàgines que tenen tres cites, d'altres deu, d'altres cap, etc.

- **A partir d'(a)**. Recíprocament, donat un objecte qualsevol de la classe `Cita`, només estarà enllaçat a un únic objecte `Pàgina`. Per tant, no es pot tenir una mateixa cita en dues pàgines diferents (però sí tenir dues citacions diferents i de contingut idèntic, amb els mateixos valors per als atributs, cadascuna a una pàgina diferent). Tampoc no hi pot haver citacions que, tot i ser en l'aplicació, no estiguin escrites a cap pàgina.

- **A partir de (b)**. Un objecte de la classe Control sempre té un objecte de la classe Lector enllaçat. Per tant, un tauler de control sempre controla un únic lector de mèdia. No es pot donar el cas que un tauler de control no controli cap lector. La inversa també es certa; tot lector és controlat per algun tauler de control.

- **A partir de (b)**. Donada la navegabilitat especificada, s'està dient que el tauler de control pot cridar operacions sobre el lector, però no a l'inrevés. Això té sentit, ja que és el tauler de control qui controla el lector.

- **A partir de (c)**. Donat un client, aquest pot estar enregistrat en més d'una sucursal, però almenys sempre ho estarà en una. Mai no es pot donar el cas d'un client donat d'alta en el sistema però que no estigui enregistrat en cap sucursal.

Cada objecte és únic i té la seva pròpia identitat, independentment dels valors dels seus atributs.

Tot i que no és habitual, res no impedeix que dues classes tinguin més d'una associació entre elles. Normalment, aquest cas s'aplica quan es vol fer una diferenciació explícita de diferents tipus de relació o de funcions entre instàncies de cada classe. En la figura 1.6 es mostra un exemple en què es pot aplicar aquest principi.

FIGURA 1.6. Múltiples associacions entre dues classes.

Atès que un transportista pot adoptar diferents papers en la seva relació amb una sucursal, això es pot representar especificant que hi ha diferents tipus d'associació entre sucursals i transportistes.

Tot i la versatilitat dels diagrames estàtics UML, aquests no sempre són capaços de reflectir totes les restriccions necessàries en els enllaços entre objectes de diferents classes. Les associacions indiquen els possibles enllaços, però no especifiquen condicions concretes per a la seva presència. En aquests casos s'utilitza una **restricció textual**. Aquesta no és més que una frase addicional al peu del diagrama, escrita en llenguatge humà, en què s'explica breument en què consisteix aquesta restricció.

L'*object constraint language* és un mecanisme formal per expressar restriccions textuals sense usar el llenguatge humà.

Els transportistes

Amb les associacions representades en la figura 1.6 és possible expressar que sempre hi ha un únic transportista de reserva i diversos de servei. Malauradament, és impossible

indicar que un transportista concret, per exemple el transportista2:, no pot ser alhora de reserva i de servei.

Per resoldre aquest problema cal afegir una restricció textual al peu del diagrama que expliqui la condició en la qual es pot complir l'associació "de reserva": un transportista de reserva no pot estar de servei i viceversa.

1.3.2 Agregacions

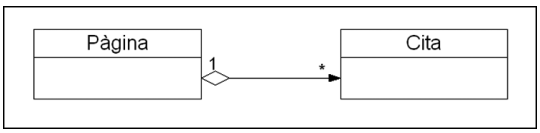
Hi ha un tipus d'associació especial mitjançant la qual el dissenyador vol donar un sentit més específic a l'enllaç, com ara que els objectes de certa classe formen part dels objectes d'una altra. Aquest subtipus d'associació s'anomena *agregació*.

Una **agregació** és un tipus d'associació especial que especifica explícitament que hi ha una relació de tot-part entre els objectes de l'agregat (el tot) i el component (la part).

En el diagrama estàtic UML, això es representa gràficament afegint un rombe blanc en l'extrem de l'associació on hi ha la classe que representa el tot. Com que amb aquest símbol ja es diu quina és la relació entre els objectes d'ambdues classes, es poden ometre el nom i la funció en els descriptors de l'associació.

Això es mostra en la figura 1.7, en la qual es refina l'associació PàginaCita que s'ha mostrat en els exemples anteriors. Una pàgina conté citacions escrites al seu interior i es pot considerar que el que s'ha escrit en una pàgina és part d'aquesta. Per tant, aquest cas és aplicable.

FIGURA 1.7. Agregació Pàgina-Cita



De totes maneres, val la pena esmentar que la definició d'aquest tipus d'associació no és gaire estricta, de manera que en la literatura hi ha diverses interpretacions de què vol dir realment *conté* o és *part de*. En darrera instància, és una interpretació personal del dissenyador considerar si un tipus d'objecte és realment part d'un altre.

1.3.3 Composicions

Un altre subtipus d'associació amb una semàntica especial són les anomenades *composicions*. Amb aquestes associacions també s'expressa el concepte de *és part de*, però anant més enllà: es considera que la classe composta no té sentit

Qualsevol associació en què el dissenyador posaria un verb de l'estil *té*, *conté*, *és part de*, etc. és una agregació o una composició.

sense els seus components. En contraposició, en una agregació, l'agregat sí que té sentit sense cap dels seus components.

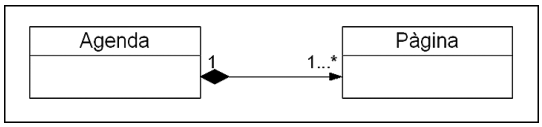
Una **composició** és una forma d'agregació que requereix que els objectes components només pertanyin a un únic objecte agregat i que, a més a més, aquest darrer no té sentit que existeixi si no n'existeixen els components.

En l'agregació , segons la mateixa definició, la cardinalitat en la banda de l'agregat sempre ha de ser 1.

Cal anar amb compte, ja que de vegades és fàcil confondre agregació i composició.

Aquest tipus d'associació es representa de manera idèntica a una agregació, només que en aquest cas el rombe és de color negre. La figura 1.8 en mostra un exemple amb l'associació Agenda-Pàgina.

FIGURA 1.8. Agregació Agenda-Pàgina Agenda-Pàgina.



Un altre exemple immediat de composició són les associacions que sortirien de en el reproductor multimèdia.

En aquest cas, en usar una composició per representar aquesta associació, el dissenyador expressa que no té sentit una agenda sense pàgines. De fet, conceptualment, el propi conjunt de pàgines és l'agenda en si. Tampoc no pot ser que una mateixa pàgina (es parla de la mateixa pàgina física, no d'una rèplica) sigui de més d'una agenda alhora. En canvi, sí que té sentit una pàgina en blanc sense cap cita, motiu pel qual el cas Pàgina-Cita és una agregació però no una composició.

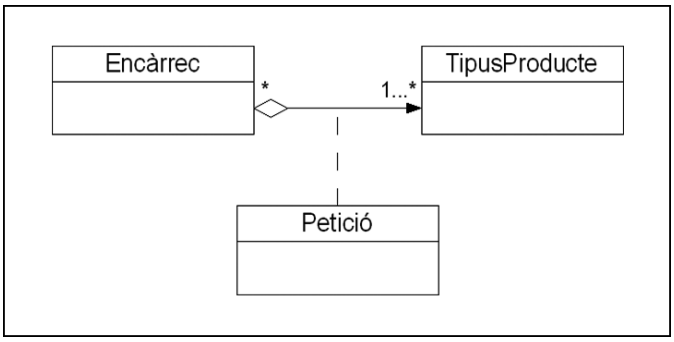
1.3.4 Classes associatives

Hi ha circumstàncies que fan que el dissenyador consideri necessari afegir propietats addicionals a una associació, el valor de les quals pot variar segons quines siguin les instàncies enllaçades. En definitiva, el dissenyador vol especificar atributs en una associació. Amb aquesta finalitat s'usen les *classes associatives*.

Les **classes associatives** representen associacions que es poden considerar classes.

Una classe associativa es representa amb el mateix format que una classe: és una nova classe que cal especificar dins la descomposició del problema. Com es mostra en la figura 1.9, una línia discontinua uneix la nova classe amb l'associació que representa.

FIGURA 1.9. Exemple de classe associativa



En la classe associativa Petició hi hauria les propietats vinculades a la petició d'un tipus de producte concret dins un encàrrec (per exemple, el nombre de productes que cal enviar o el seu color). Cap d'aquestes propietats no correspon a l'encàrrec ni al tipus de producte. Tot això es plasmarà en forma d'atributs dins d'aquesta classe.

Totes les classes associatives que apareguin en el diagrama estàtic UML s'hauran d'especificar completament, tant pel que fa als atributs com a les operacions. Tot i representar una associació i no un element identificable dins del problema, a efectes pràctics són una classe més, com qualsevol altra.

Una particularitat de les classes associatives és que es poden representar amb classes i associacions normals. Aquest fet és important, ja que és l'única manera de representar-les en un mapa d'objectes i la majoria de llenguatges de programació no suporten les associacions amb aquest tipus de classes vinculades.

Java no suporta directament classes associatives.

Seguint el sentit de la navegabilitat (classe origen - classe destinació) el desenvolupament és el següent:

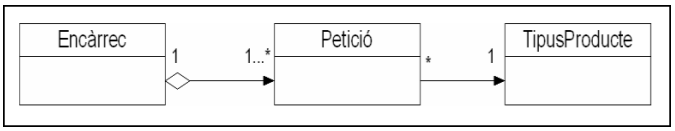
1. Eliminar l'associació original.
2. Generar una associació de la classe origen a la classe associativa. El tipus de la nova associació i la navegabilitat són idèntics a l'original.
3. Generar una altra associació de la classe associativa a la destinació. La navegabilitat és de classe associativa a destinació.

Ara la classe associativa ja és una classe normal dins el diagrama estàtic UML, però atès que ara hi ha dues associacions, cal adaptar-hi les cardinalitats:

- La cardinalitat en els extrems oposats a l'antiga classe associativa sempre és 1.
- La cardinalitat en l'extrem oposat de la classe origen, en què ara hi ha la classe associativa, és la que hi havia respecte a l'antiga classe destinació.
- La cardinalitat en l'extrem oposat de la classe destinació, en què ara hi ha la classe associativa, és la que hi havia respecte a l'antiga classe origen.

El resultat d'aplicar aquesta traducció es mostra en la figura 1.10.

FIGURA 1.10. Traducció de classe associativa.



De fet, depenent de les interpretacions que ha fet el dissenyador respecte a la descripció del problema, es pot arribar a aquest diagrama directament des del pas d'identificació de classes. De totes maneres, en fer un diagrama estàtic UML, és recomanable usar classes associatives sempre que apliqui i només fer la traducció en el moment d'implementar.

1.3.5 Associacions reflexives

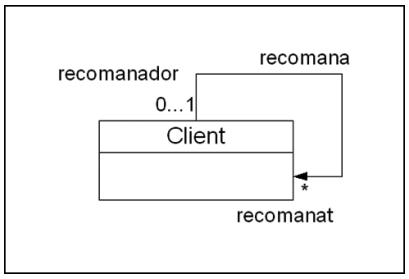
Atès que una associació indica enllaços entre instàncies d'una classe, res no impedeix que un objecte estigui enllaçat amb objectes del mateix tipus. Quan això passa, es representa mitjançant una associació reflexiva.

Per la seva definició, no es pot aplicar una associació reflexiva a una composició.

Una **associació reflexiva** és aquella en què la classe origen i la destinació són la mateixa.

Un exemple d'aquest cas es mostra en la figura 1.11, en què els clients de l'aplicació de gestió recomanen altres clients. Es tracta d'una associació reflexiva, ja que tant qui recomana com qui és recomanat, un client, pertanyen a la mateixa classe.

FIGURA 1.11. Associació reflexiva



Un dels moments en què el dissenyador ha d'anar amb més compte en aquest tipus d'associacions és quan n'especifica la cardinalitat. La cardinalitat a l'origen de la relació sempre ha de preveure el cas 0. En cas contrari, és impossible generar un mapa d'objectes que la satisfaci, ja que s'hi genera un bucle infinit. Això s'ha de tenir en compte, independentment del fet que, conceptualment, afegir-hi cardinalitat 0 tingui sentit o no.

Exemple de bucle infinit: un arbre genealògic

Un exemple d'aquesta problema és l'ús d'una associació reflexiva entre objectes d'una classe `Persona` per crear un arbre genealògic. Una persona sempre ha nascut a partir de dos pares (pare i mare), que a la vegada també són persones. Per tant, és lògic, i correcte

des del punt de vista conceptual, que la cardinalitat en origen sigui 2, i en destinació, * (una persona pot tenir un nombre indeterminat de fills).

Malauradament, si es fa així, és impossible generar un mapa d'objectes que compleixi aquesta cardinalitat. Per a cada objecte :Persona hi ha d'haver dos enllaços provinents d'altres objectes:Persona, i això és impossible tret que hi hagi cicles (cosa que no pot ser en un arbre genealògic). L'única manera d'evitar-ho és establir que la cardinalitat pot ser 0 o 2.

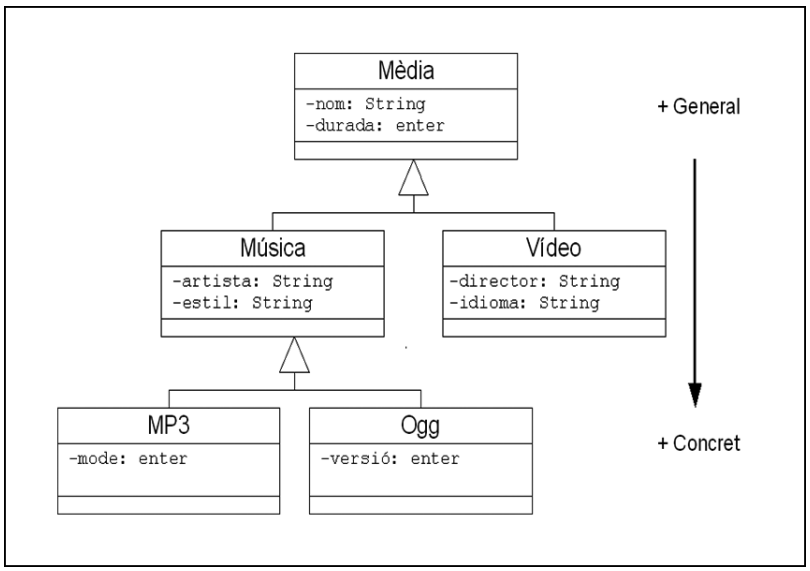
El cas 0 serà una excepció per als objectes :Persona inicials dins el sistema (per als quals no s'enregistrarà quins van ser els pares).

1.3.6 Herència

Finalment, dins un diagrama estàtic UML també s'estableixen les relacions d'herència entre els objectes de diferents classes. Això permet definir relacions d'especialització/generalització, on la classe a la part superior de la relació representa un concepte més general que el de la part inferior, més específic. Aquest tipus de relació normalment també indica que la classe més específica és idèntica a una altra, excepte en alguns petits aspectes on, o bé és diferent, o bé se suposen propietats addicionals.

Per exemple, diferents tipus de fitxers de mèdia es poden vincular entre si d'acord a una relació d'herència tal com mostra la figura 1.12. A la dreta es veu quin concepte es considera més general i quin més específic.

FIGURA 1.12. Herència dins un diagrama estàtic UML



1.4 Exemples de diagrames estàtics

Una vegada s'ha efectuat el procés de descomposició de programes es pot fer un possible diagrama estàtic UML. En cada cas, el diagrama segueix cadascuna de

les composicions proposades del problema. Per fer més entenedora la lectura dels diagrames, només s'inclou el nom de les relacions menys evidents.

Per a cada cas, també es representa un possible mapa d'objectes per a un moment determinat de l'execució de l'aplicació. És interessant veure com cada associació al diagrama estàtic es tradueix en un cert nombre d'enllaços entre objectes, sempre respectant la cardinalitat especificada en l'associació.

1.4.1 Una agenda

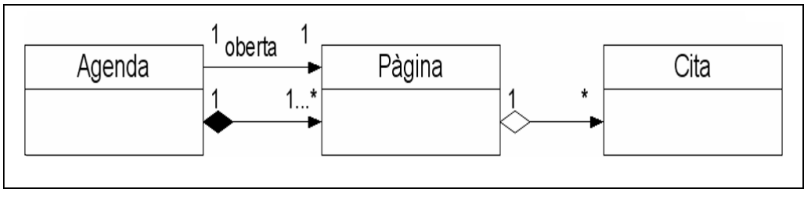
El primer exemple és molt senzill, amb molts pocs elements i funcionalitats i amb un paral·lelisme clar amb el món real per facilitar-ne la comprensió. Es vol dissenyar una agenda que permeti consultar les dates d'un calendari per a un any concret i apuntar cites a unes hores concretes. En aquest cas, és possible fer un cert paral·lelisme amb el món real, ja que el concepte d'agenda hi existeix. Es pot pensar en l'agenda com un llibre en què es van passant pàgines endavant o endarrere, cadascuna de les quals correspon a un dia. En cada pàgina es poden escriure cites establertes per a unes hores d'inici i de finalització determinades. Aquesta descripció en llenguatge humà seria la que s'ha descrit en el pas 1 de l'esquema d'aplicació de l'orientació a objectes.

Una bona manera de descompondre un problema en objectes és partir de l'element més general i, a partir d'aquí, anar extraient els elements més senzills. Així, doncs, en aquest cas es pot partir d'un objecte agenda, i deduir els elements que el componen. Una agenda es compon de les pàgines, les quals contenen cites.

Per saber exactament el tipus de relacions entre classes, cal establir la relació de dependència entre elles. Per això, també va bé fer-se una idea de si una situació té sentit o no si estiguéssiu tractant amb un objecte del món real. Així doncs, una agenda està formada per pàgines, i no té sentit una agenda sense pàgines, pel que s'està parlant d'una composició. D'altra banda, cal saber la pàgina oberta en tot moment, per la qual cosa cal una altra relació. Aquesta pot ser una simple associació, ja que no compleix cap de les característiques dels altres tipus de relacions. Finalment, les pàgines contenen cites, però sí que té sentit que una pàgina, en un moment donat, encara no tingui cap cita, per la qual cosa es tracta d'una agregació.

El diagrama estàtic UML de l'agenda es mostra en la figura 1.13.

FIGURA 1.13. Diagrama estàtic UML de l'agenda.



La instància d'Agenda pot tenir dos tipus de relacions amb les pàgines. D'una banda, la de composició: entre totes les pàgines formen l'agenda. A més a més, una agenda també sap per quina pàgina està oberta, que seria la pàgina que es pot llegir en aquest moment.

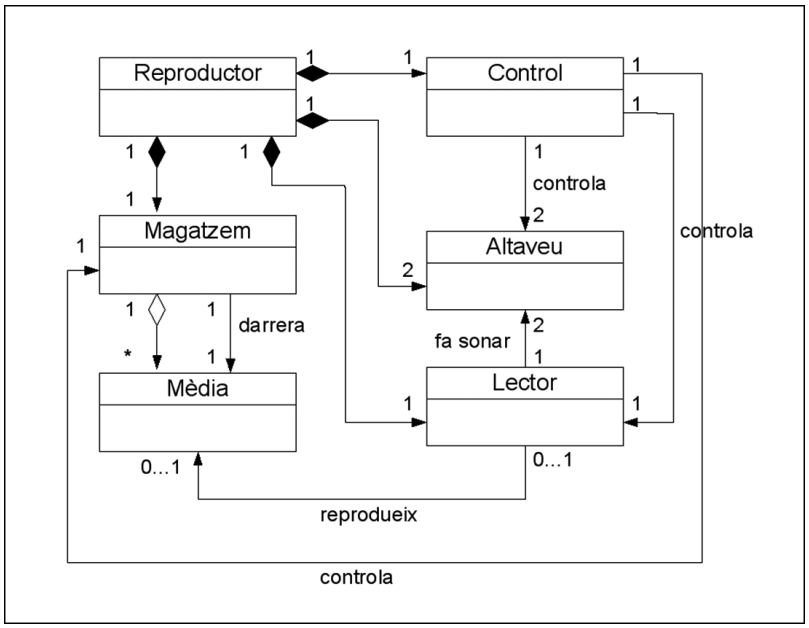
1.4.2 Un reproductor multimèdia

En aquest exemple es presenta la descomposició d'un sistema de reproducció multimèdia (música, vídeos...). La motivació pot ser crear una aplicació senzilla per a l'ordinador o generar el sistema de control d'un reproductor portàtil (un dispositiu físic). En aquest cas, es parteix de la identificació de classes següent: `Reproductor`, `Mèdia`, `Control`, `Lector`, `Magatzem` i `Altaveu`.

El diagrama estàtic UML del reproductor es pot veure en la figura 1.14. En fer-lo, s'ha decidit que el reproductor té dos altaveus, de manera que es pot controlar el mode mono o estèreo. També cal tenir en compte els punts següents:

- La cardinalitat de la relació `Mèdia-Lector` especificada en l'extrem de la classe `Mèdia` (a l'esquerra) indica que hi pot haver moments en què el lector no reprodueix cap cançó (cas 0 de la cardinalitat: el lector reprodueix 0 cançons).
- La cardinalitat en l'altre extrem expressa que, per a qualsevol cançó, o bé s'està reproduint en el lector (cas 1) o no (cas 0).
- El magatzem pot ser buit (cas 0 de la cardinalitat "*" en la relació `Magatzem - Mèdia`).
- S'ha decidit que la relació entre `Magatzem` i `Mèdia` és una agregació, però també es pot interpretar com una associació normal si no es considera que les dades emmagatzemades, les cançons, són part del magatzem i tenen entitat pròpia.
- La relació "darrera" entre `Magatzem` i `Mèdia` expressa quina és la darrera cançó a què s'ha accedit en el magatzem, per fer-ne accés seqüencial.

FIGURA 1.14. Diagrama estàtic UML del reproductor.

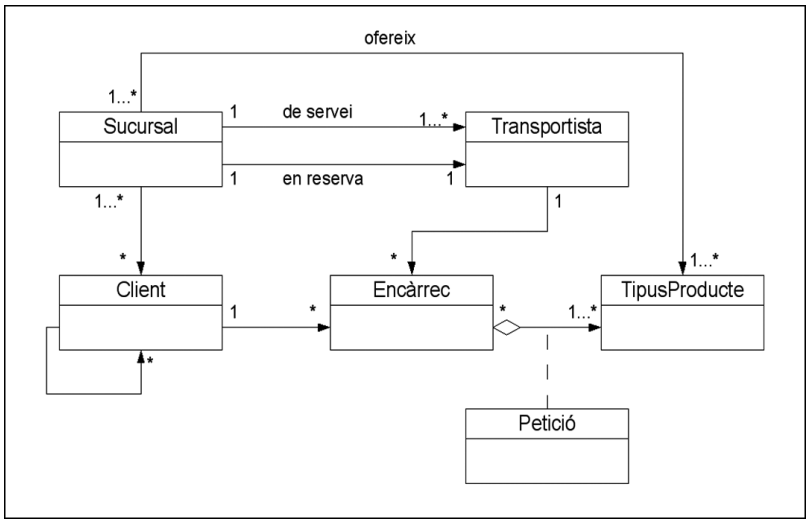


1.4.3 Una aplicació de gestió

Suposeu que una empresa vol crear una aplicació que gestioni el transport d'encàrrecs d'una sucursal d'una franquícia. Cada sucursal té un conjunt de transportistes assignats, el nombre dels quals pot variar segons la grandària de la sucursal. Cada dia hi ha un transportista que no treballa, però es considera que està en reserva. Cada un disposa del seu propi vehicle, identificat per un número de llicència. Quan un client vol fer un encàrrec, se n'enregistren les dades personals i especifica les condicions de lliurament: dia i hora, adreça... En l'encàrrec fa constar la llista de productes que vol que li serveixin. Tan bon punt es genera un encàrrec, automàticament, ja s'assigna algun transportista perquè el serveixi. Mai no hi ha encàrrecs sense transportista assignat. Els clients també tenen l'opció de recomanar amics seus perquè s'hi apuntin com a clients. Aquest fet es té en compte de cara a algunes promocions o descomptes especials.

En aquest cas, es partirà de la descomposició en les següents classes: `Encàrrec`, `Sucursal`, `Transportista`, `Client`, `TipusProducte`. El diagrama estàtic UML de l'aplicació de gestió és el següent, representat en la figura 1.15.

FIGURA 1.15. Diagrama estàtic UML de l'aplicació de gestió



1.5 Els diagrama estàtic i els mapes d'objectes

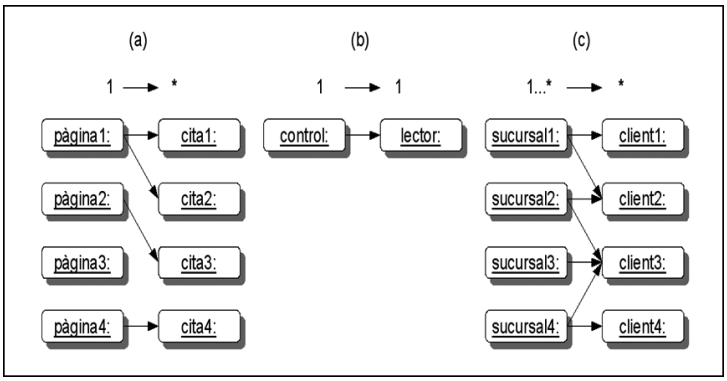
Una eina útil per reflexionar sobre si una cardinalitat representa allò que el dissenyador vol és crear mapes d'objectes. Es tracta d'esquemes que representen els diferents estats possibles de l'aplicació.

> En un **mapa d'objectes** es mostren tots els objectes instanciats i els enllaços que hi ha entre ells en un moment determinat de l'execució, l'aplicació d'acord amb el que s'ha representat en el diagrama estàtic UML.

Els mapes d'objectes només són una eina de suport, i no s'utilitzen com a mecanisme formal per representar el disseny. En el diagrama estàtic UML ja hi ha tota la informació necessària.

La figura 1.16 representa un seguit de mapes d'objectes, un per cada cas, amb diferents objectes enllaçats correctament segons la cardinalitat especificada en l'associació. Els enllaços es representen amb fletxes segons la navegabilitat de les associacions. El cas (a) correspon a l'associació PàginaCita, el cas (b) a la Control-Lector i el (c) a la Sucursal-Client. Damunt de cada mapa representat hi ha un recordatori de la cardinalitat de l'associació.

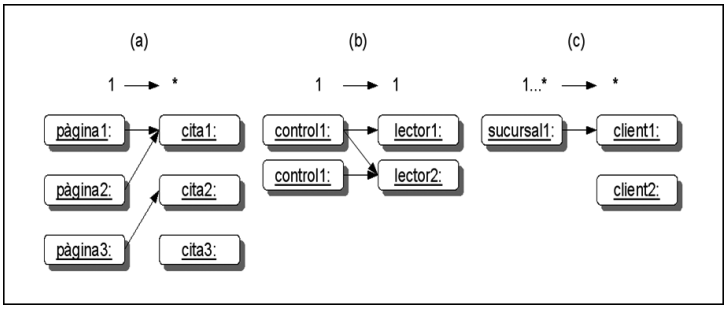
FIGURA 1.16. Exemple de mapes d'objectes



La figura 1.17 mostra alguns casos d'estats que es considerarien incorrectes segons les cardinalitats especificades en les associacions:

- **El cas (a)** és incorrecte, ja que una cita sempre ha d'estar escrita en alguna pàgina (cardinalitat esquerra = 1), i la instància cita3: no ho compleix, en no estar enllaçada a cap objecte :Pàgina. Només seria correcte si la cardinalitat fos, per exemple, 0..1.
- **El cas (b)** és incorrecte, ja que un tauler de control només controla un únic lector enllaçat (cardinalitat dreta = 1), i la instància control1: no ho compleix, en estar enllaçada a dues instàncies, lector1: ilector2:. A més a més, un lector només pot estar enllaçat a un únic tauler de control (cardinalitat esquerra = 1), i la instància lector2: tampoc no ho compleix.
- **El cas (c)** és incorrecte, ja que tot client ha d'estar enregistrat en alguna sucursal, i la instància client2: no està enllaçada amb cap.

FIGURA 1.17. Exemples d'enllaços incorrectes



2. Aplicacions amb BD no orientades a objectes

La persistència mitjançant fitxers és més que suficient si el tractament que es vol fer de les dades és totalment seqüencial: l'aplicació llegeix el fitxer en la seva totalitat i a partir del seu contingut genera el conjunt d'objectes que necessita per dur a terme la seva tasca. Per exemple, un editor estàndard de textos o gràfic. Però en cas que es vulgui fer un accés aleatori a diferents parts del fitxer, o quan el nombre de dades és realment molt gran, tant que carregar tot el fitxer superaria la memòria de l'ordinador, la utilitat d'aquest sistema cau en picat.

Una altra restricció molt important és que usar fitxers deixa de funcionar, o si més no, la seva gestió es fa massa complicada perquè realment valgui la pena, quan diverses aplicacions volen accedir concurrentment a les dades emmagatzemades. És per aquest motiu que la majoria d'aplicacions que necessiten gestionar una gran quantitat d'informació o poden existir accés concurrent utilitzen una base de dades per aconseguir la persistència.

Les files d'una BD relacional també s'anomenen tuples.

Una **base de dades (BD)** és un mecanisme per emmagatzemar informació de manera que sigui fàcil i eficient de recuperar. En la seva accepció més simple, la de base de dades relacional, aquesta pren la forma d'un seguit de taules formades per files i columnes.

Quan parlen de BD, ens referim sempre a una BD relacional.

Al contrari del que passava amb la persistència mitjançant fitxers o la seriació d'objectes, quan s'utilitza persistència mitjançant una BD, no es tracta de recuperar immediatament tots els objectes emmagatzemats i instanciar-los a memòria. Justament, una BD s'utilitza especialment quan hi ha massa objectes o diferents equips han d'accedir a les mateixes dades, pel que fer-ho no té sentit, ja que no soluciona el problema. La part del Model que es vol que sigui persistent sempre és en la BD, i quan s'instanciï un objecte, normalment serà per encapsular un conjunt d'informació recuperada de la BD per poder operar amb les seves dades de manera temporal.

BD

A part de les BD relacionals, també hi ha les orientades a objectes i les jeràrquiques, que estructuren les dades de manera diferent, en lloc de només taules.

També és important mantenir la consistència entre els objectes a memòria i la seva representació en la BD, si en algun moment es dóna el cas que durant l'execució de l'aplicació hi ha aquesta duplicitat. Si el valor d'algun atribut de la instància a memòria varia, tard o d'hora aquest nou valor s'ha de veure reflectit en la seva representació en la BD. !

La manera en què realment s'emmagatzema i s'accedeix a tota aquesta informació depèn de l'anomenat *sistema gestor de base de dades*, o *SGBD (database management system*, o DBMS, en anglès). L'aspecte que cal destacar d'aquest sistema és que és totalment transparent al desenvolupador d'una aplicació que accedeix a la BD. Ell s'encarrega de resoldre tots els aspectes vinculats a la integritat com, per exemple, que no es repeteixin claus primàries, o l'accés concurrent a les dades.

Les capacitats de cada SGBD i com funcionen internament depèn totalment de cada fabricant. De fet, força aspectes del mateix accés al sistema varien segons el fabricant, pel que només es pot fer una descripció genèrica en els aspectes més senzills. Pels aspectes vinculats al tipus de SGBD concret, aquest text es basa en la distribució de codi obert Apache Derby, desenvolupada totalment en Java.

2.1 Traducció del Model a una BD relacional

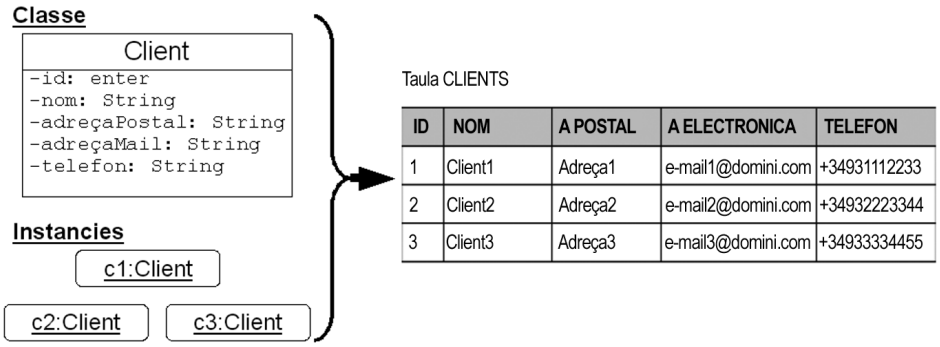
El fet que una BD estructuri la informació en forma de taules implica un procés de transformació des de la manera en què els objectes s'estructuren a memòria, el Model, a com s'organitzen en una BD:

- Cada classe instanciable del Model es materialitza en una taula en la BD.
- Cada atribut definit en una classe es materialitza en la BD com una columna dins la seva taula.
- La persistència de cada objecte del Model es materialitza en la BD en una fila dins la taula que correspon a la seva classe. En cada cel·la de la taula s'emmagatzema el valor que cada objecte en concret té assignat a l'atribut.

Hi ha una nomenclatura formal per representar gràficament BD. S'anomena model entitat-relació.

La figura 2.1 mostra un exemple senzill de traducció de classe a taula d'una BD. Cada fila correspon a la persistència de tres instàncies diferents de la classe Client.

FIGURA 2.1. Exemple de traducció de model a taula d'una base de dades.



En la BD hi ha tantes taules com tipus d'objectes es vol emmagatzemar. Quan es vol recuperar un objecte emmagatzemat, simplement se cerca en la taula corresponent per obtenir els valors dels seus atributs, de manera que si cal es pugui instanciar.

Per cercar un **objecte** concret en una taula, és molt important que sempre hi hagi algun atribut que sigui únic per a cada instància, de manera que no hi hagi cap ambigüitat. La columna que l'emmagatzema és el que es coneix com la **clau primària** d'una taula.

La necessitat de la clau primària està justificada pel fet que, en usar taules per emmagatzemar els objectes, aquests deixen de tenir referències que els identifiquin de manera única i mitjançant les quals s'hi pugui accedir. Per tant, cal afegir algun identificador únic que faci el mateix servei que una referència quan el Model es troba en la memòria. Normalment, s'escull un identificador de tipus enter, ja que ocupa poc espai i és fàcil de comparar.

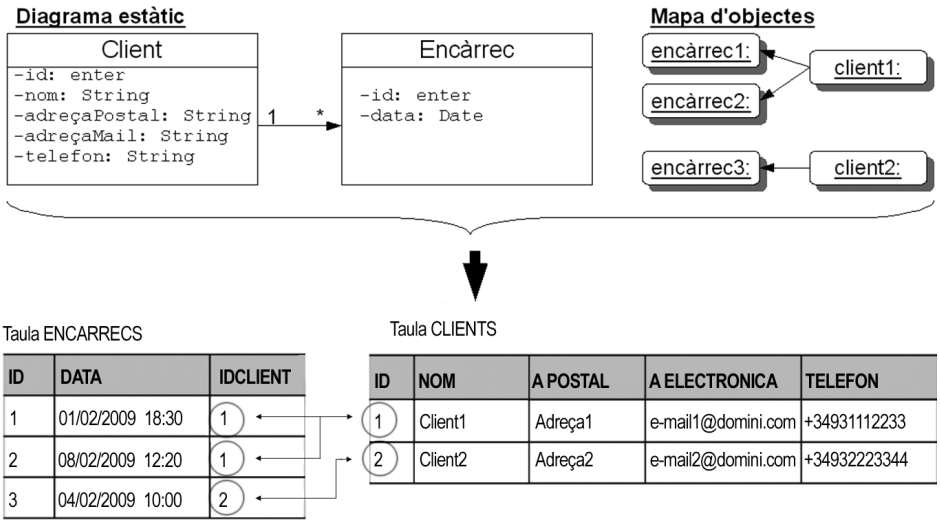
La desaparició de les referències en traspassar un model orientat a objectes a taules dins una BD també té un efecte especialment important en les associacions entre classes: els atributs que conformen llistes d'altres objectes. La traducció d'una associació varia segons la seva cardinalitat, però per tots els casos cal incloure els identificadors únics, les claus primàries, d'elements d'una taula en altres taules. Aquests identificadors que delimiten elements d'una altra taula s'anomenen **claus foranes**.

Quan la cardinalitat en un dels extrems de l'associació a traduir és unitària, 1 o 0..1, llavors, donades dues taules A i B, que representen dues classes relacionades per una associació en el diagrama estàtic UML, es faria el següent:

1. Escollir la taula associada a la classe oposada a la que té cardinalitat unitària. Suposem que aquesta taula és A.
2. Afegir a la taula A una nova columna, en què s'emmagatzemen claus primàries de la taula B. Mitjançant aquesta nova columna és com es referencien a partir d'ara els elements de la taula A.
3. No cal fer res en la taula B.
4. Si els dos extrems de la relació són unitaris, es pot escollir indistintament qualsevol taula per incloure la clau primària de l'altra.

La figura 2.2 mostra un exemple que aclareix molt millor aquest procés. En aquesta figura, d'acord amb els passos indicats, la taula ENCARRECS correspondria a la taula A de la descripció i CLIENTS, en tenir la classe que representa cardinalitat unitària a l'associació, la taula B.

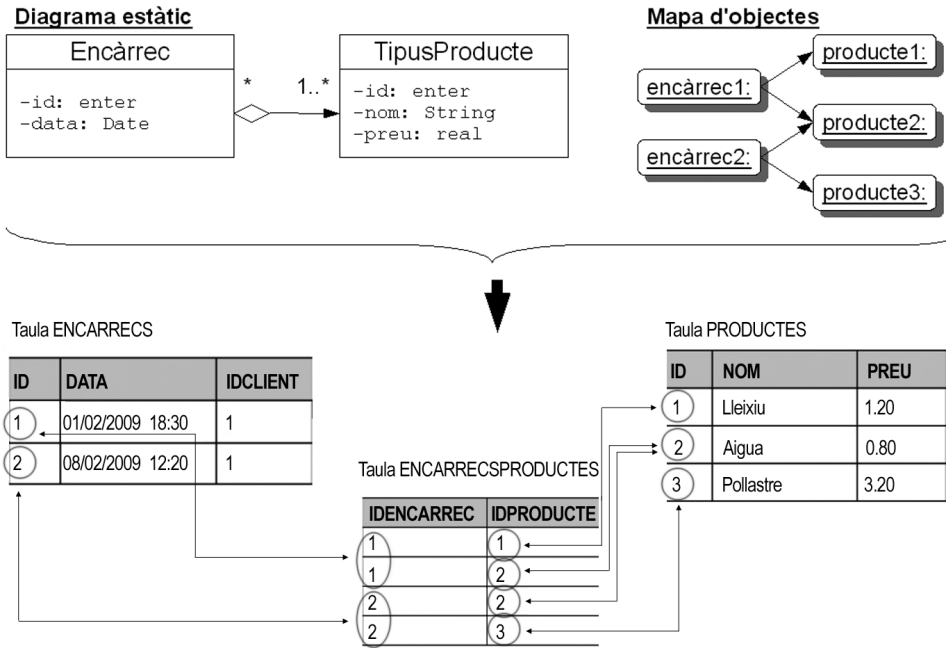
FIGURA 2.2. Tnaslació de les relacions entre objectes als camps d'una taula.



Per enumerar els encàrrecs d'un client determinat, només cal saber-ne l'identificador, i llavors llistar totes les files de la taula d'encàrrecs que tenen aquest identificador en la columna IDCLIENT. En aquest exemple, IDCLIENT és una clau forana a la taula ENCARRECS.

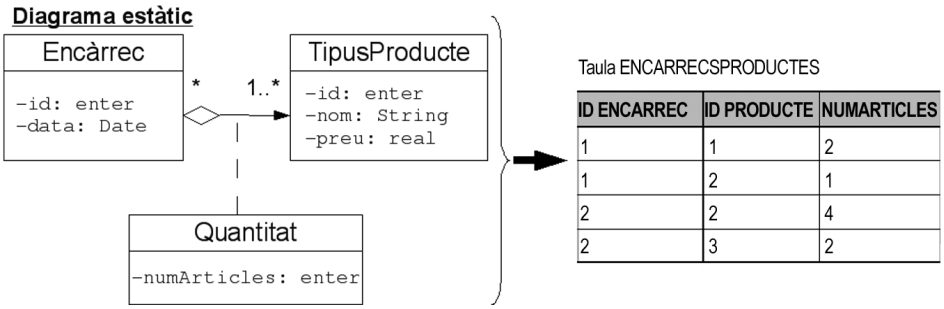
El cas d'una associació en què els dos extrems tenen cardinalitat múltiple, 1..* o *, és una mica més aparatós, ja que requereix la intervenció d'una nova taula dedicada exclusivament a enumerar totes les relacions entre elements de les dues taules associades usant-ne les claus primàries. Cada fila representa una associació entre dos elements. La figura 2.3 aclareix aquest cas:

FIGURA 2.3. Translació de cardinalitats múltiples a camps d'una taula.



Aquest sistema també permet traduir fàcilment a taula una classe associativa, ja que en el fons no és més que un conjunt d'atributs vinculat a una associació. Per tant, com es pot veure en la figura 2.4, el mecanisme és el mateix, però afegint noves columnes d'acord amb els atributs definits en la classe associativa.

FIGURA 2.4. Translació de classes associatives a camps d'una taula.



Finalment, queda exposar el cas especial de les relacions d'herència. Malauradament, l'herència és una propietat exclusiva de l'orientació a objectes, pel que no es pot traduir totalment a un model relacional. Només es pot intentar simular de la millor manera possible. Per fer-ho, hi ha dues possibilitats.

D'una banda, es pot crear una taula diferent per cada classe instanciable, de manera que els objectes s'inclouran en la taula que correspongui d'acord amb el seu tipus exacte (la subclasse més concreta). Això té l'inconvenient que en les taules relatives a classes més concretes hi haurà columnes que ja apareixen en les taules de classes més generals. A més a més, quan calgui fer operacions en les taules d'un tipus d'objecte concret que estigui en les zones més generals de la jerarquia, cal mirar cada taula.

D'altra banda, en lloc de crear diverses taules, se'n pot crear una de sola amb tots els objectes a dins. Les columnes que es definiran correspondran a l'agregació de tots els possibles atributs dins la jerarquia d'herència. Això estalvia tenir columnes repetides, però implica que hi haurà elements amb valors nuls per a certes columnes (ja que pel seu subtipus no els correspon tenir cap valor). Per tant, cal tenir-ho ben present en fer consultes i modificacions a la taula.

En qualsevol cas, sempre que es recuperin dades de les taules de la BD i es vulgui instanciar un objecte que forma part d'una jerarquia d'herència, el desenvolupador haurà d'establir alguna manera d'identificar a quina subclasse concreta pertany cada fila d'una taula.

2.2 El llenguatge SQL

Tot i les particularitats de cada fabricant, hi ha un llenguatge genèric que permet efectuar les accions més bàsiques amb una BD: el llenguatge SQL.

L'**SQL** (*structured query language*, llenguatge de consultes estructurades) és un llenguatge estàndard per a l'accés a BD relacionals, de manera que és possible operar-hi: consultar les dades emmagatzemades, modificar-les...

SQL

Si bé aquest llenguatge és un estàndard, cal tenir en compte que cada fabricant disposa de les seves extensions propietàries, totalment incompatibles entre elles.

Una sentència SQL no és més que una cadena de text, amb una sintaxi concreta, que indica un seguit d'ordres a la BD. En aquest sentit, no és gaire diferent d'un petit bocí de codi d'un programa. Aquesta sintaxi SQL és molt extensa i es poden escriure llibres sencers descrivint-ne totes les funcionalitats. En aquest apartat només es fa un breu incís sobre quines són les operacions mínimes indispensables i la seva sintaxi més bàsica. El suficient per entendre com es poden desenvolupar aplicacions en Java que utilitzin accés a una BD. Les operacions que permet SQL es poden dividir en dos tipus. D'una banda, les que componen el llenguatge de definició de dades (*data definition language*, o DDL), que serveixen per afegir informació a la BD, principalment per gestionar taules. D'altra banda, hi ha les que formen part del llenguatge de manipulació de dades (*data manipulation language*, o DML), que permeten llegir o modificar el contingut de la BD.

De moment veurem quina mena de sentències SQL accepten les BD, i després ja veurem com és possible agafar el text d'aquestes sentències i executar-les sobre la BD mitjançant codi Java.

2.2.1 Tipus de dades

Abans de poder crear cap taula, cal decidir quin és el tipus de les dades que conté a cadascuna de les seves cel·les (igual que cal triar els tipus de dades dels atributs a les classes). Els tipus de dades disponibles i quines són les seves paraules clau són específics a SQL, i per tant, no es pot reusar la nomenclatura de Java o UML. A més a més, cal tenir en compte que cada fabricant de BD defineix els seus propis tipus, per la qual cosa la llista pot variar segons el sistema utilitzat. Cal consultar la documentació de la BD per saber quins hi ha disponibles. De totes formes, tot seguit es llisten els que normalment podreu usar quan treballeu amb els mecanismes que proporciona Java per accedir a BD.

TAULA 2.1. Tipus de dades SQL.

Tipus	Descripció	Tipus Java semblant
CHAR(mida)	Cadena de text de mida fixa.	`String`
VARCHAR(mida)	Cadena de text de mida variable.	`String`
LONG VARCHAR(mida)	Cadena de text arbitràriament llarga, de mida variable.	`String`
BINARY	Bloc binari curt de mida fixa.	`byte[]`
VARBINARY	Bloc binari curt de mida variable.	`byte[]`
LONG BINARY	Bloc binari llarg de mida variable.	`byte[]`
BIT	Un bit, que pot ser només 0 o 1.	`boolean`
TINYINT	Enter de 8 bits, entre -128 i 127.	
SMALLINT	Enter de 16 bits, entre -32768 i 32767.	`short`
INTEGER	Enter de 32 bits, entre -2147483648 i 2147483647.	`int`
BIGINT	Enter de 64 bits.	`long`
REAL	Real de simple precisió.	`float`
DOUBLE	Real de doble precisió.	`double`
DATE	Una data consistent en dia, mes i any.	`java.sql.Date`
TIME	Temps consistent en hora, minuts i segons.	`java.sql.Time`
TIMESTAMP	Combinació de DATE i TIME, i, a més a més, el seu equivalent en nanosegons.	`java.sql.Date`

Vegem amb una mica més de detall algunes de les característiques més rellevants de cada tipus de dades, que caldrà tenir en compte en programar en Java aplicacions que gestionen dades dins una BD usant SQL.

Tipus relatius a cadenes de text

El tipus `VARCHAR` sol ser el més típicament usat per desar cadenes de text a la BD, i està suportat de totes les principals bases de dades. En definir-lo, cal usar un paràmetre que especifica la longitud màxima de la cadena. Per tant, `VARCHAR(25)` defineix una cadena la longitud de la qual pot ser de fins a 25 caràcters. Ara bé, cal tenir present que les BD solen tenir un límit en la mida màxima que poden suportar, que sol rondar els 254 caràcters. Tot i definir aquesta mida màxima, sempre que es desa un valor a una cel·la d'aquest tipus, quan posteriorment es consulta aquest valor, la cadena de text retornada té com a mida el seu valor original al ser desada. No s'"omple" artificialment amb espais o altres caràcters fins arribar al màxim.

En contrast, les cadenes de longitud fixa, usant el tipus CHAR, sí que sempre mantenen la seva longitud indicada en ser introduïdes, i posteriorment consultades, a la BD, i per tant, si el nombre de caràcters és inferior al valor especificat, la resta de posicions s'omplen amb espais en blanc fins arribar a la mida definida. La seva mida màxima també sol ser de 254 caràcters. Per tant, donades les definicions `VARCHAR(10)` i `CHAR(10)`, si desem el text "Hola" a la BD, en consultar posteriorment el valor, en el primer cas s'obtindrà "Hola", mentre que en el segon "Hola ". Això és un fet que cal tenir molt en compte.

Finalment, el tipus `LONGVARCHAR`, en canvi, ve justificat perquè totes les principals bases de dades han de suportar algun tipus de gran cadena de gran longitud. Per "gran longitud" s'està referint a mides de fins un gigabyte.

Tot i aquestes particularitats, en general, a un programador en Java no li cal distingir entre els tres tipus de dades a l'hora de gestionar les dades, totes elles es poden expressar dins el codi del programa com un tipus `String` o directament a un *array* de `char` (`char[]`).

A SQL els **literals de cadenes de text** s'especifiquen entre cometes simples, '...'

Tipus relatius a cadenes binàries

Els tres tipus vinculats a valors binaris segueixen una filosofia molt semblant a les cadenes de text, al menys en el que es refereix al seu aspecte de cadena de mida fixa o variable. Per tant, una dada del tipus `BINARY` es farceix amb valors addicionals fins fer-la arribar a la mida establerta. Ara bé, desafortunadament, l'ús d'aquests tipus de binaris no ha estat estandarditzada i el seu suport varia considerablement entre les principals bases de dades. Per tant, és imprescindible mirar-ne la documentació, ja que el que s'aplica per a un cas pot no ser cert per a un altre. Fins i tot, pot ser que una BD no implementi alguns d'aquests tipus.

De manera semblant a les cadenes de text, els tipus `BINARY` i `VARBINARY` solen estar limitats a 254 bytes, mentre que `LONGVARBINARY` es pot usar per a mides

arbitràriament grans, normalment fins a un gigabyte de dades. En qualsevol cas, les dades relatives a aquests tipus es poden gestionar dins d'un programa en Java usant *arrays* de `byte` (`byte[]`).

Finalment, el tipus `BIT` és especial, ja que gestiona un únic valor binari, 0 o 1. Per aquest motiu, tot i no ser estrictament el mateix, el tipus de dades recomanat per gestionar-lo dins un programa en Java és el `boolean`. Es pot fer que `true` equivalgui a 1 i `false` a 0.

Tipus relatius a valors numèrics

Tots els tipus associats a valors numèrics, ja siguin enters o reals, es comporten de manera molt similar, ja que existeix una correspondència pràcticament immediata entre el tipus de dades SQL i els tipus de dades que proporciona Java. Per tant, saber com gestionar dades d'aquests tipus en programes que accedeixen a una BD no és gaire difícil. Només cal usar el tipus equivalent i ja està, sense cap complicació extra.

L'únic cas on cal anar amb compte és per al tipus `TINYINT`, de 8 bits, ja que el tipus de valors enters més petit en Java, el `short`, és de 16 bits. De cara a consultar valors des de la base de dades no hi ha cap problema, però en emmagatzemar-ne, cal ser conscients que si el valor enter a desar és superior al rang possible (-128 i 127), el valor que acabarà a la BD no serà correcte. De fet, a nivell general, com passa en fer conversions entre tipus numèrics en Java, si en consultar un valor de la BD s'usa una variable d'un tipus amb major capacitat, mai no hi haurà problemes, per la qual cosa res no impedeix usar un `double` Java per consultar valors de tipus `REAL`.

Tipus relatius a dates

Finalment, entre els tipus bàsics de SQL es troben uns d'especials que serveixen per gestionar valors de dates. Si bé, dins una aplicació Java genèrica, l'ús de dates no ha de ser necessàriament molt usat, a les BD molt sovint sí que cal desar dates, per la qual cosa tenir un tipus especial només per a això està bastant justificat i facilita molt la feina (més que haver de desar per separat cada part o usar cadenes de text amb cert format).

El tipus `DATE` representa una data que consta de dia, mes i any, mentre que el tipus `TIME` indica només hores, minuts i segons. Com ja passava amb altres tipus de dades, aquests s'implementen per només un subconjunt de les bases de dades principals. Algunes bases de dades ofereixen alternatives de tipus SQL que serveixen per al mateix propòsit. El tipus `TIMESTAMP` és una combinació dels dos, però està suportat per un nombre molt petit de bases de dades i per tant no és gaire recomanable dependre'n molt. Aquest darrer tipus, junt amb les dades relatives a data i temps, també desa un camp auxiliar on es compta fins a una precisió de nanosegons.

Vigileu especialment si useu els mecanismes de correcció automàtica del Netbeans.

Java proporciona tres classes que serveixen específicament per tractar aquests tipus de dades SQL (i només per al cas d'SQL): `java.sql.Date`, `java.sql.Time` i `java.sql.Timestamp`, ja que les seves classes per gestionar dates a nivell genèric no encaixen exactament en els seus camps amb aquests tipus. Per tant, alerta, cal anar amb molt de compte amb la classe que s'usa per gestionar dades d'aquests tipus als vostres programes, ja que, per exemple, en Java hi ha dues classes anomenades `Date`. Una al *package* `java.util` i una altra al `java.sql`.

Cal dir que, tot i que aquestes tres classes, de fet, hereten de `java.util.Date`, i, per tant, en el pitjor cas sempre és possible fer conversions entre elles d'alguna manera (consultant les dades dins l'objecte i adaptant-les al nou format), sempre és preferible usar la classe vinculada exclusivament a SQL.

Dominis

A més dels tipus de dades predefinits, existeix la possibilitat de treballar amb dominis definits per l'usuari, on s'especifiquen un conjunt de valors concrets possibles. Per exemple, suposeu que voleu definir un camp on els únics valors possibles són un conjunt de ciutats concret i no voleu definir el tipus com un VARCHAR, on es podria escriure qualsevol cosa. En aquest cas, usar un domini seria la solució ideal. Per fer-ho cal usar la sentència **CREATE DOMAIN** sobre la base de dades.

```
1 CREATE DOMAIN nomDomini AS tipusDades
2 CONSTRAINT nomRestricció
3 CHECK (condicions)
```

Els valors "nomDomini" i "nomRestricció" els trieu vosaltres (un cop definits, es poden usar en altres comandes). El "tipus de dades" correspon al tipus genèric al qual pertanyen els valors que es volen concretar. Per exemple, per al cas de noms de ciutats, podria ser VARCHAR (25), si cap nom supera els 25 caràcters. El paràmetre de "condicions" indica què ha de complir un valor dins aquest domini per ser considerat vàlid. SQL contempla una sintaxi per establir comparacions i avaluar diferents condicions lògiques complexes (`AND`, `OR`...). De totes maneres, normalment per a aquest cas el que se sol fer és indicar directament una llista de valors, mitjançant la sentència **VALUE IN**, on s'inclou entre parèntesis els valors acceptats. Per exemple:

```
1 CREATE DOMAIN ciutats AS VARCHAR(25)
2 CONSTRAINT llistaCiutats
3 CHECK (VALUE IN ('Barcelona', 'Badalona', 'Mansou', 'Alella'))
```

Quan es defineix un domini, aquest es pot usar a nivell de sintaxi de les sentències de gestió de taules exactament igual que qualsevol altre tipus SQL. Per tant, donat el domini de ciutats, es pot usar "ciutats" com si fos un tipus de dades.

Si mai es vol eliminar la definició d'un domini a la nostra BD, es pot usar la sentència **DROP DOMAIN**.

```
1 DROP DOMAIN nomDomini (RESTRICT|CASCADE)
```

Aquesta necessita un paràmetre, que pot ser **RESTRICT** o **CASCADE**. En el primer cas, el domini només s'esborrarà realment si no s'usa enlloc de la BD (a cap taula). En el segon cas, s'esborrarà encara que estigui en ús, però allà on es doni aquest cas, es reemplaça automàticament pel tipus de dades associat al domini. Per exemple, si s'elimina el domini "ciutats", a totes les taules on s'usin les columnes passaran a ser del tipus `VARCHAR(25)` automàticament, ja que no poden quedar simplement sense cap tipus definit.

2.2.2 Gestió de taules

Per poder accedir a qualsevol dada dins els vostres programes, evidentment, abans de res aquestes dades han d'existir dins taules a la BD relacional. Normalment, les taules es creen *a priori*, i ja existeixen abans que la vostra aplicació hi interactuï. Per fer-ho, existeixen diferents programes que, mitjançant una interfície gràfica, permeten editar les propietats de les taules de les quals es vol disposar (noms de files, columnes...). Els IDE complexos com Netbeans incorporen ja una funcionalitat a aquest efecte, per tal de facilitar la feina.

De totes maneres, hi pot haver casos on pot ser necessari crear taules. El cas més immediat seria si precisament el que es vol és inicialitzar el sistema gestor de bases de dades, però fer-ho manualment és molt complicat o costós, o depèn de molts paràmetres que poden variar segons el cas. En aquest cas, res millor que un programa per automatitzar la feina. Per tant, és interessant donar una ullada a les sentències SQL per crear, esborrar i modificar les diferents taules que componen una BD.

Creació de taules

Per crear una taula s'usa la sentència **CREATE TABLE**, d'acord amb una llista de noms de columna i el tipus de dades que conté. Per a cada parell de noms de columna i tipus de dades es poden incloure un seguit de paràmetres per especificar informació addicional, com quina correspon a la clau primària, quin és el tipus de dades que es pot emmagatzemar, la seva mida màxima...

```
1 CREATE TABLE nomTaula
2 (nomColumna1 tipus [valorDefecte] [restriccions],
3 nomColumna2 tipus [valorDefecte] [restriccions],
4 ...)
```

Els camps de valor per defecte i de restriccions són opcionals.

Per al primer cas, es poden indicar valors per defecte a una cel·la d'aquella columna. Això és útil, per exemple, quan, en emmagatzemar informació dins una BD, trobeu que alguns dels valors dins una cel·la encara no se saben. Se sabrà més endavant, però ara mateix el que no voleu és deixar d'emmagatzemar la informació que ja sabeu. Que el procés d'afegir una fila a una taula no sigui tot o res. Per dur a terme això, normalment el que es fa és disposar d'algun valor especial, o valor per defecte, amb el qual s'indica que en aquella cel·la no hi ha un valor vàlid encara.

Per fer-ho, s'usa el paràmetre `DEFAULT valor`. El valor pot ser un literal que indica exactament el valor per defecte, o bé es poden usar algunes paraules clau especials d'SQL:

Taula 2.2. Paraules clau especials d'SQL.

Valor	Descripció
NULL	Valor especial nul (marca d'invàlid). Semblant a una referència a `null`.
USER	El nom de l'usuari de la BD que ha executat la sentència.
CURRENT_DATE	La data actual, només per a camps de tipus DATE.
CURRENT_TIME	L'hora actual, només per a camps de tipus TIME.

En relació amb les restriccions, n'hi ha unes quantes de disponibles, per la qual cosa ens centrarem en les més importants. Per una banda, hi ha les restriccions de columna, que indiquen condicions que ha de complir qualsevol accés a la BD per ser considerat vàlid. Si no es compleix la condició, la BD retorna un error. D'aquesta manera podeu garantir automàticament que les dades que es desen mantenen una coherència sense haver de fer-ho amb codi dins el vostre programa. Entre les més típiques es troben:

Taula 2.3. Restriccions típiques d'SQL.

Restricció	Descripció
PRIMARY KEY	La columna desa la clau primària de cada fila (els valors no poden ser repetits ni nuls).
NOT NULL	No s'admeten valors nuls.

Per exemple, si es vol fer una taula de clients on el seu identificador serà la clau primària de cada client, es faria:

```
CREATE TABLE CLIENTS
(ID INTEGER PRIMARY KEY,
NOM VARCHAR(30),
APOSTAL VARCHAR(50),
AELECTRONICA VARCHAR(25),
TELEFON VARCHAR(15))
```

Amb la qual cosa es crea la taula (buida, de moment):

Taula 2.4. Taula CLIENTS inicial

ID	NOM	APOSTAL	AELECTRONICA	TELEFON

Modificació de taules

Un cop una taula ja ha estat creada, és possible fer-hi modificacions, sense haver d'esborrar-la i crear-ne una de nova, usant la sentència **ALTER TABLE**.

```
ALTER TABLE nomTaula
acció
```

Les accions possibles més rellevants **ADD**, **ALTER** i **DROP**, per afegir, modificar o eliminar una columna dins la taula. Cada acció usa un seguit de paràmetres semblants a quan es crea una taula des de zero, només que en aquest cas es tracta d'una definició *a posteriori*. La sinxati d'aquestes accions és:

```
1 ADD nomColumna tipus [valorDefecte] [restriccions]
2 MODIFY nomColumna tipus [valorDefecte] [restriccions]
3 DROP nomColumna (RESTRICT|CASCADE)
```

En el cas de DROP, no cal definir res, ja que el que s'està fent és eliminar. Ara bé, com que les dades de la columna poden estar referenciades dins altres taules de la BD, cal preveure aquesta situació. Indicant el paràmetre RESTRICT, l'operació no es durà a terme si es dóna aquest cas. Amb el paràmetre CASCADE, tot el que referenciï aquesta columna també s'eliminarà.

Per exemple, si es vol afegir una columna amb el nombre de vegades que un client ha comprat a l'establiment, es pot fer:

```
1 ALTER TABLE CLIENTS
2 ADD NCOMANDES INTEGER NOT NULL
```

Amb això, la taula CLIENTS passa a ser la següent:

Taula 2.5. Taula CLIENTS actualitzada

ID	NOM	APOSTAL	AELECTRONICA	TELEFON	NCOMANDES

A part, també és possible canviar el nom d'una taula amb la sentència:

```
1 ALTER TABLE nomTaula
2 RENAME TO nouNom
```

Esborrat de taules

En qualsevol moment també és possible eliminar totalment una taula de la BD a partir del seu nom. La sintaxi és la següent:

```
1 DROP TABLE taula (RESTRICT|CASCADE)
```

Els paràmetres RESTRICT i CASCADE tenen el mateix significat que en modificar una columna (de fet, esborrar una taula és com esborrar totes les seves columnes). Un cop eliminada, tota la informació que conté es perd, per tant, cal anar amb molt de compte en executar aquesta sentència.

2.2.3 Consulta de dades

La tasca més freqüent, la que segurament s'executarà més vegades en un programa que utilitza una BD, és de ben segur la cerca d'informació entre les dades que té

emmagatzemades. Per fer-ho, es disposa de la sentència **SELECT**, que permet recuperar informació de la BD d'acord amb algun criteri. Atès que una cerca ha de ser capaç de poder englobar molts criteris, de manera que es trobi exactament la dada que es necessita, aquesta sentència pot incorporar opcions ben complexes. Per això, partirem de la seva sintaxi bàsica i després veurem com es pot anar refinant la cerca, ampliant-la.

Per anar seguint les diferents instruccions, partirem d'una taula anomenada CLIENTS, on es desen un seguit de clients amb certa informació. Sobre ella es veuran diferents exemples per a cada tipus de sentència:

Taula 2.6. Taula CLIENTS

ID	NOM	APOSTAL	AELECTRONICA	TELÈFON	NCOMANDES
1	Client1	Adreça1	e-mail1@domini.com	+34931112233	4
2	Client2	Adreça2	e-mail2@domini.com	+34932223344	1
3	Client3	Adreça3	e-mail3@domini.com	+34933334455	10
4	Client4	Adreça3	e-mail4@domini.com	+34933335566	7

Per començar, la seva sintaxi base, que permet una cerca molt general, és:

```
SELECT nomNolumna1 [AS nouNom], nomColumna2 [AS nouNom]...
FROM nomTaula
```

En executar-la, la BD retorna les columnes demanades de la taula indicada. La part `As nouNom`, que es pot posar darrere el nom de cada columna, és opcional i permet reanomenar la columna tal com es mostra en el resultat, enlloc d'usar el nom original que hi ha a la taula.

El format de la resposta, com s'engloben les dades consultades, ja depèn del mecanisme usat per executar la sentència. Per exemple, si s'ha usat una aplicació gràfica, les columnes es poden mostrar en una nova taula reduïda, només amb les columnes desitjades. Si és un programa per línia de comandes, potser s'imprimeixen per pantalla com un text. O si es tracta d'un accés mitjançant codi en Java, com no podria ser d'altra manera, estaran englobades dins un objecte. De moment, però, no cal donar més voltes a aquest fet. N'hi ha prou a saber que, d'una manera o una altra, existirà un mecanisme per poder accedir al resultat sempre que s'executi aquesta sentència. Per ara, es visualitzarà la resposta d'una sentència **SELECT** com una nova taula.

Per tant, donades les següents sentències, si es volen enumerar només els noms i les adreces dels clients de la taula original, es faria així:

Sentència:

```
SELECT NOM, APOSTAL AS ADREÇA
FROM CLIENTS
```

Resultat:

TAULA 2.7. Resultat de la comanda SELECT

NOM	ADREÇA
Client1	Adreça1
Client2	Adreça2
Client3	Adreça3
Client4	Adreça3

Si, en lloc d'una llista de noms de columnes, s'usa un **asterisc**, *, es retornen tots els valors de la taula.

Sentència:

```
1 SELECT *
2 FROM CLIENTS
```

Resultat:

TAULA 2.8. Resultat de la comanda SELECT

ID	NOM	APOSTAL	AELECTRONICA	TELÈFON	NCOMANDES
1	Client1	Adreça1	e-mail1@domini.com	+34931112233	4
2	Client2	Adreça2	e-mail2@domini.com	+34932223344	1
3	Client3	Adreça3	e-mail3@domini.com	+34933334455	10
4	Client4	Adreça3	e-mail4@domini.com	+34933335566	7

També val la pena comentar que, en cas de taules on pot haver-hi valors repetits a les columnes, si es vol que en el resultat d'una consulta descarti repeticions, es pot usar `SELECT DISTINCT`, en lloc de només `SELECT`. L'exemple següent té en compte el fet que els clients 3 i 4 viuen al mateix lloc.

```
1 SELECT DISCTINCT APOSTAL
2 FROM CLIENTS
```

Resultat:

TAULA 2.9. Resultat de la comanda SELECT

APOSTAL
Adreça1
Adreça2
Adreça3

Aplicació de funcions

Ara bé, sovint, el que es vol no és pas simplement llistar tota la informació d'una columna, sinó només aquelles dades que compleixen certes condicions. Atès que les sentències de consulta són les més executades, el que no té sentit, o seria molt ineficient, és que només es pugui consultar un seguit de columnes senceres i després sigui tasca vostra haver de processar-les usant codi per refinar la cerca. Per exemple, cercar quin és el darrer client que s'ha donat d'alta, o si existeix algun client amb una adreça de correu electrònic concreta. Així doncs, en realitat, mitjançant la sentència `SELECT` és possible delegar aquesta tasca a la BD, que, en definitiva, és qui emmagatzema i gestiona tota la informació, de manera que tot plegat resulti més eficient.

Les funcions acceptades són semblants a les que existeixen als programes típics de full de càlcul.

Per una banda, SQL conté un seguit de funcions predefinides que es poden aplicar sobre les columnes, de manera que es demana a la BD que treballi, no sobre els valors de la columna, sinó sobre el resultat d'aplicar aquesta funció. Entre les més destacades:

Taula 2.10. Funcions SQL

Funció	Descripció
MAX(nomColumna)	Retorna la fila amb el valor màxim en aquesta columna.
MIN(nomColumna)	Retorna la fila amb el valor mínim en aquesta columna.
SUM(nomColumna)	Retorna la suma de totes les files.
AVG(nomColumna)	Retorna el valor mitjà de totes files.
COUNT(nomColumna)	Retorna el nombre de files.
FIRST(nomColumna)	Retorna només la primera fila.
LAST(nomColumna)	Retorna només la darrera fila.
UCASE(nomColumna)	Transforma els valors de les files a tot majúscules.
LCASE(nomColumna)	Transforma els valors de les files a tot minúscules.
LEN(nomColumna)	Retorna la longitud dels valors a les files.
ROUND(nomColumna)	Arrodoneix els valors.

Evidentment, les funcions només es poden aplicar sobre aquelles columnes de tipus de dades on tingui sentit. Per exemple, només es pot fer la suma dels valors d'una columna on es desin valors numèrics.

Cerques mitjançant condicions

Ara bé, sovint, en usar aquestes funcions, la semàntica de les dades consultades canvia. Per exemple, si es vol comptar el nombre d'elements que té la taula, i s'usa la funció `COUNT`, ara el valor retornat ja no es correspon estrictament al valor de cap columna en concret. No és ni un ID, ni un Nom... Per això, la sentència `SELECT` permet demanar a la BD que, quan retorni el resultat, reanomeni la columna transformada amb un altre nom. Això pot ser útil per fer més llegible

el resultat, o, en definitiva, el codi d'un programa que la usi. Aquesta és la utilitat de l'apartat opcional `AS nouNom`.

Per exemple, si es vol veure quants clients hi ha a la taula CLIENTS, es podria fer la consulta següent:

Sentència:

```
1 SELECT COUNT(NOM) AS NCLIENTS
2 FROM CLIENTS
```

Resultat:

TAULA 2.11. Resultat d'executar SELECT

NCLIENTS
4

Si bé la sentència `SELECT`, tal com s'ha vist fins ara, és suficient per recuperar dades de manera molt bàsica, és possible fer cerques molt més potents mitjançant l'addició del paràmetre opcional `WHERE condició` al final. Aquesta condició pot combinar diferents operadors, o fins i tot crides a les funcions SQL, de manera que arribi a ser tan complexa com es vulgui, sempre i quan s'avaluï a cert o fals, de manera molt semblant a les condicions d'una sentència condicional o iterativa d'un programa en Java (`if`, `while`...). En cas de combinar diferents condicions simples per fer-ne una de més complexa, aquestes es poden anar agrupant usant parèntesi, com al Java.

Els operadors que accepta SQL són els següents. Aneu amb compte, ja que alguns són diferents que en Java. Per exemple, la igualtat no és un doble igual (==), sinó només un (=).

TAULA 2.12. Operadors SQL

Operador	Descripció
=	Igual
<	Menor
<=	Menor o igual
>	Major
>=	Major o igual
<>	Diferent
NOT	Negació lògica
AND	Conjunció lògica
OR	Disjunció lògica

Recordeu que els literals de cadenes de text en SQL s'escriuen entre cometes simples, i no dobles cometes com al Java.

Per exemple, si es vol enumerar només els clients que viuen a l'adreça "Adreça3", es podria fer la consulta que hi ha a continuació. En aquesta, és la pròpia BD qui s'encarrega de fer el processament i la discriminació dels elements, no cal que ho fem nosaltres després al codi del nostre programa, la qual cosa no només és molt més eficient des del punt de vista d'execució del programa, sinó que és més ràpid de codificar i fa el codi més simple.

Sentència:

```
SELECT *
FROM CLIENTS
WHERE APOSTAL='Adreça3'
```

Resultat:

TAULA 2.13. Resultat d'executar SELECT

ID	NOM	APOSTAL	AELECTRONICA	TELÈFON	NCOMANDES
3	Client3	Adreça3	e-mail3@domini.com	+34933334455	10
4	Client4	Adreça3	e-mail4@domini.com	+34933335566	7

Ara bé, com s'ha dit, res no impedeix fer condicions complexes on s'usen també les funcions exposades amb anterioritat. Per exemple, si es vol cercar tots els noms dels clients que han fet un conjunt de compres inferior a la mitjana (i, ja de pas, els etiquetem com a "pitjors clients"), es podria fer la consulta que ve a continuació. El valor mitjà de comandes és 5.5, per la qual cosa, el resultat estableix els clients que tenen menys comandes que aquest valor.

Sentència:

```
SELECT NOM AS PITJORCLIENT
FROM CLIENTS
WHERE NCOMANDES < AVG(NCOMANDES)
```

Resultat:

TAULA 2.14. Resultat d'executar SELECT

PITJORCLIENT
Client1
Client2

Altres cerques mitjançant WHERE

Finalment, després d'un apartat `WHERE` també es poden afegir un seguit de paraules clau que permeten anar més enllà dels típics operadors lògics o matemàtics, de manera que encara es poden fer cerques més concretes, o d'acord a criteris encara més específics, però freqüents en el context d'una cerca. Aquestes opcions es poden combinar amb altres condicions per fer cerques encara més complexes. Tot seguit s'expliquen algunes de les més destacades.

L'opció **IN**, o **NOT IN**, permet indicar una llista de valors, de manera que la condició es considera certa si el valor d'una fila a la columna corresponent és (o no) entre algun dels valors de la llista. Per exemple, si es volguessin consultar els noms i correus electrònics dels clients que viuen a l'adreça postal 2 o 3 (fixeu-vos que hi ha dos clients que viuen a l'Adreça 3), es faria així:

Sentència:

```
SELECT NOM, AELECTRONICA
FROM CLIENTS
WHERE APOSTAL IN ('Adreça2', 'Adreça3')
```

Resultat:

TAULA 2.15. Resultat d'executar SELECT

NOM	AELECTRONICA
Client2	e-mail2@domini.com
Client3	e-mail3@domini.com
Client4	e-mail4@domini.com

L'opció **NULL**, o **NOT NULL**, permet filtrar una consulta d'acord a les files que en alguna columna tenen un valor NULL, o no. Per exemple, si es vol obtenir una llisa de telèfons vàlids, partint de la suposició que es permet donar d'alta clients encara que no se sàpiga el seu telèfon, es pot fer d'aquesta manera:

Sentència:

```
SELECT NOM, TELÈFON AS CLIENTSATRUCAR
FROM CLIENTS
WHERE TELÈFON NOT NULL
```

Resultat:

TAULA 2.16. Resultat d'executar SELECT

NOM	CLIENTSATRUCAR
Client1	+34931112233
Client2	+34932223344
Client3	+34933334455
Client4	+34933335566

L'opció **BETWEEN valorInicial AND valorFinal** permet simplificar la creació de condicions on es vol veure si un valor es troba dins un rang. Per exemple, si es vol trobar el nom dels clients que han fet entre 3 i 8 compres, es faria amb aquesta sentència:

Sentència:

```
SELECT NOM
FROM CLIENTS
WHERE NCOMANDES BETWEEN 3 AND 8
```

Resultat:

TAULA 2.17. Resultat d'executar SELECT

NOM
Client1
Client4

Finalment, l'opció **LIKE patró** permet establir si els valors compleixen un patró concret, en lloc d'haver de comparar una igualtat estricta. Això és molt útil, ja que sovint, en fer cerques sense saber exactament el contingut de les dades disponibles, el que es vol és trobar elements que compleixin de manera aproximada certes condicions, ja que *a priori* és impossible saber exactament què es pot cercar realment. El patró es representa com un literal de cadena de text, amb l'opció de posar un subratllat, '_', per indicar qualsevol lletra, i un percentatge, '%', per indicar una seqüència de 0 o més lletres qualssevol.

Per exemple, suposem que es vol cercar el nom dels clients amb un número de telèfon on, en alguna part, hi ha dos nombres 4 consecutius. La consulta seria la següent:

Sentència:

```
SELECT NOM, NTELÈFON
FROM CLIENTS
WHERE NTELÈFON LIKE '%44%'
```

Resultat:

TAULA 2.18. Resultat d'executar SELECT

NOM	TELÈFON
Client2	+34932223344
Client3	+34933334455

Ordenació dels resultats

Igual que la BD ja és capaç de dur a terme cerques d'acord a certs criteris usant les comandes adients d'SQL, de manera que no cal consultar totes les dades i després processar-les dins els vostres programes, també és possible establir criteris d'ordenació, de manera que el resultat ja està ordenat automàticament i no cal que ho feu vosaltres *a posteriori*.

Per assolir-ho, es pot usar el paràmetre **ORDER BY nomColumna1, nomColumna2,...** al final de la sentència `SELECT`. Aquest paràmetre ordena les files del resultat d'acord als valors de diferents columnes, per ordre d'importància. En el cas de valors de tipus cadena de text, s'ordenen els elements alfabèticament.

Per defecte, l'ordenació és ascendent, del valor més petit al més gran, però si es desitja es pot fer el cas invers usant la paraula clau `DESC` immediatament després del nom d'una columna. Per exemple, per mostrar els clients ordenats per nombre de compres, de manera que primer es mostra el que n'ha fet més, s'usaria la sentència:

Sentència:

```
SELECT *
FROM CLIENTS
ORDER BY NCOMANDES DESC
```

Resultat:

TAULA 2.19. Resultat d'executar SELECT

ID	NOM	APOSTAL	AELECTRONICA	TELÈFON	NCOMANDES
3	Client3	Adreça3	email3@domini.com	+34933334455	10
4	Client4	Adreça3	email4@domini.com	+34933335566	7
1	Client1	Adreça1	email1@domini.com	+34931112233	4
2	Client2	Adreça2	email2@domini.com	+34932223344	1

Combinació de taules

De vegades, és necessari combinar la informació de diverses taules diferents per obtenir un únic resultat, sovint quan les condicions del paràmetre `WHERE` depenen de valors desats a altres taules. SQL permet fer operacions a partir de les dades de més d'una taula simplement especificant a l'apartat `FROM` que la informació a tractar prové de més d'una taula enlloc d'una, enumerades com una llista separada per comes: `FROM nomTaula1, nomTaula2...`. En principi, es poden combinar tantes taules com es vulgui, però ens centrarem en el cas de només dues taules.

Quan s'usa aquest mecanisme, per a cada nom de columna usada a qualsevol part de la sentència cal especificar a quina taula exactament s'està referint, no n'hi ha prou de posar només el nom de la columna. Això evita conflictes en duplicitats de noms entre taules diferents. Per fer-ho, només cal usar la sintaxi `nomTaula.nomColumna`, de manera molt semblant a l'accés a elements públics dins objectes en Java. Aquesta nomenclatura cal usar-la tant quan s'enumeren les columnes després del `SELECT` com dins la condició al `WHERE`, si n'hi ha.

Si els noms de les taules són llargues, pot passar que el text de la sentència sigui una mica farragós. Per això, en declarar les taules on fer la consulta, és possible assignar àlies abreujats, que poden ser usats a la resta de la sentència. L'àlies s'escriu directament després del nom de la taula.

```
1 SELECT àlies.nomColumna1, àlies,nomColumna2, ...
2 FROM nomTaula1 àlies1, nomTaula2 àlies2
3 WHERE condició
```

Per veure-ho més clar, utilitzem un exemple. Suposem que existeix una nova taula PREMIS on s'enumeren premis de vals de descompte segons el nombre de compres acumulades d'un client. Pot ser interessant combinar la taula de clients amb aquesta per saber si a algun client li correspon alguna oferta.

Taula 2.20. Resultat d'executar SELECT

NCOMANDES	VAL
5	Un val de descompte del 5% a la propera compra.
10	Un val per a una tovallola de platja de regal.
15	Un val per a un 2 per un en qualsevol producte.
20	Dos vals de descompte de 10 euros en comandes diferents.

Si es vol processar un enviament de vals de premi a les diferents adreces de clients que s'ho han guanyat, caldrà combinar les dues taules: saber quins premis mereix cada client i saber a quin nom i a quina adreça cal enviar el premi.

Sentència:

```
SELECT c.NOM, c.APOSTAL, p.VAL
FROM CLIENTS c, PREMIS p
WHERE (c.NCOMANDES >= p.NCOMANDES)
```

Resultat:

Taula 2.21. Resultat d'executar SELECT

NOM	APOSTAL	VAL
Client3	Adreça3	Un val de descompte del 5% a la propera compra.
Client3	Adreça3	Un val per a una tovallola de platja de regal.
Client4	Adreça3	Un val de descompte del 5% a la propera compra.

Un fet molt important és veure com en el resultat s'han combinat dades de totes dues taules. Ara hi ha columnes tant de la taula CLIENTS com de la taula PREMIS, barrejades. Atès que el client 3 es mereix dos premis, ara apareix dues vegades repetit, una per cada cop que el seu nombre de compres compleix la condició establerta (major o igual que) amb qualsevol de les files a la columna NCOMANDES de la taula de PREMIS. Com que succeeix dues vegades, hi ha dues aparicions, i s'associa cada columna VAL a cada aparició. Bàsicament, la comparació es fa a nivell de totes les files d'una taula contra totes les files de l'altra, una a una.

En versions posteriors d'SQL, la sintaxi per dur a terme aquesta tasca s'ha modificat, de manera que per fer la mateixa tasca es pot usar un format de sentència diferent, mitjançant el paràmetre **JOIN ... ON ...**. Amb els dos es poden obtenir els mateixos resultats, però internament la consulta feta amb JOIN ... ON ... és molt més ràpida.

```
SELECT àlies.nomColumna1, àlies,nomColumna2, ...
FROM nomTaula1 àlies1
JOIN nomTaula2 àlies2
ON condicióOn
WHERE condicióWhere
```

En aquest cas, la segona taula, que es pot considerar com a auxiliar, ja que és la que conté informació extra sobre la qual fer comparacions o extreure dades, s'identifica explícitament. La condició a la part `ON` indica també explícitament quina és la condició sota la qual es vinculen les dues taules. En aquesta sintaxi, pot ser que no calgui afegir el `WHERE` si amb la condició `ON` és suficient per vincular les dades entre taules.

Per exemple, la sentència per llistar els premis que cal enviar es podria adaptar a aquest altre format de la manera que es veu a continuació, amb idèntics resultats. Cal comptar que en aquesta consulta, la condició que vincula les dues taules és la relació entre les columnes NCOMANDES d'amdues, per la qual cosa, aquesta és la que correspon a l'`ON`:

Sentència:

```
SELECT c.NOM, c.APOSTAL, p.VAL
FROM CLIENTS c
JOIN PREMIS p
ON (c.NCOMANDES >= p.NCOMANDES)
```

Un fet molt curiós de la unió de taules és que, de fet, res no impedeix operar amb la mateixa taula alhora, usant el mateix nom de taula al `FROM` i al `JOIN`. Tot i que pot semblar un cas molt estrany, suposeu la següent consulta: llistar tots els clients que han fet menys comandes que el client 4. Amb la unió de taules aquesta consulta és possible, ja que el que cal fer és creuar totes les dades de la taula CLIENTS amb les del client 4 (que també és a la mateixa taula). Per tant, es pot fer:

Sentència:

```
SELECT c.NOM
FROM CLIENTS c1
JOIN CLIENTS c2
ON (c1.NCOMANDES < c2.NCOMANDES)
WHERE c2.NOM='Client4'
```

Resultat:

TAULA 2.22. Resultat d'executar SELECT

NOM
Client1
Client2

Per veure més clar què ha succeït, podeu dividir la sentència en dues parts. Per una banda, el `WHERE` indica que es treballa amb una versió de la taula CLIENTS, "c2", on només hi ha els clients amb nom "Client4" (una única fila). Per altra banda, hi ha la taula "c1", que és la versió completa (4 files). Llavors, mitjançant l'`ON`, es fa la combinació de "c1" i "c2", comparant totes les files d'una contra les de l'altra i obtenint només els casos on el valor de les comandes de "c1" és menor que el de "c2". En total, 4 comparacions, i dues són certes, per als clients 1 i 2.

2.2.4 Manipulació de dades

L'objectiu final de disposar de taules correctament creades és, en definitiva, poder desar-hi informació per consultar-la o modificar-la posteriorment. Sense dades no té sentit poder fer consultes. Si bé res no impedeix que una BD tingui un conjunt de dades estàtic que mai no varia en el temps, i per tant ja n'hi ha prou d'afegir-les tot just després de crear la taula, el més normal, en una aplicació, és que les dades d'una taula puguin anar variant al llarg del temps, a mesura que es va afegint o eliminant informació.

Per dur a terme aquesta tasca, SQL disposa del conjunt de sentències corresponent que es poden executar sobre la BD. Tot seguit, veurem les més importants amb més detall.

Inserció de dades

Mitjançant la sentència **INSERT INTO** es pot afegir una nova línia a una taula concreta. La sintaxi base és:

```
1 INSERT INTO nomTaula
2 VALUES (valor1, valor2,...)
```

Els termes "valor1", "valor2" es corresponen als valors de cadascuna de les columnes de la taula en qüestió, enumerats exactament en el mateix ordre en què s'han enumerat les pròpies columnes en la creació de la taula i usant el mateix tipus de dades, de manera semblant, per exemple, a com s'especifica una llista de paràmetres en la crida d'un mètode (en el mateix ordre que en la seva definició). Evidentment, el nombre de valors també ha de concordar exactament amb el nombre de columnes. Si no es compleix alguna d'aquestes condicions, hi haurà un error.

Els valors individuals s'expressen com literals, tot i que també es pot usar la paraula clau `NULL` per indicar que es vol emmagatzemar un valor nul, o `DEFAULT` si volem que la BD emmagatzemi un valor per defecte. Ara bé, per poder fer això, cal que no s'entri en conflicte amb els paràmetres usats en definir la columna durant la creació de la taula. Per tant, per al primer cas, no es poden usar valors nuls si s'ha usat el paràmetre `NOT NULL` o `PRIMARY KEY`, i per al segon cas, cal que hi hagi realment un valor per defecte definit per a la columna. En cas contrari, no es procedirà a la inserció de les dades.

Per exemple, si sobre la taula CLIENTS s'executés la següent comanda d'inserció, llavors passaria a tenir la informació següent:

Sentència:

```
1 INSERT INTO CLIENTS VALUES (5,'Client5', 'Adreça5','e_mail5@domini.com','
      +34933336677', 3)
```

Taula resultant:

TAULA 2.23. Resultat d'executar INSERT

ID	NOM	APOSTAL	AELECTRONICA	TELÈFON	NCOMANDES
1	Client1	Adreça1	e-mail1@domini.com	+34931112233	4
2	Client2	Adreça2	e-mail2@domini.com	+34932223344	1
3	Client3	Adreça3	e-mail3@domini.com	+34933334455	10
4	Client4	Adreça3	e-mail4@domini.com	+34933335566	7
5	Client5	Adreça5	e-mail5@domini.com	+34933336677	3

Eliminació de dades

Si mai es vol esborrar una fila, o un conjunt de files, cal usar la sentència **DELETE**. Aquesta usa el paràmetre `WHERE`, de manera idèntica a quan es fan consultes, però ara com a criteri d'esborrat. Per tant, aquelles files que en una sentència `SELECT` serien el resultat, en una sentència `DELETE` són les esborrades. Un cop més, la condició pot ser tan complexa com es desitgi i, fins i tot, es poden usar funcions SQL (esborrar la fila amb el valor més baix, la primera fila).

```
1 DELETE FROM nomTaula
2 WHERE condició
```

Per exemple, per eliminar de la taula original (amb 4 clients) tots els que viuen a l'adreça 3, es faria així:

Sentència:

```
1 DELETE FROM CLIENTS
2 WHERE APOSTAL='Adreça3'
```

Taula resultant:

TAULA 2.24. Resultat d'executar DELETE

ID	NOM	APOSTAL	AELECTRONICA	TELÈFON	NCOMANDES
1	Client1	Adreça1	email1@domini.com	+34931112233	4
2	Client2	Adreça2	email2@domini.com	+34932223344	1

Modificació de dades

Un cop es disposa ja de dades dins la taula, també es poden modificar directament, sense haver d'esborrar-les i afegir-les de nou. La sentència **UPDATE** permet canviar el valor d'una cel·la. La sintaxi és:

```
UPDATE taula
SET nomColumna1=valor1, nomColumna2=valor2,...
WHERE condició
```

Per seleccionar només una fila, pot resultar molt útil usar la **clau primària** de la taula dins la condició.

Els paràmetres després de SET indiquen quins són els nous valors per a un conjunt de columnes, que poden ser totes o només una part, de manera que se seleccionen exactament les cel·les a canviar. La condició dins el WHERE serveix, novament, per delimitar quines files cal considerar de cara a l'actualització. Si més d'una fila compleix la condició, s'actualitzen totes amb els valors indicats. Per tant, cal anar amb una mica de compte en aquest cas i, si volem modificar només una cel·la, assegurar-se que la condició delimita exactament una única fila.

Per exemple, si volem canviar l'adreça del client 4 per una de nova, es podria fer així:

Sentència:

```
UPDATE CLIENTS
SET APOSTAL='novaAdreça'
WHERE ID='4'
```

Taula Resultant:

Taula 2.25. Resultat d'executar UPDATE

ID	NOM	APOSTAL	AELECTRONICA	TELÈFON	NCOMANDES
1	Client1	Adreça1	email1@domini.com	+34931112233	4
2	Client2	Adreça2	email2@domini.com	+34932223344	1
3	Client3	Adreça3	email3@domini.com	+34933334455	10
4	Client4	novaAdreça	email4@domini.com	+34933335566	7

Com en el cas de la inserció, es poden usar els valors especials DEFAULT o NULL. També és possible usar expressions com a nou valor d'actualització, en el cas de dades numèriques: sumes, restes... Per exemple, si des de l'adreça 3 s'han fet 5 comandes noves, es podria fer amb aquesta sentència:

Sentència:

```
UPDATE CLIENTS
SET NCOMANDES=NCOMANDES + 5
WHERE APOSTAL='Adreça3'
```

Taula Resultant:

TAULA 2.26. Resultat d'executar UPDATE

ID	NOM	APOSTAL	AELECTRONICA	TELÈFON	NCOMANDES
1	Client1	Adreça1	email1@domini.com	+34931112233	4
2	Client2	Adreça2	email2@domini.com	+34932223344	1
3	Client3	Adreça3	email3@domini.com	+34933334455	15
4	Client4	Adreça3	email4@domini.com	+34933335566	12

2.3 JDBC

Java proporciona l'API Java Database Connectivity (connectivitat Java a bases de dades) com a mecanisme per poder generar i invocar sentències SQL sobre un BD relacional mitjançant codi en programes Java. La seva particularitat és que, en contrast amb altres sistemes existents, ofereix una interfície comuna per a l'accés a qualsevol tipus de BD, independentment del fabricant. Per al desenvolupador, la BD real que hi ha al darrere és totalment transparent i obvia la necessitat d'efectuar cap mena de configuració en la màquina on s'executa l'aplicació que accedeix a les dades. Aquesta biblioteca es troba principalment en els paquets `java.sql` i `javax.sql`.

El controlador de BD Microsoft és `sun.jdbc.obdc.JdbcOdbcDriver`.

Partint de la suposició que ja hi ha una BD correctament configurada i a la qual volem accedir des del codi d'un programa Java, el resum de passos que cal fer dins l'aplicació és:

1. Importar correctament els *packages* corresponents.
2. Carregar el controlador (*driver*) per a l'accés a la BD. Aquest depèn de la BD a accedir.
3. Establir la connexió a la BD.
4. A partir d'aquí, ja es poden executar sentències SQL en la BD i processar les respostes.
5. Quan ja no es vol treballar més amb la BD, cal tancar la connexió.

JDBC, igual que moltes altres API en Java, està dissenyat amb la simplicitat en el pensament i intenta que l'ordre de les operacions que ha de fer l'operador sigui genèric i, fins a cert punt, lògic. Igual que per llegir dades d'un fitxer el que cal fer és dir quina és la seva ubicació, obrir-lo, llegir o escriure les dades i tancar-lo, en aquest cas la idea és similar. Simplement, "llegir-lo o escriure'l" vol dir invocar una sentència SQL, enlloc de posicionar un apuntador. Tot i així, cal tenir un cert domini d'SQL per poder fer correctament aquesta feina, és clar.

Tot seguit, es veurà amb més detall els passos més importants (es dóna per feta la importació correcta de *packages*), però només a títol d'introducció, per fer-se una idea del significat de cada pas. Es mostra un fragment de codi que consulta tots els clients d'una BD i en mostra el nom i l'adreça postal per pantalla. En aquest cas, totes les classes implicades en l'accés a una BD pertanyen al paquet `java.sql`.

```
//Importar classes
import java.sql.*;
...

//Carregar el controlador per la BD Apache Derby
Class.forName("org.apache.derby.jdbc.ClientDriver");

//Establir la connexió
String urlBaseDades = "jdbc:derby://localhost:1527/GestioEncarrecs";
String usuari = "administrador";
String contrasenya = "pswdificil";
Connection c = DriverManager.getConnection(urlBaseDades , usuari, contrasenya);

//Enviar una sentència SQL per recuperar els clients
Statement cerca = c.createStatement();
ResultSet r = cerca.executeQuery("SELECT * FROM CLIENTS");
while (r.next()) {
  System.out.println("Nom: " + r.getString("NOM") +  ", Adreça: " + r.getString
      ("APOSTAL"));
}

//Tancar la connexió
c.close();
```

2.3.1 Càrrega del controlador

Per permetre la independència de la plataforma, JDBC proporciona un gestor de controladors que gestiona dinàmicament tots els aspectes específics de l'accés a un tipus de BD concret. JDBC és qui s'encarrega de transformar totes les crides genèriques als accessos corresponents d'acord als mecanismes específics de la BD amb què s'interactua. Com a desenvolupadors, us podeu desentendre, fins a cert punt, de tots aquests detalls. Per tant, si es disposa de quatre tipus diferents de bases de dades, de diferents fabricants, per connectar-se caldrà disposar de quatre controladors diferents.

Els controladors sempre prenen la forma d'una classe Java, identificada de manera absoluta amb el nom complet del paquet que la conté i el nom de la classe en si. El seu registre de controladors es fa automàticament quan es carrega la classe a memòria, cosa que es fa amb la crida `Class.forName(nomControlador)`. Des del punt de vista del programador, no cal fer res més. Per exemple, per a l'accés a una BD Apache Derby, cal carregar el seu controlador específic, amb la crida següent:

```
Class.forName("org.apache.derby.jdbc.ClientDriver");
```

Atès que aquest controlador només serveix per a aquest tipus de BD, si s'intenta usar per connectar-se a una BD de qualsevol altre tipus (Oracle, PostgreSQL...), el programa no funcionarà.

El CLASSPATH indica on es troben les classes accessibles quan s'executa un programa Java.

Cal tenir en compte que el fet que des del punt de vista del desenvolupador la càrrega del controlador sigui senzilla no vol dir que el procés de disposar i configurar correctament un controlador sigui immediata. La classe que conté el controlador del SGBD serà diferent per a cada cas, per la qual cosa cal consultar la documentació del fabricant del programari per esbrinar quin és el nom de la classe en qüestió i saber com instal·lar-la al sistema. Tot i que per carregar el controlador no cal usar cap instrucció `import`, sí que cal que aquesta classe estigui inclosa en el `CLASSPATH` perquè es pugui localitzar correctament.

En usar **JDBC**, la instrucció "Class.forName" és l'única que canvia, d'acord al tipus concret de BD al qual s'accedeix. La resta d'instruccions del programa serà exactament igual, sempre que no s'usin extensions propietàries de l'SQL.

2.3.2 Establiment de la connexió

Un cop s'ha carregat correctament el controlador, l'aplicació es pot connectar remotament a algun servidor que conté la BD mitjançant la crida al mètode estàtic `getConnection` de la classe `DriverManager` que es descriu a continuació. Aquesta classe ofereix els serveis bàsics de gestió de controladors JDBC i és el punt d'accés a ells des dels vostres programes.

La definició d'aquest mètode és:

```
1 Connection getConnection(String url, String user, String psw) throws
      SQLException
```

Un URL és el que s'escriu en la barra d'adreces d'un navegador web per accedir a una pàgina web concreta.

El paràmetre "url" és una cadena de text amb l'identificador de la ubicació de la BD. Quan es configura una BD, aquesta normalment ens informa de quin és aquest identificador, que dependrà en part de la màquina on s'ha instal·lat.

Un **URL** (*uniform resource locator*, localitzador uniforme de recurs) és un apuntador a algun recurs o servei disponible a internet. Aquest recurs pot ser quelcom tan simple com un fitxer, o elements més complexos com objectes, un servidor web o, és clar, una BD accessible remotament.

Adreça IP

Es tracta d'un identificador únic per a cada màquina connectada a internet. Consta de 4 bytes i normalment es representa amb cada byte en notació decimal, separats per punts. Per exemple: 192.168.0.34

A nivell general, normalment, un URL es caracteritza perquè està dividida en fragments prou significatius:

- El protocol que cal usar per accedir al recurs. O sigui, quin serà el format de les dades que arribaran al servei, de manera que les pugui interpretar correctament.
- El nom de la màquina, o la seva l'adreça IP, de manera que s'identifica a l'equip on es troba disponible el recurs o el servei.

- El port del servei, un identificador únic assignat a tots els serveis que s'executen en la màquina, de manera que és possible dirigir peticions a un servei concret donada una màquina.
- Finalment, especifica el nom del recurs pròpiament dins la màquina.

Així, doncs, un URL és, per exemple:

```
1 http://ioc.xtec.cat:80/educacio/ioc-estudis
```

Indica que, mitjançant el protocol anomenat HTTP, el que usa la web, cal accedir al port 80, on se sol executar el servidor web, i obtenir el recurs "/educacio/ioc-estudis", que és una pàgina web.

En el cas d'un URL per accedir a una BD mitjançant JDBC, té la particularitat que la part del protocol es divideix en dos: el protocol principal, que és "jdbc" i el subprotocol. Sense entrar en detall, es tracta d'un identificador que especifica el nom del controlador o mecanisme de connectivitat a la BD que cal usar. Per exemple, per a un servei de BD Apache Derby, el subprotocol és "derby". Cal mirar la documentació de la BD. La resta de la URL és idèntica a qualsevol altra: nom de màquina, port on s'executa el servei de la BD i nom de la BD concreta a què es vol accedir.

Per accedir a la BD anomenada "gestioEncarrecs" a un servei de BD de tipus Apache Derby que s'executa al port 1527 de la màquina amb adreça IP 192.168.2.1, la URL seria la següent:

```
1 jdbc:derby://192.168.2.1:1527/GestioEncarrecs
```

L'identificador `localhost` es pot usar si la màquina a la qual s'accedeix és la mateixa on s'executa l'aplicació.

Sempre que es desplega una BD, normalment es poden configurar comptes d'usuari, protegides amb contrasenya, de manera que no es permet que qualsevol aplicació aliena pugui llegir alegrement les dades contingudes. Per tant, perquè el programa pugui accedir a la BD, si està correctament protegida, també caldrà disposar d'un nom d'usuari i d'una contrasenya correctes, que és el que representen els paràmetres "user" i "psw". Si no hi ha cap compte d'usuari o contrasenya habilitat, cosa poc recomanable, es pot posar un text buit ("") als dos paràmetres.

Si la connexió remota s'estableix de manera correcta (la BD realment existeix, el servei està en marxa, correctament configurat i no hi ha cap problema en la xarxa), es retorna una instància de la classe `Connection`, a partir de la qual es pot interactuar per dur a terme qualsevol acció amb la BD. Per tant, el codi per connectar-se a la BD amb la URL anterior, si s'ha configurat amb un compte d'usuari "administrador", amb la contrasenya "pswdificil", seria:

```
1 String urlBaseDades = "jdbc:derby://localhost:1527/GestioEncarrecs";
2 String usuari = "administrador";
3 String contrasenya = "pswdificil";
4 Connection c = DriverManager.getConnection(urlBaseDades , usuari, contrasenya);
```

Val la pena comentar que, tot i que a l'exemple els paràmetres de la crida estan definits com a cadenes de text al mateix codi de l'aplicació, en una aplicació real

aquesta seria una solució poc encertada, ja que si mai canvia la ubicació de la BD (la màquina o el port) o el seu nom, caldria modificar el programa i tornar a compilar el programa. En la realitat, aquesta informació hauria de ser configurable, de manera que es pugui canviar sense haver de tocar el codi del programa. Per exemple, en un fitxer de configuració o com a paràmetre d'execució.

2.3.3 Execució de sentències SQL

Totes les interaccions amb la BD pràcticament sempre s'efectuen mitjançant l'enviament i l'execució de sentències SQL i el processament de les respostes. Les sentències SQL prenen la forma de simples cadenes de text que podeu generar al vostre gust (`CREATE TABLE...`, `INSERT...`, `SELECT.....`), sempre que sigui amb la sintaxi correcta. En aquest aspecte, res no varia.

Un cop es disposa de la cadena de text amb la sentència que es vol invocar a la BD, el mecanisme ofert per JDBC per enviar-les a través d'una connexió establerta és mitjançant l'ús de la classe `Statement` (sentència). Si es tracta d'una operació de consulta de dades, JDBC processa la resposta de la BD i la presenta dins el programa com un objecte del tipus `ResultSet`, del qual podem extreure la informació associada a partir dels mètodes que proporciona aquesta classe. Ambdues es troben al *package* `java.sql`.

La classe Statement

Les instàncies de la classe `Statement` només es poden instanciar mitjançant la crida al mètode `createStatement`, de la classe `Connection`. No es pot invocar directament el constructor usant el `new` (de fet, aquesta classe és, en realitat, una interfície Java).

És possible assignar valors per defecte usant el mètode sense paràmetres `createStatement()`.

```
1 Statement createStatement(int tipus, int concurrencia)
2 throws SQLException
```

Els paràmetres d'entrada especifiquen quines seran les propietats de les respostes de l'execució de la sentència, només per al cas d'executar consultes (`SELECT`). Dins la classe `ResultSet` és on es poden trobar definides el seguit de constants estàtiques que s'accepten com a paràmetre. Entre les més destacades es troben:

TAULA 2.27. Tipus de paràmetres d'entrada

Paràmetre	Constant	Descripció
tipus	`ResultSet.TYPE_FORWARD_ONLY`	La resposta només es pot navegar unidireccionalment, només endavant. És el que normalment s'usa.
tipus	`ResultSet.TYPE_SCROLL_INSENSITIVE`	La resposta només es pot navegar endavant i endarrere.
tipus	`ResultSet.TYPE_SCROLL_SENSITIVE`	Com l'anterior i, a més a més, si hi ha canvis a la mateixa BD mentre s'usa la resposta, aquests es reflecteixen a les dades que conté.
concurrència	`ResultSet.CONCUR_READ_ONLY`	Les dades contingudes a la resposta no es poden modificar (mode de lectura, només).
concurrència	`ResultSet.CONCUR_UPDATABLE`	Les dades contingudes a la resposta es poden modificar (mode lectura-escriptura).

Els mètodes sobrecarregats permeten recuperar valors per número de columna, segons l'ordre que ocupi en la taula...

Cal tenir en compte, però, que no tots els controladors permeten totes aquestes opcions.

Un cop s'ha inicialitzat correctament l'objecte i ja es disposa de la cadena de text amb la sentència SQL, aquesta sentència es pot enviar a la BD invocant el mètode corresponent. En el cas d'una consulta de dades (`SELECT`), cal emprar el mètode `executeQuery`, mentre que en qualsevol altre cas (`INSERT`, `UPDATE`, `DELETE`), quan es fan modificacions al contingut de la mateixa BD, cal usar el mètode `executeUpdate`. Ambdós estan sobrecarregats, de manera que permeten concretar certes particularitats de l'execució de la sentència. La crida més senzilla és la que té com a paràmetre, simplement, la cadena de text amb la sentència SQL a enviar.

```
1 ResultSet executeQuery(String sql) throws SQLException
2 int executeUpdate(String sql) throws SQLException
```

Si no es produeix cap excepció, llavors això voldrà dir que la sentència s'ha executat correctament. Fixeu-vos que només en el primer cas, quan es fa una consulta, realment es retorna un conjunt de dades. El valor de retorn d'una modificació sol ser 0, o el nombre de files que han estat manipulades per la sentència.

Per exemple, per consultar tots els clients que viuen a l'adreça 3 d'una taula, es faria:

```
1 Connection c = ...
2
3 Statement s = c.createStatement(ResultSet.TYPE_FORWARD_ONLY, ResultSet.
      CONCUR_READ_ONLY);
4 String sentencia= "SELECT * FROM CLIENTS WHERE APOSTAL='Adreça3'";
5 ResultSet res = c.executeQuery(sentencia);
```

D'altra banda, si es vol afegir un nou client, llavors caldria fer:

```
Connection c = ...

Statement s = c.createStatement(ResultSet.TYPE_FORWARD_ONLY, ResultSet.
    CONCUR_READ_ONLY);
String sentencia= "INSERT INTO CLIENTS VALUES (5,'Client5',
                  'Adreça5','e_mail5@domini.com','+34933336677', 3)";
int res = c.executeUpdate(sentencia);
```

Estrictament, la dificultat en l'execució de la sentència mitjançant la classe `Statement` recau íntegrament en la vostra habilitat per crear sentències SQL sintàcticament i semànticament correctes. Com passava en crear la connexió, recordeu que les sentències no han de ser necessàriament cadenes de text estàtiques definides dins el codi (segurament, poques vegades ho siguin), també poden dependre d'altres variables o d'altres paràmetres. Per exemple:

```
private static int LAST_ID = 0;

private void crearClient(String nom, String ad, String, String ml, String tlf,
    int nc)
throws SQLException {

  Connection c = ...
  ...

  Statement s = c.createStatement(ResultSet.TYPE_FORWARD_ONLY, ResultSet.
      CONCUR_READ_ONLY);
  LAST_ID++;
  String sentencia= "INSERT INTO CLIENTS VALUES (" + LAST_ID + ",'" +
                    nom + "','" +
                    ad + "','" +
                    ml + "','" +
                    tlf + "'," +
                    nc + ")";
  int res = c.executeUpdate(sentencia);
  ...
}
```

La classe PreparedStatement

Mitjançant la classe `Statement` es pot executar qualsevol sentència SQL sense límit. Ara bé, molt sovint, en un programa que accedeix a una BDD, l'usuari no disposarà de la possibilitat de fer qualsevol consulta que es pugui imaginar, sinó que es trobarà limitat a les funcionalitats que proporciona la seva interfície d'usuari. Així doncs, per exemple, en un programa que gestiona encàrrecs de clients usant una interfície gràfica, hi haurà un seguit de menús o botons que enumeraran un seguit d'operacions finites i molt concretes que es poden dur a terme: llistar noms de clients ordenats alfabèticament, cercar els encàrrecs pendents d'un client... Mitjançant sentències SQL es poden fer consultes ben complicades, però el programa ja té dins el seu codi un conjunt que són les úniques que s'usaran mai. L'usuari normalment no haurà de treballar directament amb el llenguatge SQL (SELECT, INSERT...), només indicar certs paràmetres molt concrets de la consulta (el nom del client a cercar, les dades d'un client que es volen llistar).

La classe `PreparedStatement` és una subclasse de `Statement` que permet executar sentències parametritzades, de manera que facilita aquest tipus de compor-

tament dins un programa. La seva particularitat principal és l'eficiència superior en relació amb `Statement`, ja que en el moment de definir-la es precompila, de manera que estalvia feina a la BDD. Per tant, és preferible usar-la per a sentències senzilles que depenen de paràmetres molt concrets aportats per l'usuari, i que cal usar repetides vegades, només canviant aquests paràmetres, al llarg de l'execució del programa.

Per instanciar un objecte `PreparedStatement`, també cal cridar un mètode estàtic de la classe `Connection` (n'hi ha diversos, en estar sobrecarregat). Ara bé, en aquest cas, hi ha un paràmetre addicional, que és la sentència SQL parametritzada. En aquesta sentència, el text no està complet, sinó que s'escriu un interrogant ('?') a cada lloc on es vol ubicar un paràmetre. Per exemple, suposem que un programa vol usar aquesta classe per dur a terme la funció de cercar les dades d'un client donat el seu nom.

```
1 Connection c =...
2
3 //Ara el text de la sentència es posa en crear-la, no en executar-la
4 String sentencia= "SELECT * FROM CLIENTS WHERE NOM=?";
5 PreparedStatement s = c.preparedStatement(sentencia);
```

Alerta, els valors dels índexs dels paràmetres van de 1..N, no de 0..N com en altres classes que gestionen llistes.

Atès que la sentència parametritzada no és vàlida en si mateixa, abans de poder-la executar cal assignar valors als seus paràmetres, un per a cada interrogant dins el seu text. Internament, l'objecte `PreparedStatement` organitza els paràmetres ordenant-los amb un índex de 1 a N, de manera que el primer '?' és el paràmetre 1, el segon és el 2... Mitjançant un seguit de mètodes, permet assignar valors a cadascun dels índexs. Hi ha un mètode per a cada tipus de dades. Cal invocar-ne tants com paràmetres (nombre d'interrogants) hi ha al text de la sentència.

- `setString(int parameterIndex, String x)`
- `setInt(int parameterIndex, int x)`
- `setDouble(int parameterIndex, double x)`
- etc.

Evidentment, cal ser molt acurat i usar el mètode que correspon al tipus de dades d'acord a la definició de la BDD, de manera que la sentència final sigui correcta. Per exemple, si s'està fent una cerca per nom de client, en l'exemple anterior caldrà usar el mètode `setString` per assignar correctament el paràmetre, ja que la columna NOM és de tipus VARCHAR (una cadena de text). Un cop l'objecte està correctament inicialitzat amb els valors dels paràmetres adients, de manera que cap queda sense un valor assignat, es pot executar usant `executeQuery` o `executeUpdate`, com amb `Statement`. Aquest cop, però no cal definir cap sentència en cridar aquests mètodes.

```
//Es pregunta el nom a la interfície d'usuari
String nom = ...

s.setString(1, nom);
//Si "nom" és "Client1", ara la sentència equival a:
//"SELECT * FROM CLIENTS WHERE NOM='Client1'"

ResultSet res = s.executeQuery();
```

La classe ResultSet

La instància de `ResultSet` retornada per una crida `executeQuery` conté la llista de files resultant de la consulta a la BD. Si la consulta no ha obtingut cap resultat, la instància de `ResultSet` estarà buida, però mai no es donarà el cas que retorni una referència a `null`. Si hi ha cap error (la sentència SQL executada no era correcta), es produirà una excepció.

`ResultSet` ofereix els mètodes necessaris per navegar per la llista i accedir als valors emmagatzemats. Aquesta navegació sempre és fila per fila, però el mode d'accés variarà segons els paràmetres emprats en instanciar l'objecte `Statement` associat: unidireccional, de manera semblant a com ho faria un `Iterator`, o bidireccional. En qualsevol dels casos, l'objecte `ResultSet` disposa d'un apuntador intern on recorda quina és la posició actual. Mitjançant la invocació de certs mètodes (`next()` o `previous()`), es pot fer avançar o recular l'apuntador. Ambdós s'avaluen a `true` si la nova posició de l'apuntador després de desplaçar-se conté una fila vàlida, o `false` si ja ens hem passat de la llista, tant sigui per l'inici com o pel final.

En el cas d'un `ResultSet` unidireccional, un cop s'arriba al final, ja no es pot accedir més a les dades. Cal tornar a executar una consulta.

Aquest sistema té la particularitat que, a l'inici de tot, l'apuntador està una posició per endavant de la primera fila, de manera que la primera crida a `next()` sempre es posiciona a la primera fila. Per tant, si s'intenta obtenir cap dada des del `ResultSet`, només obtenir-la, sense posicionar-se almenys en alguna fila correcta, hi haurà un error.

Un cop posicionats en la fila que es vol llegir, és possible consultar el valor de les cel·les per a cada columna definida en la taula, usant el nom de la columna. Per fer-ho, cal cridar el mètode `getXXX` adequat segons el tipus de dades de la columna a consultar, usant com a paràmetre el nom de la mateixa columna. En escollir quin mètode usar per obtenir les dades d'una columna concreta, cal tenir en compte les equivalències entre tipus de dades SQL i Java. Per exemple, si la dada a la BD és de tipus `INTEGER`, no podeu usar un mètode per obtenir una cadena de text (`getString`), i viceversa. Si el valor emmagatzemat en la columna no es correspon amb el tipus de dades del mètode cridat, es produirà un error. Dins el `ResultSet` totes les dades ja s'hauran convertit del tipus original SQL a la BD a tipus tractables en el codi Java, per la qual cosa no cal fer cap conversió concreta, només ser conscients de les equivalències.

Existeix un mètode per a cada tipus de dades, per exemple:

- `String getString (String nomColumna)`
- `int getInt (String nomColumna)`

- `short getShort (String nomColumna)`
- `double getDouble (String nomColumna)`
- `java.sql.Date getDate(String nomColumna)`
- etc.

Cal anar amb compte, però, amb la possibilitat que el valor que hi ha a la cel·la de la BD sigui `NULL` i, per tant, invàlid. En el cas dels mètodes que retornen objectes (`String`, `Date`), aquests senzillament retornaran una referència a `null`, en aquest cas. Ara bé, en el cas de mètodes que retornen un tipus primtiu, el resultat serà 0 (o `false` per als booleans). En aquest cas, com que no és possible a simple vista saber si el 0 en qüestió és perquè el valor és realment un 0 o `NULL`, cal usar el mètode auxiliar `wasNull()`, immediatament després de la crida a `getXXX`, el qual ens dirà si el valor retornat a la darrera consulta era un valor `NULL` a la BD o no.

Si es vol llistar per pantalla la llista dels noms de tots els clients de la BD, es pot fer així:

```
1  Connection c = ...
2
3  Statement s = c.createStatement(ResultSet.TYPE_FORWARD_ONLY, ResultSet.
       CONCUR_READ_ONLY);
4  String sentencia= "SELECT NOM FROM CLIENTS";
5  ResultSet res = c.executeQuery(sentencia);
6
7  while (res.next() ) {
8    String nom = res.getString("NOM");
9    if (!res.wasNull())
10     System.out.println(nom);
11 }
```

Aquest codi faria exactament el mateix si, en lloc de consultar només els noms, a la sentència SQL es consultessin totes les dades (`SELECT *`), o qualsevol subconjunt de dades que inclogués el nom (`SELECT NOM, APOSTAL...`). Atès que la crida a `getString` especifica la columna que vol extreure del resultat, s'obtindrien sempre les mateixes dades per pantalla. Ara bé, recordeu que, si es vol fer un codi eficient i no carregar la BD innecessàriament, cal escriure sentències SQL que es limitin a consultar estrictament les dades que necessiteu i cap altra.

Finalment, val la pena comentar que encara que un `ResultSet` només sigui navegable unidireccionalment, per exemple perquè no queda més remei ja que la BDD no suporta l'execució de consultes que retornin navegació bidireccional, això no vol dir que us hàgiu de resignar i haver de recórrer diverses vegades les dades per fer diferents operacions amb les mateixes dades dins un programa. Recordeu que res no impedeix agafar els valors i desar-los en estructures Java que sí que permetin gestionar la informació de manera més còmoda, com ara Lists, Sets o Maps, i, a partir d'aquí, treballar amb elles.

```
Connection c = ...

Statement s = c.createStatement(ResultSet.TYPE_FORWARD_ONLY, ResultSet.
    CONCUR_READ_ONLY);
String sentencia= "SELECT NOM FROM CLIENTS";
ResultSet res = c.executeQuery(sentencia);

List<String> llistaNoms = new ArrayList<String>();

while (res.next() ) {
  String nom = res.getString("NOM");
  if (!res.wasNull())
    llistaNoms. add(nom);
}

//A "llistaNoms" tenim tots els noms resultants de la consulta
```

Modificacions a les dades via ResultSet

Si a l'hora d'instanciar l'objecte `Statement` s'ha definit que el `ResultSet` retornat és de lectura-escriptura (paràmetre tipus `ResultSet.CONCUR_UPDATABLE`), llavors també és possible modificar el contingut de les files que conté, de manera que la manipulació es trasllada a la BD. És una manera alternativa de fer actualitzacions a la BD des d'una perspectiva més d'orientació a objectes, manipulant instàncies, en lloc d'usar sentències de text SQL.

Aquest sistema és possible ja que la generació d'un `ResultSet` no es basa a executar la resposta a la BD, obtenir absolutament totes les dades corresponents, encapsular-les, retornar-les al codi Java via JDBC, i oblidar-nos de la BD. En realitat, internament, un `ResultSet` és una associació viva amb la BD, que va obtenint les dades a poc a poc des de la BD a través de la connexió, a mesura que es van consultant les files de la resposta. Això permet que, igual que es pot llegir, també es pugui escriure a través d'aquesta associació. O que, si hi ha canvis a la BD, surtin immediatament també al `ResultSet` sense haver d'esperar a fer una altra consulta (si s'ha usat el paràmetre `ResultSet.TYPE_SCROLL_SENSITIVE`).

Com amb una simple consulta, cal posicionar-se a la fila a modificar. En aquest cas, els mètodes d'escriptura s'anomenen `updateXXX`, de manera anàloga als de lectura. En termes generals, tot el que s'aplica als mètodes de lectura, com el tractament de tipus de dades Java-SQL, s'aplica a aquests.

- `void updateString (String nomColumna, String s)`
- `void updateInt (String nomColumna, int i)`
- `short updateShort (String nomColumna, short s)`
- `double updateDouble (String nomColumna, double d)`
- `void updateDate(String nomColumna, java.sql.Date d)`
- ...

No obstant això, si es vol escriure un valor NULL, el que cal usar és el mètode updateNull(String nomColumna), independentment del tipus de dades desat a la columna.

Ara bé, un cop feta la modificació al ResultSet, en ser un conjunt de dades locals a l'aplicació, cal avisar-lo que forci una actualització sobre la BD. Això es fa amb el mètode updateRow(). Aquest mètode actualitza la informació només de la fila on es troba actualment l'apuntador i cap altra.

Per exemple, el codi següent passa a majúscules els noms de tots els clients de la BD, aprofitant els mètodes que ofereix Java per manipular cadenes de text. Fixeu-vos que ara el segon paràmetre de creació de l'Statement és ResultSet.CONCUR_UPDATABLE.

```
Connection c = ...

Statement s = c.createStatement(ResultSet.TYPE_FORWARD_ONLY, ResultSet.
    CONCUR_UPDATABLE);
String sentencia= "SELECT NOM FROM CLIENTS";
ResultSet res = c.executeQuery(sentencia);

while (res.next() ) {
  String nom = res.getString("NOM");
  String nouNom = nom.toUpperCase();
  res.updateString("NOM", nouNom);
  res.updateRow();
}
```

Un cop més, cal ser curós de no consultar més dades a la BD de les imprescindibles per fer la tasca encomanada (no fer SELECT * si realment no hem de treballar amb totes les dades al final).

A part de modificar dades, també és possible afegir i eliminar files. Per al primer cas, s'usa primer el mètode moveToInsertRow(), per indicar que la posició actual es considera ara una fila nova, una successió de crides a updateXXX per posar les dades a cada columna, i finalment el mètode insertRow(). Per eliminar la fila actual, només cal invocar el mètode deleteRow().

```
res.moveToInsertRow();
  res.updateInteger("ID", 5);
  res.updateString("NOM", "Client5");
  res.updateString("APOSTAL","Adreça5");
  res.updateString("AELECTRONICA","e_mail5@domini.com");
  res.updateString("TELÈFON","+34933336677");
  res.updateInteger("NCOMANDES",3);
  res.inserRow();
```

2.3.4 Tancament de la connexió

El manteniment d'una connexió oberta a una BD és una tasca costosa per al sistema, tant pel vostre programa com pel servei de BD a l'altre extrem de la connexió; per tant, quan ja no s'ha d'usar més, cal tancar-la, de la mateixa manera que cal tancar un flux després d'haver-ne llegit tota la informació. El tancament s'efectua, simplement, cridant sobre la connexió (l'objecte Connection) el mètode close:

```
Connection c = ...
...
c.close();
```

En qualsevol cas, si un se n'oblida, la connexió no es manté oberta indefinidament, ja que quan el recol·lector de memòria de Java elimini l'objecte `Connection` generat també s'encarregarà de tancar la connexió. De totes maneres, aquest mecanisme és només un últim recurs que proporciona Java en cas que us hàgiu despistat. Cal que sempre tanqueu vosaltres la connexió al codi.

Atès que el procés de creació d'una connexió també és costós, les aplicacions que fan ús de l'accés a una BD normalment obren una connexió en iniciar-se i no la tanquen fins tot just abans d'acabar-ne l'execució. De totes maneres, si es preveu que l'aplicació no farà ús de la connexió en un interval llarg de temps, val la pena tancar-la i tornar-la a obrir quan realment torni a fer falta, ja que si el servei de la BD té limitat el nombre de connexions, pot ser que el nostre programa estigui impedint que un altre s'hi connecti, tot i no estar accedint realment a la BD.

2.3.5 Exemple d'aplicació JDBC: El gestor d'encàrrecs

Tot just es mostra el codi d'un exemple d'aplicació que usa JDBC per gestionar informació a una BD. Concretament, aquest programa gestiona objectes de tipus `Client` de manera que les seves dades es troben a una BD a través de la qual es poden fer cerques o afegir-ne. Es composa de tres classes:

- `Client`, que defineix quina informació conté un client.
- `GestorBD`, que conté els mètodes associats a l'accés a la base de dades.
- `GestorEncarrecs`, que és la classe principal i ofereix la interfície d'usuari (per consola).

Abans de poder-la executar correctament, es clar, s'ha d'haver configurat correctament una BD d'acord al codi de connexió a `GestorBD`.

```
//Fitxer Client.java
public class Client {
  private int id;
  private String nom;
  private String apostal;
  private String aelectronica;
  private String telefon;

  public Client(int i, String n, String ap, String ae, String t){
    id = i;
    nom = n;
    apostal = ap;
    aelectronica = ae;
    telefon = t;
  }

  public int getId() { return id; }
```

```
  public String getNom() { return nom; }
  public String getAPostal() { return apostal; }
  public String getAElectronica() { return aelectronica; }
  public String getTelefon() { return telefon; }

  @Override
  public String toString() {
    return id + "\t" + nom + "\t" + apostal + "\t" + aelectronica + "\t" +
        telefon ;
  }
}
```

```
import java.util.*;
import java.sql.*;

//Aquesta classe fa el mecanisme de persistència independent de la GUI.
public class GestorBD {
  Connection conn;

  public GestorBD() throws Exception {
    Class.forName("org.apache.derby.jdbc.ClientDriver");
    conn = DriverManager.getConnection("jdbc:derby://localhost:1527/
        GestioEncarrecs", "administrador", "pswdificil");
  }

  public void tancar() throws Exception {
    conn.close();
  }

  public int obtenirNouIDClient() throws Exception {
    //Cercar ID maxim
    Statement cercaMaxId = conn.createStatement();
    ResultSet r = cercaMaxId.executeQuery("SELECT MAX(ID) FROM CLIENTS");
    if (r.next()) return (1 + r.getInt(1));
    else return 1;
  }

  public List<Client> cercarClient(String nom) throws Exception {
    Statement cerca = conn.createStatement();
    ResultSet r = cerca.executeQuery("SELECT * FROM CLIENTS WHERE NOM='" + nom
        +  "'");
    LinkedList<Client> llista = new LinkedList<Client>();
    while (r.next()) {
      llista.add(new Client(r.getInt("ID"),r.getString("NOM"),r.getString("
          APOSTAL"), r.getString("AELECTRONICA"),r.getString("TELEFON")));
    }
    return llista;
  }

  public void afegirClient(Client c) throws Exception {
    Statement update = conn.createStatement();
    String valors = c.getId() + ",'" + c.getNom() + "','" + c.getAPostal() +  "
        ','" + c.getAElectronica() + "','" + c.getTelefon() + "'";
    update.executeUpdate("INSERT INTO CLIENTS VALUES(" + valors + ")");
  }
}
```

```
//Fitxer GestorEncarrecs.java

import java.io.*;
import java.util.*;

//Classe Principal
public class GestorEncarrecs {

  GestorBD gestor;
  BufferedReader entrada;

```

```
12  public static void main(String[] args) throws Exception {
13    GestorEncarrecs gbd = new GestorEncarrecs();
14    gbd.start();
15  }
16
17  public GestorEncarrecs() throws Exception{
18    gestor = new GestorBD();
19    entrada = new BufferedReader(new InputStreamReader(System.in));
20  }
21
22  public void start() throws Exception {
23    int opcio;
24    while (0 != (opcio = menuPrincipal())) {
25      try {
26        switch (opcio) {
27          case 1:
28            cercarClient();
29            break;
30          case 2:
31            afegirClient();
32            break;
33          default: mostrarDades("Opció incorrecta\n");
34        }
35      } catch (Exception ex) {
36        mostrarDades("S'ha produït un error: " + ex + "\n");
37      }
38    }
39    gestor.tancar();
40  }
41
42  private int menuPrincipal() throws Exception {
43    String menu = "\nQuina acció vols realitzar?\n" + "[1] Cercar client\n"+  "
          [2] Afegir client\n" +  "[0] Sortir\n" + "Opció>";
44    String lin = entrarDades(menu);
45    try { int opcio = Integer.parseInt(lin); return opcio; }
46    catch (Exception ex) { return -1; }
47  }
48
49  //Amb els metodes entrarDades i mostrarDades, fem el codi independent
50  //de la interficie. Si mai es fan canvis, nomes cal canviar aquests
51  //dos metodes.
52
53  private String entrarDades(String pregunta) throws IOException {
54    mostrarDades(pregunta);
55    return entrarDades();
56  }
57
58  private String entrarDades() throws IOException {
59    String linia = entrada.readLine();
60    if ("".equals(linia)) return null;
61    return linia;
62  }
63
64  private void mostrarDades(String dades) throws IOException {
65    System.out.print(dades);
66  }
67
68  //Cercar un element d'acord al seu nom
69  private void cercarClient() throws Exception {
70    String nom = entrarDades("Introdueix el nom del client: "); if (null == nom
          ) return;
71    List<Client> llista = gestor.cercarClient(nom);
72    Iterator it = llista.iterator();
73    mostrarDades("Els clients trobats amb aquest nom son:\n
          ----------\n");
74    while (it.hasNext()) {
75      Client c = (Client)it.next();
76      mostrarDades(c.toString() + "\n");
77    }
78  }
```

```
79
80  //Afegeix un nou client
81  public void afegirClient() throws Exception {
82     mostrarDades("Introdueix les seguents dades del nou client (deixa en blanc
           per sortir).\n");
83     String nom = entrarDades("Nom: "); if (null == nom) return;
84     String apostal = entrarDades("Adreça postal: "); if (null == apostal)
           return;
85     String aelectronica = entrarDades("E-mail: "); if (null == aelectronica)
           return;
86     String telefon = entrarDades("Telefon: "); if (null == telefon) return;
87     int id = gestor.obtenirNouIDClient();
88     gestor.afegirClient(new Client(id,nom,apostal,aelectronica,telefon));
89     mostrarDades("Operació completada satisfactòriament.\n");
90   }
91 }
```

2.4 Seguretat en l'accés a la BD

Una BD que s'executa en una màquina és una porta d'accés a totes les dades emmagatzemades, les quals moltes vegades són confidencials (noms, adreces, etc.). Protegir aquestes dades, de manera que només les persones realment autoritzades les puguin llegir, és una tasca molt important, no tan sols de l'administrador del SGBD (per exemple, configurant un nom d'usuari i contrasenya prou segurs), sinó del mateix desenvolupador d'aplicacions. Si una aplicació que accedeix a una BD no es codifica correctament, pot succeir que persones no autoritzades poden arribar a accedir a les dades aprofitant males pràctiques en el codi. Per tant, quan es treballa amb una BD és molt important programar de manera segura.

> Mitjançant la disciplina de la **programació segura**, es pot donar un cert nivell de garantia que una aplicació es comportarà sempre de la manera esperada i un usuari amb males intencions no serà capaç de comprometre el sistema aprofitant errors en el codi.

Aquesta disciplina és molt extensa, però no es pot finalitzar aquest capítol sense mostrar algun exemple significatiu dels aspectes que cal tenir molt en compte en crear una aplicació que accedeix a una BD.

Un dels punts més importants en desenvolupar aplicacions d'aquest tipus és que, quan s'executen sentències SQL, en cap concepte aquestes han d'incloure directament cap tipus d'entrada de l'usuari. Totes les entrades s'han de validar prèviament i s'ha de comprovar si tenen el format esperat.

Suposem el fragment de codi següent, que permet fer cerques sobre la taula donats noms coneguts de clients:

```
...
String nom = entrarDades("Introdueix el nom del client");
Statement cerca = c.createStatement();
ResultSet r = cerca.executeQuery(
"SELECT * FROM CLIENTS WHERE NOM='" + nom +  "'");
System.out.println("Els clients trobats son:");
while (r.next()) {
...
```

Si l'usuari introdueix la cadena "Client1", s'executa la sentència SQL:

```
SELECT * FROM CLIENTS WHERE NOM='Client1'
```

De manera que, en fer l'accés a la BD, l'aplicació retorna una sortida de l'estil:

Els clients trobats són:

Taula 2.28.

Client1	Adreça1	email1@domini.com	+34931112233	4

Si hi ha més d'un client amb el nom "Client1", es llisten diverses files de la taula. Fins aquí, tot correcte. També, d'acord amb aquest fragment de codi, un usuari no pot accedir a les dades d'un client si no en coneix el nom, ni molt menys llistar tota la taula de cop. Sempre es podrien provar totes les combinacions possibles de noms, però intuïtivament ja es veu que aquesta via d'acció no és realment factible. Fins i tot en cas que es vagin provant noms a l'atzar, mai no es pot estar segur de tenir realment tot el contingut de tota la taula.

Ara bé, en aquest codi es comet el gegantí error d'utilitzar directament l'entrada de l'usuari per formar una sentència SQL de consulta (`SELECT`). Suposem que com a nom d'usuari s'introdueix la cadena de text següent: "Client1' OR '1'='1".

Atès que el codi traspassa directament l'entrada de l'usuari a la sentència SQL mitjançant una concatenació de cadenes de text, llavors la sentència que realment s'executaria seria:

```
SELECT * FROM CLIENTS WHERE NOM='Client1' OR '1'='1'
```

Ara l'expressió que s'executa és una mica estranya i inesperada, però és el resultat d'aplicar exactament el procés que s'ha programat. En aquest cas, com '1'='1'avalua cert i una expressió OR amb una de les seves entrades a cert sempre retorna també cert, això equival a fer:

```
SELECT * FROM CLIENTS
```

Per tant, s'acaba de llistar tota la taula de clients de manera inesperada a causa de la manca de comprovació del format de l'entrada de l'usuari abans de traspassar-la a una sentència SQL. Això és un error molt greu per part del desenvolupador. L'administrador del SGBD no hi pot fer res que protegeixi la màquina on es troba la BD contra aquest error, ja que es tracta d'un error de programació, no de configuració de la BD.

Malauradament, aquesta vulnerabilitat és molt comuna (entre les cinc primeres del rànquing mundial), especialment en aplicacions basades en formularis web. Es coneix com a **SQL-Injection**.

Una eina útil per fer comprovacions en el format de cadenes de text d'entrada són les expressions regulars.

Per evitar aquest problema, cal que el codi dels vostres programes processi qualsevol cadena de text que depèn d'una entrada de l'usuari abans de permetre que formi part del text d'una sentencia SQL. Per exemple, es pot veure si les dades introduïdes tenen el format esperat. No hi ha gaires telèfons o adreces de correu electrònic amb apòstrofs o símbols d'igualtat. Una altra opció és marcar els apòstrofs que hi ha dins una entrada de dades de l'usuari de manera que la BD no els interpreti com part de la sintaxi SQL, sinó com un caràcter qualsevol estrictament. Cada sistema de BD ofereix la seva manera d'escapar caràcters i diferenciar entre un caràcter especial d'SQL o un que són dades estrictament.

Una altra opció és usar sentències parametritzades, `PreparedStatement`, ja que aquestes són immunes a aquest atac.

3. Aplicacions amb BD orientades a objectes

Si bé les BD relacionals són les més populars i les que tenen més acceptació, la seva utilització dins una aplicació orientada a objectes implica un procés de traducció del diagrama UML original a un model relacional, totalment basat en taules. En aquesta traducció es perden moltes de les funcionalitats bàsiques de l'orientació a objectes, que s'han de simular d'alguna manera: referències a objectes, classes associatives, llistes d'objectes, herència, etc. Quan el diagrama és de certa complexitat, la traducció pot esdevenir molt complicada.

Per resoldre aquest problema hi ha les BD orientades a objectes (BDOO). Aquestes, en lloc d'organitzar les dades en taules, les organitzen exactament tal com ho fa un diagrama UML, mitjançant la definició del conjunt de classes i relacions entre elles. Per tant, no cal fer cap traducció.

Per evitar confusions, fem servir el terme *BDR* per referir-nos explícitament a una BD relacional i el terme *BDOO* per referir-nos a una BD orientada a objectes, de manera que ambdós quedin diferenciats.

El Java té una especificació per a BDOO anomenada JDO (*Java data objects*).

Actualment, l'aplicació de BDOO es limita a àmbits molt concrets, especialment els vinculats a àrees científiques. La seva implantació en aplicacions comercials d'àmbit general és molt baixa. Un dels problemes principals de les BDOO és que els fabricants tendeixen a crear solucions incompatibles, que no obeeixen cap especificació concreta. Al contrari que en el cas de les BDR, és molt possible que una aplicació client feta pel producte d'un fabricant concret no funcioni sobre una BDOO d'un altre fabricant. De fet, a les BDOO que suporten Java no s'accedeix mitjançant JDBC, ja que aquest mecanisme és específic per a BDR, sinó que normalment s'hi accedeix usant biblioteques específiques per a cada fabricant.

3.1 Els llenguatges ODL i OQL

De la mateixa manera que hi ha l'SQL com a llenguatge estàndard per accedir a una BDR independentment del fabricant, hi ha un llenguatge per accedir a les dades d'una BDOO: el llenguatge de consultes a objectes (*object query language*, OQL). Addicionalment, hi ha el llenguatge de descripció d'objectes (*object description language*, ODL), que serveix per especificar el format d'una BDOO: quina mena d'objectes pot contenir i les seves relacions. Malauradament, si bé aquests llenguatges estan especificats, i com ja s'ha dit, no es pot comptar amb el fet que qualsevol fabricant realment els suporti. De cap manera arriben al grau d'acceptació de l'SQL.

3.1.1 El llenguatge ODL

El llenguatge ODL s'utilitza per definir classes d'objectes persistents dins una BDOO, de manera que els seus objectes es puguin emmagatzemar. Dins la declaració de cada classe s'inclou:

- El nom de la classe.
- Declaracions opcionals de claus primàries.
- La declaració de l'extensió: el nom del conjunt d'instàncies existents.
- Declaracions d'elements: atributs, relacions o mètodes.

La sintaxi és la següent (entre claudàtors s'indiquen camps opcionals):

```
class nomClasse [(key nomAtribut)] {
attribute tipusAtribut nomAtribut;
...
relationship tipus<nomClasseDestinació> nomRelacio;
...
tipusRetorn nomMetode(params) [raises (tipusExcepcio)]
...
}
```

Com es pot apreciar, simplement és un canvi de sintaxi respecte al llenguatge Java pròpiament, però la majoria d'elements d'una classe són clarament identificables.

L'única diferència és la declaració explícita de les relacions en forma de la paraula clau relationship, en contrast en Java, que es tradueixen a atributs. Hi ha diferents tipus de relacions segons la cardinalitat que es vol expressar. De fet cadascun d'aquests tipus té una certa correspondència amb les classes que s'usen en Java per implementar relacions. Per a cardinalitat 1, és suficient de posar el nom de la classe destinació. En cas de cardinalitat múltiple, es pot triar entre diferents tipus:

- `<nomClasseDestinació>`, si la relació és només a un únic objecte.
- `Set<nomClasseDestinació>`, un conjunt no ordenat sense repeticions.
- `Bag<nomClasseDestinació>`, un conjunt no ordenat amb repeticions.
- `List<nomClasseDestinació>`, un conjunt ordenat amb repeticions, amb insercions eficients.
- `Array<nomClasseDestinació>`, un conjunt ordenat amb repeticions.

El seu significat és el mateix que el de les classes homònimes del Java (capacitat d'haver-hi repeticions d'elements, ordenades o no per índex, etc.). Normalment, la més usada és `Set<nomClasseDestinació>`.

Si retornem a un model orientat a objectes, amb referències, no cal l'atribut "id".

Tot seguit es mostra com es podria representar dues classes interdependents anomenades `Client` i `Encarrec`, que emmagatzemen dades a una aplicació de gestió de clients, mitjançant ODL.

```
class Client (key id) {
 attribute int id;
 attribute String nom;
 attribute String adreçaPostal;
 attribute String adreçaMail;
 attribute String telefon;

 relationship Set<Encarrec> encarrecs;

 String getId();
 ...
}

class Encarrec (key id)  {
 attribute int id;
 attribute Date data;
 ...
}
```

L'herència entre classes s'inicia en la seva declaració mitjançant la paraula clau `extends` seguit del nom de la superclasse:

```
class nomClasse extends nomSuperClasse {
  ...
}
```

3.1.2 El llenguatge OQL

El llenguatge OQL es limita a permetre consultes sobre una BDOO. El seu operador principal és `SELECT`, el qual té una gran similitud amb l'equivalent SQL. Tot i així, té algunes particularitats degudes a la manera com s'estructuren les dades mitjançant l'orientació a objectes (per exemple, no hi ha taules, és clar).

La sintaxi general és:

```
SELECT valor1,valor2,...
FROM llista de col.lecions i noms per membres típics
WHERE condició
```

Atès que ara ja no hi ha taules, cal tenir en compte dues coses. D'una banda, la llista de col·leccions especificada en l'apartat `FROM` correspon a alguna de les classes declarades. Juntament al nom d'aquesta classe s'especifica la variable que s'usarà en els termes `SELECT` i `WHERE` per referenciar valors. D'altra banda, els valors que es volen consultar o comparar són atributs de classes, pel que la manera de referir-s'hi és mitjançant la nomenclatura: `nomClasse.nomAtribut`. Aquesta possibilitat també es pot usar per indicar-ne les relacions.

Per exemple, a una aplicació de gestió de clients, si es volen consultar els clients de la BDOO d'acord amb la definició de les seves classes, es pot fer:

```
SELECT c.adreçaPostal, c.telefon
FROM Clients c
WHERE c.nom = "Client1"
```

Aquesta consulta retorna l'adreça postal i el telèfon del client amb nom "Client1".

També és possible accedir als encàrrecs per mitjà dels clients, seguint la seva relació:

```
SELECT e.data
FROM Clients c, c.encarrecs e
WHERE c.adreçaMail = "email1@domini.com"
```

Aquesta consulta retorna la data de tots els encàrrecs del client amb adreça de correu "email1@domini.com".

3.2 La llibreria db4O

Tot i els esforços per estandarditzar l'ús de les BDOO, no es pot dir que actualment hi hagi cap equivalent al llenguatge SQL. Tot i que sobre el paper hi ha l'ODL i l'OQL, a la pràctica ara mateix no hi ha cap llengua franca que es pugui garantir que està suportada, almenys en els seus aspectes fonamentals, per totes les bases de dades, tot i que després cada fabricant pugui afegir igualment les seves pròpies extensions propietàries. Per tant, cada tipus de BDOO ofereix el seu propi sistema per accedir als objectes emmagatzemats. Afortunadament, com aviat veureu, això no és gaire problemàtic, ja que l'avantatge d'usar una BDOO és poder crear codi on operar amb objectes persistents; és gairebé igual que treballar amb objectes a memòria, i, per tant, els mecanismes que ofereixen les diferents BDOO sovint són molt semblants a treballar amb objectes directament a memòria. El que varia són les llibreries de classes a usar, però no la idea general.

Les classes de db4o estan dins els *packages* `com.db4o...`

En aquest apartat es veurà un cas concret d'accés a una BDOO anomenada **db4o**, de lliure distribució, actualment amb versions per al Java i per a .NET. Evidentment, aquesta secció se centra exclusivament en la versió per al Java. No es troba en la distribució estàndard del Java, i, per tant, s'ha de descarregar i afegir als vostres projectes a part.

La llista de passos per interactuar amb aquesta BDOO és molt semblant a l'emprada mitjançant JDBC, si bé la manera com es fa amb codi Java és completament diferent en alguns aspectes.

Un cop s'han emmagatzemat objectes a una **BDOO db4o**, ja no es pot modificar la classe d'aquests objectes. Si es modifica (i, per tant, es torna a compilar el fitxer JAVA), totes les dades que hi ha dins deixen de tenir validesa. Si es vol usar la nova versió de la classe, cal tornar a generar el contingut de la BDOO des de zero.

3.2.1 Obertura de la BDOO

Normalment, els fitxers de db4o s'escriuen amb l'extensió ".db4o"

Una BDOO *db4o* no és més que un fitxer, si bé força complex en la seva estructura interna. Per tant, alguns dels aspectes són semblants, com ara l'accés a les dades emmagatzemades dins, que és semblant a com es faria amb un fitxer qualsevol. Aquest és el cas de la seva obertura, per tal de poder llegir les seves dades o escriure'n, i el seu posterior tancament quan ja no cal usar-lo més. El mètode estàtic `openFile` de la classe `Db4oEmbedded` permet que es pugui obrir aquest fitxer. Com succeeix en treballar amb fitxers, si aquest no existeix, se'n crea un de nou, amb la BDOO buida.

```
1 import com.db4o.*;
2 ...
3 ObjectContainer db = Db4oEmbedded.openFile("BDOOClients.db4o");
4 //Accions amb la BDOO
5 db.close();
```

Mitjançant l'objecte resultant de la crida, una instància d'`ObjectContainer` , es podran dur a terme totes les accions amb la BDOO. Fins a cert punt, és l'equivalent a una connexió a una BDR mitjançant JDBC. Un cop s'ha acabat de treballar, sempre cal tancar la BDOO usant el mètode `close`.

Ara bé, aquesta és una aproximació molt simple a l'ús d'una BD, ja que el fitxer ha d'estar emmagatzemat en local a la mateixa màquina que executa l'aplicació. Per poder tractar les dades des d'una altra màquina, caldrà copiar tant l'aplicació com també el fitxer de la BD. Normalment, el model de treball amb una BD es basa que hi ha un servidor, on s'executa la BD, i el desenvolupador genera el client, que s'hi connecta per xarxa. Un cop establerta la connexió, pot enviar peticions a la BD, i un cop dutes a terme totes les tasques, la tanca.

Les llibreries de *db4o* no proporcionen cap servidor en forma de programa que només cal instal·lar i configurar en un equip. Afortunadament, aquesta tasca és molt simple, ja que es pot fer en poques línies de codi. Dins aquest model d'accés a les dades, la BD continua essent un únic fitxer, en aquest cas emmagatzemat al servidor central, i les llibreries ja proporcionen tots els mecanismes necessaris per publicar el servei a la màquina i poder accedir-hi remotament de manera transparent, com si en realitat fos un fitxer en local en l'equip que executa l'aplicació client.

Per posar en marxa un servidor, cal usar el mètode estàtic `openServer` de la classe `Db4oClientServer` (al *package* `com.db4o.cs`). Un cop s'executa el mètode, el servei d'accés a la DBOO es posa en marxa a l'equip local on s'ha executat. Mentre el programa segueixi en execució, estarà disponible. Aquest mètode requereix tres paràmetres:

- Un nou objecte de configuració del servidor, que sempre es genera cridant `Db4oClientServer.newServerConfiguration()`.

- El nom del fitxer on es desa la BDOO. Si no existeix, se'n crearà un de nou buit.
- El port on s'executarà el servei.

Sovint, en engegar o apagar un servei db4o apareix un missatge de depuració per la consola d'errors del Java.

La creació d'un servidor retorna un objecte `ObjectServer`, a partir del qual es poden configurar certs aspectes del comportament del servei. El més important de tots és poder afegir usuaris i contrasenyes que limitin qui pot accedir a la BDOO remotament. Això es fa usant el mètode `grantAccess`.

Vegem un exemple de servidor *db4o*, molt senzill, però més que suficient per provar-ne el funcionament i els exemples d'aquest apartat si es desitja. En aquest cas, s'ha fet que el programa que executa el servei no finalitzi fins que l'usuari pitgi "Q" o "q". En fer-ho, el servei s'apaga i deixa d'estar accessible remotament.

```
1  import java.io.Scanner;
2  import com.db4o.*;
3  import com.db4o.cs.Db4oClientServer;
4
5  public class Server {
6    public static void main (String[] args) throws Exception {
7      ObjectServer sv = Db4oClientServer.openServer(Db4oClientServer.
           newServerConfiguration(), "BDRemota.db4o", 7000);
8      sv.grantAccess("usuari", "contrasenya");
9      Scanner in = new Scanner (System.in);
10     System.out.println("Pitja [Q] per tancar el servidor.");
11     while (in.hasNext()) {
12       if  ("Q".equalsIgnoreCase(in.next())) break;
13     }
14   }
15 }
```

Des del punt de vista de l'aplicació que vol accedir a la BD remota, cal usar el mètode `openClient` de la classe `Db4oClientServer`. Ara bé, calen alguns paràmetres per poder identificar on es vol accedir:

- L'identificador de la màquina remota.
- El port del servei, tal com s'ha configurat en fer `openServer`.
- Un nom d'usuari i una contrasenya vàlids.

Si algun d'aquests paràmetres no és correcte, es llançarà una excepció indicant que la connexió no s'ha pogut establir.

```
1  import com.db4o.*;
2  import com.db4o.cs.Db4oClientServer;
3  ...
4  ObjectContainer db = Db4oClientServer.openClient("lamevamaquina.domini.cat",
       7000, "usuari","contrasenya");
5  //Accions amb la BDOO
6  db.close();
```

Localhost

Per connectar-se a l'equip local, pel cas on tant el client com el servidor s'executen la mateixa màquina,es pot usar el nom de *host*, *localhost*. Això és útil per fer proves en un únic equip.

L'objecte retornat en aquest cas també és un `ObjectContainer`, per la qual cosa, un cop establerta la connexió amb la DB remota amb aquesta crida, totes les operacions que es poden dur a terme són exactament iguals que si es fessin accedint a un fitxer en local.

3.2.2 Emmagatzematge de nous objectes

Per emmagatzemar qualsevol objecte del vostre programa dins la BDOO, només cal cridar el mètode `store` que proporciona l'`ObjectContainer`, obtingut en obrir la BD (ja sigui en un fitxer local o remot). Aquest mètode només té un paràmetre, que és l'objecte a emmagatzemar. Mitjançant aquest mètode es pot desar qualsevol tipus d'objecte, sense que per aquest fet s'hagi de fer cap modificació al seu codi font.

Per exemple, suposeu que es volen gestionar els encàrrecs que duen a terme clients d'una empresa i, per a tal efecte, s'han generat les classes següents, que inicialment no es van desenvolupar amb el propòsit de ser integrades dins cap BDOO *db4o*. Noteu que la classe `Client` només permet canviar l'adreça electrònica un cop s'ha instanciat (usant el mètode `setAElectronica`).

```
1  import java.util.*;
2  public class Client {
3    private String nom;
4    private String aPostal;
5    private String aElectronica;
6    private String telefon;
7    private List<Encarrec> liComandes = new LinkedList<Encarrec>();
8    public Client(String n, String ap, String ae, String t) {
9      nom = n;
10     aPostal = ap;
11     aElectronica = ae;
12     telefon = t;
13   }
14   public String getNom() {
15     return nom;
16   }
17   public String getAPostal() {
18     return aPostal;
19   }
20   public String getAElectronica() {
21     return aElectronica;
22   }
23   public void setAElectronica(String ae) {
24     aElectronica = ae;
25   }
26   public String getTelefon() {
27     return telefon;
28   }
29   public int getNreComandes() {
30     return liComandes.size();
31   }
32   public void addComanda(Encarrec e) {
33     liComandes.add(e);
34   }
35   public List<Encarrec> getComandes() {
36     return liComandes;
37   }
38   @Override
39   public String toString() {
40     String res = nom + " : " + aPostal + " : (" + aElectronica + ", " + telefon
         + ")\n";
41     Iterator<Encarrec> it = liComandes.iterator();
42     while (it.hasNext()) {
43       Encarrec e = it.next();
44       res += e.toString() + "\n";
45      }
46     return res;
47   }
```

```java
import java.util.Date;

public class Encarrec {
  private String nomProducte;
  private int quantitat;
  private Date data;

  public Encarrec(String n, int q) {
    nomProducte = n;
    quantitat = q;
    data = new Date();
  }
  public String getNom() {
    return nomProducte;
  }
  public int getQuantitat() {
    return quantitat;
  }
  public Date getData() {
    return data;
  }

  @Override
  public String toString() {
    return getData()+ " – " + getNom() + " (" + getQuantitat() + ")";
  }
}
```

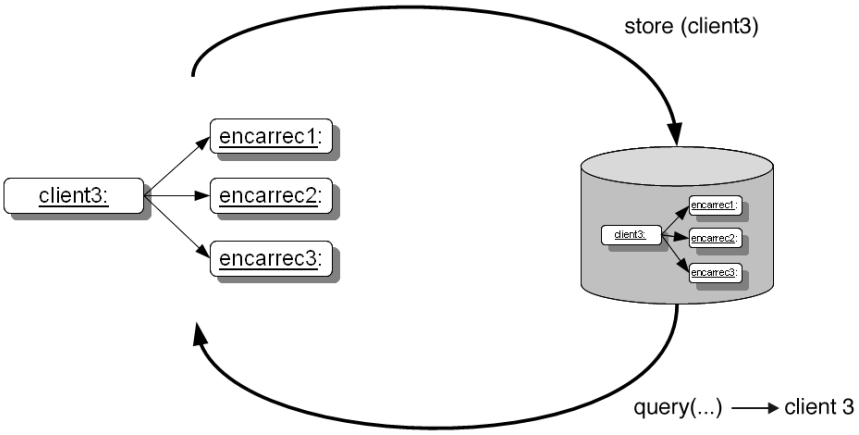
El codi d'un programa que emmagatzema quatre clients a la BDOO, entre els quals un d'ells ja té tres encàrrecs associats, seria el que hi ha a continuació, basat, senzillament, a fer crides successives a `store`, passant com a paràmetre cada objecte que es vol emmagatzemar. Per simplificar, se suposa que s'obre una BDOO ubicada en un fitxer local, però per al cas remot, seria exactament el mateix. Recordeu que sempre cal controlar les excepcions en accedir a les dades.

```java
import com.db4o.*;
public class EmmagatzemaClients  {
  public static void main(String[] args) throws Exception {
    ObjectContainer db = Db4oEmbedded.openFile("BDOOClients.db4o");
    try {
      Client[] clients = {
        new Client("Client1", "Adreça1", "e–mail1@domini.com", "+34931112233"),
        new Client("Client2", "Adreça2", "e–mail2@domini.com", "+34932223344"),
        new Client("Client3", "Adreça3", "e–mail3@domini.com", "+34931112233"),
        new Client("Client4", "Adreça3", "e–mail4@domini.com", "+34931112233")
      };
      clients[2].addComanda(new Encarrec("Impressora",1));
      clients[2].addComanda(new Encarrec("Toner Impressora",4));
      clients[2].addComanda(new Encarrec("Paquest A4", 20));
      for(int i = 0; i < clients.length; i++) {
        db.store(clients[i]);
      }
    } finally {
      db.close();
    }
  }
}
```

En aquest codi hi ha un aspecte molt important en què val la pena fixar-se en detall. Si l'examineu, veureu que, tot i que els objectes que cal emmagatzemar són tant els clients com els encàrrecs, enlloc es fa cap `store` per als encàrrecs. Només es fa per als clients. Això es deu al fet que, en les BDOO, en fer persistent un objecte, aquesta persistència es propaga a tots els objectes enllaçats

també, de manera transitiva, fins que tot el mapa d'objectes, el *graf* d'enllaços que parteix de l'objecte emmagatzemat es troba al complet a la BDOO. Això es fa automàticament sense necessitat que ho faci el programador. Aquest comportament, esquematitzat a la figura 3.1, també succeeix quan es recuperen les dades amb alguna cerca, com veureu properament.

FIGURA 3.1. Tractament dels mapes d'objectes sobre una BDOO



Una pregunta que pot sorgir és: què passa si, *a posteriori*, s'emmagatzema un objecte que, a causa d'aquest comportament, ja existeix a la BDOO? Per exemple, si es fa un `store` d'un dels tres encàrrecs, quan aquest de fet ja està a la BDOO, ja que s'ha desat automàticament en emmagatzemar el client 3. La resposta és que no passa res. La BDOO ja detecta que es tracta del mateix objecte i, per tant, no es generen dues còpies.

Això és possible ja que, recordeu que una de les bases de la OO és "Tot és un objecte, amb una identitat pròpia". O sigui, tot objecte s'identifica amb una única referència. Aquesta pot estar replicada en diferents variables, però totes apunten a un únic objecte a memòria. Això també es compleix dins la BDOO, i, per tant, aquesta és capaç d'identificar diferents operacions amb un mateix objecte.

Ara bé, aquest comportament té unes altres implicacions que cal tenir també ben presents. Suposeu que aquest mateix programa l'executeu 3 vegades consecutives. Quants objectes client hi haurà emmagatzemats a la BDOO després de la darrera execució? La resposta és que n'hi haurà 12, ja que els objectes de cada execució són independents entre si. Tot i que el contingut dels objectes en successives execucions és exactament el mateix, l'objecte en si és diferent, tenen diferents referències, i, per tant, es considera un nou element a la BDOO. En conseqüència, cal anar amb molt de compte en emmagatzemar nous objectes entre execucions diferents del programa, ja que això sempre implicarà la creació d'un nou element a la BDOO.

3.2.3 Cerca d'objectes

Els mecanismes de lectura d'una BD solen ser els més importants, ja que normalment són els que s'usen més sovint. La llibreria *db4o* ofereix dos sistemes diferents per dur a terme aquest procés, diferenciats únicament per com es discrimina quins objectes cal retornar de la BDOO. En qualsevol dels dos casos, el que es retorna és un conjunt d'objectes, exactament tal com els heu definit a les vostres classes, empaquetat dins un contenidor específic de *db4o* anomenat `ObjectSet<T>`. Aquesta és una classe genèrica, de manera que, en declarar-ne una variable, cal establir sempre el tipus d'objectes que contindrà (com passa amb altres contenidors del Java:`List`, `Set`...). Per exemple, si es volen consultar clients, caldrà usar la definició `ObjectSet<Client>`.

Una de les característiques més interessants de les cerques dins una BDOO és la recuperació d'objectes enllaçats entre si, de manera que si, en recuperar un objecte, aquest contenia a la vegada referències a altres objectes, aquests objectes també són recuperats.

I així fins a un cert nivell de profunditat, que per defecte val 5, però que es pot modificar utilitzant el codi:

```
1 EmbeddedConfiguration conf = Db4oEmbedded.newConfiguration();
2 conf.common().activationDepth(novaProfunditat);
```

L'exemple més senzill d'aquest comportament són els atributs de tipus `String`, que també són pròpiament objectes, i són restaurats junt amb l'objecte original. Però aquest comportament també es compleix per a objectes de qualsevol altra classe, ja sigui del Java o creada per vosaltres.

Cerques per exemple

Les cerques per exemple (*Query-By-Example*) són les més senzilles. Es basen a crear una instància del tipus d'objecte a cercar, i només assignar valors als atributs sobre els quals es vol cercar una coincidència exacta. La resta, es posen a `null` (en el cas de valors numèrics, a 0). Llavors, s'invoca el mètode `queryByExample`, usant com a paràmetre aquesta instància.

Per exemple, per cercar tots els clients que tenen com a adreça la cadena de text "Adreça3" es faria d'acord al codi que hi ha a continuació. En executar-se el codi, atesos els quatre clients d'exemple existents a la BDOO, dins l'`ObjectSet` hi haurà 2 objectes `Client`: el que té com a `Client3` i el que té `Client4`.

```
1 ObjectContainer db = Db4oEmbedded.openFile("BDOOClients.db4o");
2   ...
3   Client ex = new Client(null, "Adreça3", null, null);
4   ObjectSet<Client> result = db.queryByExample(ex);
5   ...
```

L'`ObjectSet` pot ser recorregut seqüencialment mitjançant els mètodes `hasNext()`, que indica si encara hi ha elements per recórrer, i `next()`, que llegeix un element i avança una posició. En aquest sentit, es comporta exactament igual que un `Iterator` de les llibreries estàndard del Java. Per exemple, per tal de mostrar per pantalla tots els elements obtinguts per la consulta, es podria fer el següent:

```
1 ObjectSet<Client> result = ...
2 while (result.hasNext()) {
3   Client cli = result.next();
4   System.out.println(cli);
5 }
```

Atès que `ObjectSet` és una classe genèrica, el mètode `next()` retorna directament una instància del tipus indicat, i no cal fer cap "cast". Ara bé, cal anar amb compte a definir sempre el tipus dels elements de l'`ObjectSet`, en declarar-ne la variable, de la mateixa classe que la instància usada en invocar `queryByExample`. En cas contrari, es produirà un error per manca de concordança de tipus.

Si bé aquesta mena de cerques són molt senzilles de fer, no permeten res més que la comparació directa d'atributs. Tanmateix, tampoc no permeten fer cerques sobre valors que siguin `null` o 0, ja que són les condicions per ignorar-los com a criteri de cerca. També tenen la restricció que no poden usar-se a partir d'objectes que no permeten inicialitzar atributs a valors `null` o 0.

Cerques natives

Normalment, les cerques que es voldran fer dins una BD van més enllà de les simples concordances directes entre valors d'atributs, i es desitjarà poder avaluar tota mena de condicions, a gust del desenvolupador, tal com permet SQL (o més). Hi ha dos mecanismes per dur a terme cerques d'acord a criteris complexos dins *db4o*, però la més potent, i, a la vegada, la més senzilla, són les cerques natives (*Native Queries*). Es basen en l'execució d'un codi Java per avaluar si un objecte dins la BDOO compleix la condició de cerca o no. En basar-se només en codi Java, la seva versatilitat és la mateixa que en qualsevol programa possible, o sigui, molt gran.

Per executar una cerca nativa s'usa el mètode `query`, que necessita com a paràmetre una implementació de la classe `Predicate<T>` (pertanyent al *package* `com.db4o.query`).

`Predicate<T>` és una classe genèrica abstracta, per la qual cal indicar quina mena d'objectes és capaç de processar en emprar-la. L'únic mètode abstracte que cal implementar és `match`, que s'ha de fer que s'avaluï a `true` si es considera que l'objecte passat com a paràmetre compleix el criteri de cerca, o `false` en cas contrari. Aquest mètode s'executarà passant com a paràmetre tots els objectes de la BDOO que es corresponguin al tipus escollit, un per un. Ara bé, el codi d'aquest mètode l'heu d'implementar vosaltres, ja que és el que ha de prendre la decisió de si un objecte compleix o no el criteri de cerca, i, per tant, el podeu fer al vostre gust. Per a cada cerca diferent que es vol fer al programa, caldrà crear una implementació diferent d'aquesta classe.

Tot seguit, es mostra un exemple d'implementació de la classe `Predicate<T>`, de manera que avalua si, donat un client, aquest té com a adreça "Adreça3", ignorant majúscules i minúscules (cosa que no es pot fer amb una cerca per exemple, ja que compara textos estrictament). Normalment, per implementar aquesta classe s'usa una classe anònima, de manera que tan bon punt es declara una variable d'aquest tipus, ja s'indica el seu codi immediatament, en lloc de fer-ho en un fitxer a part.

```
ObjectContainer db = Db4oEmbedded.openFile("BDOOClients.db4o");

//Declaració de la implementació de Predicate<T> com a classe anònima
Predicate p = new Predicate<Client>() {
  @Override
  public boolean match(Client c) {
    //Codi pel criteri de cerca
    return "Adreça3".equalsIgnoreCase(c.getAPostal());
  }
};
//Fi de la declaració

ObjectSet<Client> result = db.query(p);
...
```

L'avantatge d'usar classes anònimes és que permeten incloure, dins el seu propi codi, atributs declarats dins la mateixa classe que les conté. Això atorga gran flexibilitat si es volen fer cerques en base a variables, i no a valors constants. Això permet crear cerques parametritzades mitjançant objectes `Predicate`. Per exemple, suposem que es volen cercar els clients que han superat cert valor en el nombre de comandes, però aquest valor depèn d'una variable dins el codi del vostre programa, ja que es demana pel teclat i per tant pot ser diferent en diferents execucions. El codi següent fa tot just això.

```
import com.db4o.*;
import com.db4o.query.Predicate;
import java.util.Scanner;

public class CercaParametritzada  {
  private int valor = 0;

  public void cercaClients() throws Exception {
    ObjectContainer db = Db4oEmbedded.openFile("BDOOClients.db4o");
    try {
      Predicate p = new Predicate<Client>() {
       @Override
       public boolean match(Client c) {
         return valor <= c.getNreComandes();
       }
      };
      ObjectSet<Client> result = db.query(p);
      while (result.hasNext()) {
        Client cli = result.next();
        System.out.println(cli);
      }
    } finally {
      db.close();
    }
  }

  public void setValor(int v) {
    valor = v;
  }

  public static void main(String[] args) throws Exception {
      CercaParametritzada cp = new CercaParametritzada();
```

```
33        Scanner in = new Scanner(System.in);
34        System.out.print("Quin és el valor mínim a cercar? ");
35        cp.setValor(in.nextInt());
36        cp.cercaClients();
37    }
38  }
```

En fer cerques, recordeu que a la BDOO no només hi ha aquells objectes dels quals s'ha fet una crida `store` explícita, sinó que també es troben disponibles els objectes emmagatzemats implícitament a causa d'enllaços amb altres objectes. Per tant, a la BDOO, també es poden fer cerques sobre encàrrecs. El següent codi permet fer una cerca parametritzada d'encàrrecs dins el sistema, en base a un valor mínim en la seva quantitat.

```
1   import com.db4o.*;
2   import com.db4o.query.Predicate;
3   import java.util.Scanner;
4
5   public class CercaParametritzada  {
6
7     private int valor = 0;
8
9     public void cercaEncarrecs() {
10      ObjectContainer db = Db4oEmbedded.openFile("BDOOClients.db4o");
11      try {
12        Predicate p = new Predicate<Encarrec>() {
13         @Override
14         public boolean match(Encarrec c) {
15           return valor <= c.getQuantitat();
16         }
17        };
18        ObjectSet<Encarrec> result = db.query(p);
19        while (result.hasNext()) {
20          Encarrec e = result.next();
21          System.out.println(e);
22        }
23      } finally {
24        db.close();
25      }
26    }
27
28    public void setValor(int v) {
29      valor = v;
30    }
31
32  public static void main(String[] args) throws Exception {
33        CercaParametritzada cp = new CercaParametritzada();
34        Scanner in = new Scanner(System.in);
35        System.out.print("Quin és el valor mínim a cercar? ");
36        cp.setValor(in.nextInt());
37        cp.cercaEncarrecs();
38      }
39  }
```

En aquests exemples, el codi per establir si cada objecte compleix o no el criteri de la cerca és relativament simple, d'una sola línia, però el codi del mètode `match` pot ser tan complex com es desitgi i basat en qualsevol informació disponible dins el programa. Ara bé, malauradament, això vol dir que no es pot disposar de funcions executables directament a la BD (com passava amb MAX, AVG... a SQL).

3.2.4 Actualitzacions d'objectes

Normalment, l'actualització d'objectes té sentit quan, primer de tot, s'afegeix un element a la BDOO, i, en posteriors execucions del programa, el contingut dels objectes emmagatzemats veuen modificats els seus valors al llarg del seu cicle de vida (un client canvia la seva adreça electrònica, o va afegint encàrrecs). És el que dóna sentit a la persistència dels objectes en una BD, en definitiva.

L'actualització d'objectes amb *db4o* es porta a terme usant la crida `store`, tal com s'ha usat per emmagatzemar un nou objecte, només que, en aquest cas, en lloc de ser un objecte nou, és un que ja existia prèviament a la BD. Per fer això, primer cal carregar a memòria l'objecte des de la BD, usant una cerca, i un cop es disposa de la seva referència, ja s'hi pot accedir per fer canvis usant els mètodes que proporciona la seva classe, tal com es faria normalment. Aquests canvis no es propagaran a la BD fins tornar a executar `store`.

Podeu comprovar que s'ha modificat executant els exemples anteriors de cerca amb un valor que llisti tots els clients (com el 0).

Per exemple, el següent codi permet canviar l'adreça electrònica d'un client, donat el seu nom (suposarem que el nom ha de ser únic perquè funcioni). Per millorar la llegibilitat, s'han omès algunes comprovacions d'errors (si no s'escriu res amb el teclat, per exemple).

```
1  import com.db4o.*;
2  import java.util.Scanner;
3
4  public class ModificaAElectronica {
5
6    public static void main(String[] args) throws Exception {
7      ObjectContainer db = Db4oEmbedded.openFile("BDOOClients.db4o");
8      Scanner in = new Scanner(System.in);
9      System.out.print("Quin és nom del client? ");
10     String nom = in.nextLine();
11
12     //Cercar clients a la BDOO i obtenir-los a memòria com a objectes del
           programa
13     Client qbe = new Client(nom, null, null, null);
14     ObjectSet<Client> clients = db.queryByExample(qbe);
15
16     if (clients.size() != 1) {
17       System.out.println("No es pot modificar aquest nom.");
18     } else {
19       System.out.print("Quina és la nova adreça? ");
20       String ad = in.nextLine();
21       Client c = clients.next();
22       c.setAElectronica(ad);
23       db.store(c);
24     }
25     db.close();
26   }
27
28 }
```

L'única excepció a aquest comportament és si l'objecte enllaçat és una cadena de text (com s'ha vist, precisament, a l'exemple anterior).

Ara bé, en el cas d'objectes que contenen enllaços a altres objectes (que, a la vegada, poden tenir enllaços a altres objectes, i així fins a molts nivells de profunditat), el comportament de *db4o* no és aquest. Per poder garantir un rendiment òptim, és obligat, en el cas d'accés a fitxers en local, canviar una mica la declaració de l'obertura de la BD, indicant que ha d'estar configurada per acceptar

actualitzacions en cascada. Això es fa usant un constructor diferent i una inicialització prèvia d'un objecte `EmbeddedConfiguration`. En aquesta inicialització, cal llistar totes les classes on es vol que la BDOO controli actualitzacions en cascada. O sigui, que es vol que, si es fa un `store` sobre un objecte d'aquest tipus, també es comprovi si cal actualitzar tots els seu *graf* d'objectes enllaçats.

El codi d'inicialització és el que hi ha a continuació. On diu "NomClasse1", "NomClass2"... caldria posar el nom de la classe a controlar les actualitzacions en cascada. Hi haurà tants registres de classes com classes es volen controlar. Per al cas de qualsevol classe que no s'enumeri explícitament a la inicialització, amb una línea corresponent, en fer un `store`, només se n'actualitzaran els atributs que siguin tipus primitius o cadenes de text, però no altres objectes.

```
EmbeddedConfiguration config = Db4oEmbedded.newConfiguration();

  //Configurar totes les classes on cal propagar canvis
  config.common().objectClass(NomClasse1.class).cascadeOnUpdate(true);
  config.common().objectClass(NomClasse2.class).cascadeOnUpdate(true);
  //etc.

  ObjectContainer db = Db4oEmbedded.openFile(config, "BDOOClients.db4o");
```

Per exemple, el codi següent permet afegir un encàrrec a un client, donat el seu nom (suposant que aquest és únic). Com que els encàrrecs estan enllaçats dins una llista, cal forçar les actualitzacions en cascada per a aquesta classe.

```
import com.db4o.*;
import com.db4o.config.EmbeddedConfiguration;
import java.util.Scanner;

public class AfegirEncarrec {
  public static void main(String[] args) throws Exception {
    EmbeddedConfiguration config = Db4oEmbedded.newConfiguration();

    config.common().objectClass(Client.class).cascadeOnUpdate(true);

    ObjectContainer db = Db4oEmbedded.openFile(config, "BDOOClients.db4o");

    Scanner in = new Scanner(System.in);
    System.out.print("Quin és nom del client? ");
    String nom = in.nextLine();

    //Cercar clients a la BDOO i obtenir-los a memòria com a objectes del
        programa
    //S'usa una cerca per exemple
    Client qbe = new Client(nom, null, null, null);
    ObjectSet<Client> clients = db.queryByExample(qbe);

    if (clients.size() != 1) {
      System.out.println("No es pot modificar aquest nom.");
    } else {
      System.out.print("Quin és nom del producte? ");
      String prod = in.nextLine();
      System.out.print("Quants en vols encarregar? ");
      String txtQuan = in.nextLine();
      int quant = Integer.parseInt(txtQuan);

      Encarrec ne = new Encarrec(prod, quant);
      Client c = clients.next();
      c.addComanda(ne);
      db.store(c);
    }
    db.close();
```

```
  }
}
```

3.2.5 Esborrat d'objectes

Per esborrar un objecte s'aplica la idea general de les actualitzacions, però en lloc del mètode `store`, cal usar el mètode `delete`. Primer cal recuperar l'objecte de la BD, carregar-lo a memòria, i després esborrar-lo usant la seva referència. Per exemple, per esborrar un client, d'entrada, es podria fer així:

```
import com.db4o.*;
import java.util.Scanner;

public class EsborraClient {
  public static void main(String[] args) throws Exception {
    ObjectContainer db = Db4oEmbedded.openFile("BDOOClients.db4o");
    Scanner in = new Scanner(System.in);
    System.out.print("Quin és nom del client? ");
    String nom = in.nextLine();
    //Cercar clients a la BDOO i obtenir-los a memòria com objectes del
        programa
    Client qbe = new Client(nom, null, null, null);
    ObjectSet<Client> clients = db.queryByExample(qbe);
    if (clients.size() != 1) {
      System.out.println("No es pot modificar aquest nom.");
    } else {
      Client c = clients.next();
      db.delete(c);
    }
    db.close();
  }
}
```

Si mostreu els clients que hi ha a la BDOO, veureu que ja no existeix el client que heu escrit en executar-lo. Ara bé, aquest programa no és del tot correcte. Si esborreu el client 3, executeu el programa que fa cerca de comandes i cerqueu totes les comandes amb una quantitat superior a 0, observareu que les comandes del client 3 encara estan a la BD. La crida a `delete` esborra, estrictament, l'objecte associat a aquest client, però absolutament res més.

Com passava amb les actualitzacions d'objectes que enllacen altres objectes, aquest cas s'ha de tractar d'una manera una mica especial. Malauradament, no hi ha cap paràmetre de configuració que resolgui el problema. Per tal d'eliminar objectes, el programador ha de fer un codi que, manualment i un per un, vagi recorrent el mapa d'objectes i eliminant els que correspongui. Per als clients, el programa correcte hauria de ser el següent:

```
import com.db4o.*;
import java.util.*;

public class EsborraClient {

  public static void main(String[] args) throws Exception {
    ObjectContainer db = Db4oEmbedded.openFile("BDOOClients.db4o");
    Scanner in = new Scanner(System.in);
    System.out.print("Quin és nom del client? ");
```

```
10      String nom = in.nextLine();
11
12      //Cercar clients a la BDOO i obtenir-los a memòria com objectes del
            programa
13      Client qbe = new Client(nom, null, null, null);
14      ObjectSet<Client> clients = db.queryByExample(qbe);
15
16      if (clients.size() != 1) {
17        System.out.println("No es pot modificar aquest nom.");
18      } else {
19        Client c = clients.next();
20        List<Encarrec> li = c.getComandes();
21        Iterator<Encarrec> it = li.iterator();
22        //anem esborrant tots els encàrrecs, un per un
23        while (it.hasNext()) {
24          Encarrec e = it.next();
25          db.delete(e);
26        }
27        //Ja es pot esborrar el client
28        db.delete(c);
29      }
30      db.close();
31    }
32  }
```

En aquest cas concret, la tasca no és gaire complicada, ja que només hi ha un nivell d'objectes enllaçats, i donat un client, les seves comandes només les gestiona ell i cap altre objecte. Ara bé, cal ser conscients que gestionar l'esborrat correcte d'objectes enllaçats dins la BDOO es pot arribar a complicar força en casos complexos, ja que sempre cal garantir la consistència de tot el mapa d'objectes emmagatzemat. Si un objecte és referenciat per més d'un altre objecte, cal anar amb molt de compte de no esborrar-lo, perquè es podrien deixar referències a `null` sense voler, cosa que comportaria errors d'execució en el futur. Per exemple, si els encàrrecs es poguessin compartir entre més d'un client, en esborrar un client no es podrien esborrar alegrement tots els seus encàrrecs. Caldrà comprovar que cada encàrrec només està assignat a un únic client i, si cal, l'esborrarem, però, en cas contrari, no.

3.3 JDO (Java Data Objects)

Normalment, el motiu principal per treballar amb una BDDO és poder generar codi que s'integri de manera natural en un programa orientat a objectes. O sigui, poder desar i cercar directament objectes dins la base de dades, de manera que si estan enllaçats a altres objectes mitjançant referències, també es recuperin automàticament. A més a més, els mètodes d'accés a la BD ja retornen directament els conjunts d'objectes que es vol cercar.

En contraposició, l'ús d'una BDR implica haver de fer traduccions d'objectes a taules, i les cerques retornen files de valors de taules, que cal recuperar, i a partir d'aquests cal instanciar les classes desitjades. Per desar un objecte, cal fer exactament el mateix, però a la inversa: extreure'n els atributs i convertir-los a valors dins una taula. En cas de voler desar objectes enllaçats entre si, la tasca es pot arribar a complicar força. A part, cal fer conversions de tipus Java a SQL i

viceversa. En general, la seva integració dins el codi Java no és gens natural, ja que s'està usant un sistema basat en taules per desar elements que, a memòria, no són pas a taules.

Per tant, des del punt de vista de simplicitat del codi, l'ús d'una BDOO dins un programa orientat a objectes ofereix enormes avantatges. Malauradament, el seu grau de maduresa i estandardització no arriba ni de bon tros al que actualment tenen les BDR, per la qual cosa no és tan senzill decidir quin tipus usar. Ara bé, el Java disposa d'una especificació pròpia per definir la persistència d'objectes usant una estratègia semblant a l'accés a una BDOO, però sense lligar-se a un tipus concret de mecanisme d'emmagatzematge.

Java Data Objects (JDO) API és una interfície estàndard basada en l'abstracció del model de persistència de Java. Les aplicacions escrites amb l'API de JDO són independents del sistema d'emmagatzematge subjacent. Diferents implementacions poden donar suport a diferents tipus de bases de dades, incloent bases de dades relacionals i d'objecte, XML, arxius...

Una BDOO que es basa en JDO, per a l'accés a les seves dades, és JDOInstruments.

Aquesta especificació només serveix per al llenguatge Java i no és portable a cap altre llenguatge, en contrast amb l'ODL i OQL, que intenten ser llenguatges genèrics independents del llenguatge (com l'SQL). De fet, l'objectiu d'aquesta especificació és haver d'obviar la necessitat d'aprendre cap llenguatge extra que no sigui el mateix Java i res més. En aquesta secció es presenta només una breu introducció dels aspectes generals del seu funcionament.

Les classes de JDO pertanyen al paquet `javax.jdo`. Aquest no pertany a la distribució estàndard del Java.

JDO defineix tres tipus de classes:

- **Habilitades per a persistència**. Representen les classes els objectes de les quals poden passar a un estat persistent. Al llarg de l'execució de l'aplicació poden passar de la memòria a ser emmagatzemades en la BD, i viceversa.
- **Conscients de persistència**. Són les classes que manipulen el tipus anterior. Concretament, la classe `JDOHelper` proporciona diferents mètodes per descobrir si un objecte concret es troba en un estat persistent o no.
- **Normals**. Els seus objectes no poden passar a un estat persistent, només existiran en la memòria de l'aplicació.

Objectes transitoris

S'anomena objectes transitoris els que només són en la memòria, però no en la BD, i objectes buits (*hollow*) els que només estan en la BD, sense representació en la memòria.

Els objectes de classes habilitades per a la persistència poden passar per diferents estats dins el seu cicle de vida, depenent de diversos factors: si només es troben en la memòria o també estan representats en la BD, si la seva representació actual en la memòria difereix de l'escrita en la BD, etc. La transició entre estats és controlada per la classe `PersistenceManager`, que serveix d'*interface* primària entre l'aplicació i la BD. Aquesta classe és, fins a cert punt, l'equivalent a la connexió JDBC en el cas d'accés a BDR.

Totes les accions cap a la BD vindran determinades a partir de crides a mètodes definits en aquesta classe. Alguns dels seus mètodes més significatius són:

- `void makePersistent(Object o)`. Fa un objecte persistent a la BD de manera que quan es manipulin les seves dades, quedaran desades.
- `void makeTransient(Object o)`. Fa que un objecte deixi de ser persistent en la BD i passi a ser transitori.
- `void deletePersistent(Object o)`. Esborra un objecte persistent de la BD.

3.3.1 Instanciació d'un objecte PersistenceManager

Per instanciar un objecte `PersistenceManager` no es pot usar directament una sentència `new`, cal usar la classe `PersistenceManagerFactory`:

```
1 Properties props = new Properties();
2 props.put(...);
3 PersistenceManagerFactory pmf = JDOHelper.getPersistenceManagerFactory(props);
4 PersistenceManager pm = pmf.getPersistenceManager();
```

Un cop disposem de l'objecte `PersistenceManager`, ja es pot començar a operar amb la BDOO.

3.3.2 Interacció amb la BD

Per interactuar amb la BD, el JDO usa transaccions, per modificar l'estat d'un objecte, i consultes, per accedir a les dades persistents. Aquesta filosofia no és gaire diferent de la de JDBC. Les transaccions estan representades per la classe `Transaction`, mentre que les consultes per la classe `Query`. Els objectes d'ambdós tipus s'obtenen a partir de crides al `PersistenceManager`:

- **Transaction currentTransaction()**. Obté una transacció per modificar l'estat d'objectes dins l'aplicació.
- **Query newQuery(java.lang.Class cls)**. Crea una consulta, per ser executada posteriorment per accedir als objectes persistents dins la BDOO. El paràmetre cls indica la classe esperada dels objectes que es vol consultar. Aquest mètode es troba sobrecarregat per poder proporcionar diferents opcions.

L'aspecte més important en el procés d'emmagatzemament i recuperació de dades en la BD és que tot codi Java generat segueix exactament la mateixa filosofia orientada a objectes com si sempre s'operés sobre instàncies a memòria (només s'opera amb referències). El mecanisme de persistència és transparent.

El mètode `rollback()` permet fer enrere una transacció en cas d'error durant el procés.

Tot seguit es mostra com un objecte passa de ser transitori a persistent. El resultat final és que aquest queda emmagatzemat en la BDOO.

```
Client cli = new client("Client1", "Adreça1",...);
Transaction tr;
try {
  tr = pm.currentTransaction();
  tr.begin();
  pm.makePersistent(cli);
  tr.commit();
  } catch (Exception e) {
    if(tx.isActive()) {
      tx.rollback();
   }
  }
```

El mètode `closeAll()` finalitza la consulta, alliberant els objectes resultants.

Per executar una consulta cal cridar el mètode `execute()` definit en la classe `Query`. Aquest retorna una col·lecció d'objectes resultants. Per exemple, per recuperar objectes `Client` emmagatzemats es faria:

```
try {
  Query query = pm.getQuery(Client.class);
  query.setFilter("(nom == param)");
  query.declareParameters("String param");
  Collection c = (Collection)query.execute("Client1");
  Iterator it = c.iterator();
  while(it.hasNext()) {
    Client cli = (Client)it.next();
    System.out.println("S'ha trobat el client:" + cli);
  }                              closeAll()
  query.closeAll();
} catch (Exception e) {
  ...
}
```

Els mètodes `setFilter` i `declareParameters` permeten establir condicions de cerca. Amb el primer s'estableix la condició d'acord amb certs paràmetres d'entrada (param) que s'usaran en fer la consulta i es compararan amb els camps dels objectes (nom). Amb el segon establim els tipus dels paràmetres d'entrada (String param). En fer execute, s'indiquen els valors dels paràmetres (Client1).