

# Llenguatge Java. Introducció

Isidre Guixà Miranda

Programació orientada a objectes



# Índex

<b>Introducció .....</b>	<b>5</b>
<b>Objectius .....</b>	<b>7</b>
<b>1. Fonaments de programació estructurada en Java .....</b>	<b>9</b>
1.1. Història .....	9
1.2. Components principals de la tecnologia Java .....	13
1.2.1. La màquina virtual Java (JVM) .....	13
1.2.2. El llenguatge Java .....	16
1.3. Eines bàsiques per desenvolupar programes Java .....	26
1.4. El primer programa Java .....	27
1.4.1. El codi font .....	27
1.4.2. La fase de compilació .....	29
1.4.3. La fase d'execució .....	30
1.5. Components bàsics del llenguatge .....	31
1.6. Sentències de control de flux .....	45
1.6.1. Sentències condicionals .....	45
1.6.2. Sentències repetitives .....	47
1.6.3. Trencament d'estructures repetitives .....	50
1.7. Taules .....	53
1.8. Cadenes .....	58
<b>2. Programació orientada a objectes en Java .....</b>	<b>61</b>
2.1. Classes i objectes .....	61
2.2. Gestió d'objectes d'una classe .....	64
2.2.1. Com es creen els objectes? .....	65
2.2.2. Com es fa referència als objectes? .....	66
2.2.3. Com s'inicialitzen els objectes? .....	67
2.2.4. Com es manipulen els objectes? .....	68
2.2.5. Com es destrueixen els objectes? .....	69
2.3. Disseny de classes .....	71
2.3.1. Definició de classes: dades, iniciadors i mètodes .....	72
2.3.2. Encapsulació. Ocultació de dades .....	76
2.3.3. Sobrecàrrega de mètodes. Polimorfisme? .....	80
2.3.4. Construcció i inicialització d'objectes .....	81
2.3.5. La paraula reservada this .....	84
2.3.6. La paraula reservada static .....	86
2.3.7. Paquets de classes .....	89
2.3.8. Arxius jar .....	94
2.4. Herència .....	97
2.4.1. Definició de classes derivades .....	100

2.4.2. Sobreescritura de membres. Polimorfisme? .....	101
2.4.3. Construcció i inicialització d'objectes derivats .....	113
2.4.4. Classes abstractes .....	116
2.5. Interfícies .....	118
2.6. Classes internes .....	123
2.6.1. Classes i interfícies internes static .....	124
2.6.2. Classes internes membre .....	126
2.6.3. Classes internes locals .....	128
2.6.4. Classes internes anònimes .....	129
2.7. Classes genèriques .....	131
<b>3. Classes fonamentals .....</b>	<b>138</b>
3.1. L'API de Java. <i>Frameworks</i> .....	138
3.2. Representació i manipulació de col·leccions .....	141
3.2.1. Interfícies .....	141
3.2.2. Implementacions .....	151
3.2.3. Algorismes .....	157
3.3. Excepcions .....	158
3.3.1. Captura .....	161
3.3.2. Gestió: captura o delegació .....	168
3.3.3. Llançament .....	173
3.3.4. Creació .....	175
3.3.5. Efecte de l'herència .....	176
3.4. Obtenció de dades per l'entrada estàndard .....	177
3.5. API de reflexió .....	178

## Introducció

El desenvolupament de programes informàtics s'ha de fer de la mà d'un llenguatge que suporti l'orientació a objectes. En el moment actual de la informàtica hi ha molts llenguatges que suporten l'orientació a objectes i ens havíem de decidir per un: l'elecció ha estat fàcil en decantar-nos pel llenguatge que potser té més adeptes en el moment actual, el llenguatge Java.

El llenguatge Java facilita un munt de possibilitats i es pot utilitzar en qualsevol àmbit (gestió, indústria...); per tant, és immens, fet que fa impossible abordar-lo en la seva totalitat. Aquesta unitat didàctica vol ser una introducció al llenguatge que ens permeti conèixer-ne els fonaments i, a partir d'aquí, avançar en l'àmbit que més interressi.

En el nucli d'activitat "Fonaments de programació estructurada en Java" s'introdueix el llenguatge Java amb les capacitats que l'han dut a ser tan estimat per la comunitat de programadors i les característiques de programació estructurada i modular que proporciona (tipus de dades, definició de variables i constants, i estructures de control que proporciona). En aquest nucli d'activitat es fa una ullada força detallada a les característiques de programació estructurada i modular que proporciona el llenguatge Java, fet que no s'ha de confondre amb l'aprenentatge de la programació estructurada i modular. A més, a fi de realçar els punts que el fan fort davant altres llenguatges, quan es possible, es presenten comparacions amb els llenguatges C i C++.

En el nucli d'activitat "Programació orientada a objectes en Java" comença el periple per la programació orientada a objectes. En aquest nucli s'introdueixen els mecanismes que el llenguatge Java proporciona per implementar l'orientació d'objectes i, en especial, aconseguir les tres fites principals d'aquesta metodologia: encapsulació, herència i polimorfisme.

El llenguatge Java és immens i, per tant, incorpora un munt de classes, agrupades en paquets, que els programadors poden utilitzar com a classes bàsiques o de suport en les aplicacions. En el nucli d'activitat "Classes fonamentals en Java" s'introdueixen alguns dels paquets de classes més útils per a la comunitat de programadors, que podem resumir en el paquet de classes que permet gestionar conjunts de dades (*framework* de col·leccions), el paquet de classes que permet gestionar els errors (excepcions) que es poden produir durant l'execució dels programes i que faciliten molt la programació en comparació amb les tècniques que proporcionava la programació estructurada i modular, les classes d'entrada i sortida bàsi-

ques pels canals d'entrada i sortida estàndard (teclat i pantalla) i un paquet de classes que permet, en temps d'execució, obtenir informació important sobre les classes carregades o proporcionades pel llenguatge (reflexió).

Així, doncs, es prou important i bàsic aconseguir una base sòlida en el llenguatge Java i, per aconseguir-la és imprescindible que aneu reproduint en el vostre ordinador tots els exemples incorporats en el text com també les activitats i els exercicis d'autoavaluació.

## Objectius

En acabar la unitat didàctica heu de ser capaços del següent:

1. Desenvolupar codis font de programes amb objectes, en llenguatge de programació orientats, segons la sintaxi i a partir dels algorismes dissenyats.
2. Obtenir els codis objecte i executable a partir dels codis font i objecte, respectivament.
3. Definir els objectes necessaris per a la resolució de problemes amb un llenguatge de programació orientat a objectes.
4. Depurar els mòduls de programació desenvolupats, segons els criteris d'eficàcia i eficiència.
5. Controlar la qualitat del programa desenvolupat, a partir de les especificacions establertes en el disseny i de les prestacions que espera l'usuari.
6. Verificar el funcionament dels mòduls de l'aplicació, les integracions i els enllaços.
7. Elaborar la guia d'usuari i la documentació completa de les fases de disseny, codificació, traducció, enllaç, depuració i verificació de programes, de manera estructurada i clara.
8. Determinar les causes i les formes de resolució de les incidències que han sorgit en les fases de codificació, traducció, enllaç i depuració de programes, segons el manual de referència.
9. Comprovar l'eficàcia i l'eficiència de les prestacions i dels rendiments del programa elaborat.





## 1. Fonaments de programació estructurada en Java

El llenguatge Java és un llenguatge de programació orientat a objectes desenvolupat per l'empresa nord-americana Sun Microsystems al principi dels anys noranta i va aparèixer per donar resposta a l'objectiu de disposar d'un llenguatge que permetés escriure el codi una única vegada i executar-lo en qualsevol lloc (*write once, run anywhere* en paraules del seu creador, James Gosling).

El seu desenvolupament es va veure clarament influenciat pels llenguatges existents: C++, Smalltalk, Eiffel i Objective-C. D'ells pren les millors característiques i intenta minimitzar-ne els punts crítics. Així, per exemple, del llenguatge C++ elimina la manipulació directa de les direccions de memòria per punters, fet que acostuma a provocar molts errors.

El llenguatge Java és un llenguatge orientat a objectes i, per tant, proporciona mecanismes per implementar les característiques principals de la programació orientada a objectes (encapsulació, herència i polimorfisme), però no deixa de ser un llenguatge de programació que incorpora els fonaments de la programació estructurada i, abans de conèixer els mecanismes que permeten implementar l'encapsulació, l'herència i el polimorfisme, ens cal dominar els fonaments que incorpora de programació estructurada: tipus de dades (simples i compostes) i estructures de control (condicionals i repetitives).

### 1.1. Història

El naixement de la tecnologia Java es pot considerar que té lloc el 15 de gener de l'any 1991 en la reunió inicial del grup de treball Green Project promogut per l'empresa Sun Microsystems. L'objectiu del grup consistia a anticipar-se i preparar-se per al futur de la informàtica i la conclusió inicial va ser que com a mínim es produiria una tendència important cap la convergència dels dispositius digitals i l'electrònica de consum. Així, doncs, era molt convenient disposar d'algun entorn de programació únic que pogués ser utilitzat per tots els dispositius d'electrònica de consum.

Amb l'objectiu marcat inicien la feina l'1 de febrer de 1991, en una petita oficina, de manera secreta i amb total autonomia respecte a la línia directiva de Sun Microsystems. L'equip de treball el formen tretze persones i el seu desenvolupament l'allarga divuit mesos. Es reparteixen les feines a fer i James Gosling és l'encarregat de definir el llenguatge de programació, motiu pel qual se'l considera el creador de Java.

#### Java és un llenguatge multiplataforma

Un dels grans inconvenients que tenen la majoria de llenguatges de programació és que no són multiplataforma, és a dir, les fases de compilació i enllaç s'efectuen en un sistema operatiu i el resultat només és portable sobre màquines en què corri el mateix sistema operatiu.

El llenguatge Java no té aquest inconvenient i aconsegueix ser un llenguatge multiplataforma gràcies a una màquina virtual instal·lada en cada sistema operatiu que s'encarrega d'executar el resultat de la compilació del codi font Java independentment de la màquina en què s'hagi efectuat la compilació.

#### Tecnologia Java

El llenguatge Java no té cap sentit per si sol i sempre va vinculat a la màquina virtual Java que és l'encarregada de dur a terme l'execució dels compilats obtinguts a partir del codi font Java.

Així, doncs, quan es parla de la història o evolució del llenguatge Java, seria millor emprar el terme *tecnologia Java*.

En un principi l'equip va considerar C++ com a llenguatge a utilitzar i Gosling va intentar estendre i modificar C++. El resultat va ser el llenguatge C++ + + -- (+ + -- perquè s'afegien i eliminaven característiques a C++), però va abandonar aquesta línia per crear un nou llenguatge des de zero que va anomenar Oak (roure).

El llenguatge Oak havia de ser independent de la plataforma i per aconseguir-ho es va optar per un llenguatge interpretat. A més, el llenguatge havia de ser robust i senzill per evitar errors per part del programador i això va fer que s'eliminessin les característiques que feien el codi propens a errors. El resultat va ser un llenguatge que tenia similituds amb C, C++ i Objective C i que no estava lligat a cap tipus de CPU en concret.

Al mateix temps que es desenvolupava el llenguatge, l'equip treballava en un prototipus de dispositiu anomenat \*7 (*Star Seven*), similar a un PDA, el nom del qual venia de la combinació de tecles del telèfon de l'oficina Green Project que permetia als usuaris respondre al telèfon des de qualsevol lloc.

El Green Project es va donar per finalitzat el 3 de setembre de 1992. En la demostració que van fer als dirigents de Sun Microsystems amb la \*7 controlant diversos dispositius d'electrònica de consum, hi apareixia en el paper d'executor de les ordres de l'usuari, el personatge Duke, creat per Joe Palrang, i que acabaria per convertir-se en la mascota de Java.

Una vegada el Green Project es dona per finalitzat, Sun Microsystems crea una nova empresa filial, FirstPerson, amb l'objectiu de comercialitzar la nova tecnologia. L'objectiu inicial d'aplicar la nova tecnologia al mercat de l'electrònica de consum no va prosperar perquè es disparen els preus dels nous dispositius. FirstPerson intenta cercar altres canals on aplicar la nova tecnologia, apropant-se al mercat de la televisió per cable interactiva, però aquest camí també resulta infructuós. A la fi, Sun Microsystems tanca FirstPerson. En definitiva, el 1993 no va ser un bon any per a la tecnologia Java o, millor dit, tecnologia Oak.

Mentrestant, el 22 d'abril del 1993, el NCSA (National Center for Supercomputing Applications) havia presentat el navegador gràfic Mosaic que obria moltes possibilitats en la Internet incipient. El nombre de llocs web creixia dia a dia i Internet esdevenia un fenomen social.

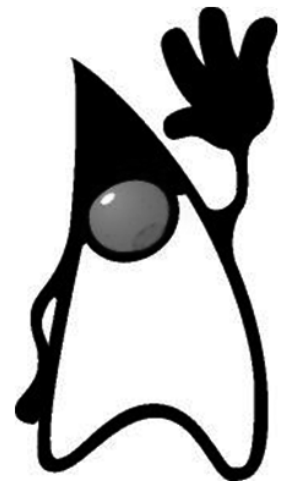
El juny i juliol de 1994, després de tres dies de pluja d'idees dels cervells del Green Project i Sun Microsystems, es decideix orientar la nova tecnologia per al web, ja que s'adonen que, amb l'arribada del navegador Mosaic, Internet estava evolucionant en un medi altament interactiu com havien previst per a la televisió per cable. A partir d'aquí aposten pel llenguatge Oak i desenvolupen un navegador (clon de Mosaic) que inicialment



Dispositiu \*7



Pantalla del dispositiu \*7 mostrant una interfície d'usuari experimental



Duke, mascota de Java

bategen com a WebRunner. Aquell any, el llenguatge Oak fou rebatejat per Java perquè el mot Oak ja estava registrat i, de retruc, el navegador s'acabaria anomenant HotJava.

El 29 de setembre de 1994 s'acaba el desenvolupament del prototipus de HotJava i, en efectuar-se la demostració als executius de Sun Microsystems, es reconeix el potencial de la nova tecnologia.

El 23 de maig de 1995 en la conferència SunWorld '95, John Gage, director de ciència de Sun Microsystems, i Marc Andreessen, cofundador i vicepresident de Netscape Communications Corporation, anunciaven la versió alfa de Java, que en aquell moment només corria en Solaris, i el fet que Java s'incorporaria al Netscape Navigator, el navegador més utilitzat. Aquest fet va portar al llançament definitiu de la tecnologia Java.

### Origen del mot Java

Hi ha diverses llegendes que intenten explicar els orígens del nom *Java*: que si es va posar el nom pensant en un cafè de l'illa de Java que els desenvolupadors de la tecnologia Java preni en una cafeteria propera al seu lloc de treball; que si és l'acrònim corresponent a *just another vague acronym* (només un altre acrònim ambigu); que si són les inicials dels seus creadors (James Gosling, Arthur Van Hoff i Andy Bechtolsheim); que si és un mot d'una llista aleatòria de paraules.

La teoria basada en el tipus de cafè ha estat des de sempre una de les més acceptades, ja que hi ha hagut diversos signes que hi donaven força: el logotip de la tecnologia Java és una tassa de cafè fumejant i els quatre primers bytes de totes les compilacions de codi Java són, en hexadecimal, 0xCAFEBAE.

Però qui millor pot explicar l'origen del mot és el responsable de la creació de la tecnologia Java, James Gosling, que ho explica en un correu del 24 d'agost del 2007 adreçat al llavors president de Sun Microsystems, Jonathan Schwartz, i que es pot consultar en el web del destinatari. La seva transcripció literal en castellà és:

**De:** James Gosling  
**Fecha:** 24 de agosto de 2007 20:16:58 AM PDT  
**A:** Jonathan Schwartz  
**Asunto:** ¿Cómo surgió el nombre de Java?

La historia es la siguiente:

Necesitábamos un nombre. Habíamos estado utilizando la palabra *oak* (roble), que yo había elegido totalmente al azar. Sin embargo, aunque el equipo ya le había cogido cariño, los abogados especialistas en marcas comerciales la rechazaron. Mantuvimos numerosos debates por correo electrónico acerca del nombre, pero no llegamos a ningún acuerdo. Acabamos en una situación muy curiosa: el principal obstáculo para la distribución del producto era el nombre.

Nuestro director de marketing conocía a alguien que era "asesor de nomenclatura" (no recuerdo su nombre, pero era muy bueno). No podíamos permitirnos el dinero ni el tiempo que conllevaba un proceso convencional de nomenclatura de productos. Dicho asesor accedió a hacer algo bastante extraño, pero efectivo y rápido: se prestó como facilitador para una reunión en la que algunos de nosotros nos encerramos en una sala durante toda una tarde. Comenzó formulándonos preguntas del tipo "¿Cómo te hace sentirte esto?" (¡Entusiasmado!). "¿Qué más te hace sentirte de esta forma?" (¡Java!). Acabamos con una pizarra repleta de palabras, en su mayor parte aleatorias. A continuación, nos hizo llevar a cabo un proceso de selección tras el que tuvimos una clasificación de las palabras. Al final, nos quedamos con una docena de posibles candidatos y se los enviamos a los abogados, quienes examinaron la lista hasta que encontraron uno que les convenció. "Java" era el cuarto nombre de la lista. El primero era "Silk" (Seda), que yo odiaba, pero le gustaba a todos los demás. Mi favorito era "Lyric" (Lírico), el tercero de la lista, aunque no pasó la prueba de los abogados. No recuerdo cuáles eran los otros candidatos.

Por tanto, la respuesta a la pregunta “¿Quién le puso el nombre a Java?” no es fácil de responder. El equipo de marketing organizó la reunión, el asesor la dirigió y varios de nosotros soltamos palabras al azar. La verdad es que no estoy seguro de quién fue el primero que dijo “Java”, pero creo que fue Mark Opperman.

Lo que es completamente cierto es que no hubo ninguna mente brillante de marketing que llevara a cabo un proceso creativo coherente.

Aquest document aclareix, per sempre més, l'origen del mot Java i, és interessant tenir en compte la conclusió a què arriba Jonathan Schwartz després d'haver llegit el missatge i que plasma en la mateixa pàgina web:

Como ha pasado con muchas innovaciones, no todas las decisiones (ni nombres de productos, ni blogs ni líneas de códigos) comienzan en una hoja de cálculo. A menudo, la oportunidad es mucho más difícil de identificar.

La taula 1 recull les diverses versions de la tecnologia Java que han aparegut des dels inicis fins al final del 2008 i també la que està en desenvolupament.

Taula 1. Versions de Java aparegudes des dels inicis fins al final del 2008

Versió	Número intern	Data	Nom clau
JDK 1.0		23/01/1996	
JDK 1.1		19/02/1997	
J2SE 1.2		08/12/1998	Playground
J2SE 1.3		08/05/2000	Kestrel
J2SE 1.4		06/02/2002	Merlin
J2SE 5.0	1.5	30/09/2004	Tiger
Java SE 6	1.6	11/12/2006	Mustang
Java SE 7		En desenvolupament	Dolphin

En la taula 1 es veu que les diverses versions s'han batejat amb noms com JDK i J2SE, que corresponen a sigles que tenen significat. Per tant, ens convé introduir el significat de les diverses sigles que per a la nomenclatura de productes i versions s'utilitzen en el món Java. Així tenim:

- JDK, que significa '*Java development kit*' i conté les eines imprescindibles per desenvolupar programes en el llenguatge Java i obtenir-ne compilats.
- JRE, que significa '*Java runtime environment*' i que és necessari tenir instal·lat en un ordinador en què es vulguin executar programes Java. Aquest entorn inclou la JVM (*Java virtual machine*) que es posa en marxa quan executem un programa Java en el nostre ordinador.
- J2SE / Java SE, que correspon a *standard edition*, és el paquet bàsic per desenvolupar aplicacions en llenguatge Java que no necessitin un servidor d'aplicacions i està format pels JDK i JRE corresponents de la versió.

- J2EE / Java EE, que correspon a *enterprise edition*, és una plataforma de programació que amplia la Java SE i permet desenvolupar i executar aplicacions Java amb arquitectura d'N nivells i distribuïda basada en components de programari modulars que s'executen sobre servidors d'aplicacions. Inclou tot el contingut de la Java SE corresponent.
- J2ME / Java ME, que significa '*micro edition*', és una col·lecció d'API de Java per al desenvolupament de programari per dispositius de recursos limitats, com PDA, telèfons mòbils i altres aparells de consum.

## 1.2. Components principals de la tecnologia Java

La tecnologia Java està dissenyada per suportar aplicacions que s'executin en els entorns de xarxa més variats, des del Linux fins al Windows, passant pel Mac i estacions de treball, sobre arquitectures diferents i amb sistemes operatius diversos. Per complir aquests requisits d'execució tan variats, el compilador de Java genera *bytecodes*: un format intermedi independent de l'arquitectura dissenyat per transportar el codi eficientment a diferents plataformes de maquinari i de programari. Els *bytecodes* són interpretats i executats en les diverses plataformes per una màquina virtual Java (JVM).

Així, doncs, la tecnologia Java es basa en dos components essencials: el llenguatge Java i la màquina virtual Java (JVM).

### 1.2.1. La màquina virtual Java (JVM)

La màquina virtual Java (JVM) és l'entorn en què s'executen els programes Java i és un programa natiu, és a dir, executable en una plataforma específica, que és capaç d'interpretar i executar instruccions expressades en un codi binari especial (els *bytecodes* de Java), que és generat pel compilador del llenguatge Java.

El codi binari de Java (*bytecodes*) no és un llenguatge d'alt nivell, sinó un veritable codi màquina de baix nivell, viable fins i tot com a llenguatge d'entrada per a un microprocessador físic.

La màquina virtual Java és una peça fonamental de la tecnologia Java i se situa en un nivell superior al maquinari sobre el qual es vol executar l'aplicació i actua com un pont entre el *bytecode* a executar i el sistema sobre el qual es vol executar. Així, quan un programador escriu una aplicació Java, ho fa pensant en la JVM encarregada d'executar l'aplicació i no hi ha cap motiu per pensar en la plataforma física sobre la qual s'ha d'executar

#### L'API d'un llenguatge de programació...

... és el conjunt més o menys ampli de biblioteques contenidores de tipus de dades, accions i funcions que els llenguatges de programació proporcionen perquè la comunitat de programadors els utilitzin.

En el llenguatge Java, en ocasions es parla de l'API de Java per referir-nos a totes les biblioteques que facilita el llenguatge Java i a vegades s'utilitza el nom API de Java per referir-se a subconjunts de l'API global de Java.

l'aplicació. La JVM serà l'encarregada, en executar l'aplicació, de convertir el codi *bytecode* a codi natiu de la plataforma física.

El gran avantatge de la JVM és que possibilita la portabilitat de l'aplicació a diferents plataformes i, així, un programa Java escrit en un sistema operatiu Windows es pot executar en altres sistemes operatius (Linux, Solaris i Apple OS X) amb l'únic requeriment de disposar de la JVM per al sistema corresponent. D'aquesta manera s'assoleix l'objectiu inicial de Gosling quan va iniciar el desenvolupament de la tecnologia Java (*write once, run anywhere*).

El concepte de màquina virtual Java s'usa en dos àmbits: d'una banda, per fer referència al conjunt d'especificacions que ha de complir qualsevol implementació de la JVM; d'altra banda, per fer referència a les diverses implementacions de la màquina virtual Java existents i de les quals cal utilitzar-ne alguna per executar les aplicacions Java.

L'empresa Sun Microsystems és la propietària de la marca registrada Java, i aquesta s'utilitza per certificar les implementacions de la JVM que s'ajusten i són totalment compatibles amb les especificacions de la JVM, en el prefaci de les quals es diu: "Esperem que aquesta especificació documenti suficientment la màquina virtual de Java per fer possibles implementacions des de zero. Sun proporciona tests que verifiquen que les implementacions de la màquina virtual Java operen correctament."

Hi ha hagut diverses versions, en ordre cronològic, d'especificacions de la màquina virtual Java. En general, la definició dels *bytecodes* no canvia significativament entre versions i, si ho fa, els desenvolupadors del llenguatge Java procuren que hi hagi compatibilitat cap enrere amb les versions anteriors.

Respecte a les implementacions de la JVM, les podem trobar en diferents formes:

- En l'entorn d'execució JRE de Sun Microsystems que podem tenir instal·lat en el nostre sistema (Windows, Linux, Solaris, Apple OS X) si volem executar una aplicació Java.
- En entorns d'execució propietaris, com Excelsior Jet, J9 d'IBM, MS JVM de Microsoft, JRockit originària de BEA Systems i ara a les mans d'Oracle i d'altres.
- En entorns d'execució de codi obert o lliure, com HotSpot, Jamiga, Jessica, Kaffe, Mika VM, NanoVM i moltes altres.

- En processadors que directament acceptarien els *bytecodes* de Java com el seu llenguatge màquina, anomenats *processadors Java*.

### Processadors Java

Fins ara, només uns pocs processadors o projectes de processador Java estan disponibles:

- Especificació de processador *pico.Java* a càrrec de l'empresa Sun Microsystems, que no ha arribat a implementar en cap processador però que ha donat llicència per a l'ús d'aquesta tecnologia a companyies com Fujitsu, NEC i Siemens.
  - Processadors *aJ-100*, *aJ-102* i *aJ-200* de l'empresa californiana aJile Systems, dedicada als dispositius mòbils.
  - Processador *Vivaja CPU* de l'empresa romana Vivaja Technologies LLC, dedicada als dispositius mòbils.
  - Processador *IM1101* de l'empresa sueca lmsys Technologies AB, que dissenya i subministra solucions multiprocessador en xarxa a OEMs en el mercat de sistemes telemàtics i de control.
  - Projecte Komodo, de les universitats alemanyes de Karlsruhe i Augsburg, consistent en un microcontrolador multifil en temps real.
  - Processador *ARM926EJ-S* de l'empresa anglesa ARM Holdings, considerada empresa dominant en el mercat dels xips per telèfons mòbils.
  - Processador *JOP (Java optimized processor)* de codi obert amb llicència GNS GLP, implementat sobre dispositius FPGA (*field programmable gate array*) que permeten ser programats un cop fabricats.
  - Processador *SHAP* de la universitat alemanya de Dresden.
  - Processador *jHISCH* de la universitat de Hong Kong.
  - Projecte de recerca *Femto.Java* en què treballen diverses universitats per implementar processadors sobre dispositius FPGA.
- En la majoria de navegadors web, que incorporen un intèrpret per la tecnologia Java i un entorn d'execució de manera que els *applets* que es descarreguen puguin ser executats.

Per finalitzar amb el coneixement bàsic de la JVM, cal fer una reflexió entorn del fet que la JVM és un intèrpret i els entorns d'execució interpretats sempre han estat molt lents comparats amb els entorns d'execució de codi màquina natiu.

Els *bytecodes* de Java són en realitat codi màquina per a la JVM, motiu pel qual el procés d'interpretació és molt més ràpid que en altres entorns interpretats en què la interpretació es realitza a partir del llenguatge d'alt nivell (aquí seria el llenguatge Java).

A més, hi ha màquines virtuals que incorporen els anomenats JIT (*just in time compiler*) que tradueixen els *bytecodes* a codi natiu optimitzat una sola vegada, de manera que cada vegada que la JVM torna a cridar el mateix codi s'executa directament la versió ja interpretada. Això implica una mica més de lentitud la primera vegada que s'executa (interpreta) el codi, però en les execucions posteriors del mateix codi es guanya eficiència.

### Applets

Els *applets* són programes escrits en llenguatge Java que resideixen en els servidors web i es descarreguen en els navegadors dels sistemes clients, on són executats pel mateix navegador. Acostumen a ser de petita grandària per minimitzar el temps de descàrrega i es criden des de pàgines HTML.

El Centre de Terminologia Termcat per a la llengua catalana dóna tres traduccions per al mot *applet*:

- Miniaplicació.
- Miniaplicació Java.
- Miniaplicació de servidor.

Nosaltres considerem que cap de les traduccions cospa el veritable significat del mot *applet*, motiu pel qual en aquest material es mantindrà aquest mot.

### 1.2.2. El llenguatge Java

Abans d'iniciar l'aprenentatge del llenguatge Java, cal que en coneguem les característiques principals.

Java és un llenguatge que permet diferents tipus d'aplicacions.

Amb el llenguatge Java podem crear aplicacions de diferents tipologies:

- *Applets*, que resideixen en els servidors web i es descarreguen en els navegadors dels sistemes clients, on són executats pel mateix navegador. Acostumen a ser de petita grandària per minimitzar el temps de descàrrega i es criden des de pàgines HTML.
- *Servlets*, miniaplicacions de servidor executades per un servidor d'aplicacions i que responen a crides HTTP per servir pàgines web dinàmiques.
- *JavaBeans*, components de programari Java reutilitzables que es poden manipular visualment en una eina de desenvolupament.
- Aplicacions independents per executar directament per un entorn d'execució com JRE de Sun Microsystems.

Java és un llenguatge simple.

El llenguatge Java va néixer després d'intentar evolucionar el llenguatge C++ cap una versió més simple i, per tant, facilita tota la funcionalitat d'un llenguatge potent com el llenguatge C++ però sense les característiques menys utilitzades i més complicades d'aquell.

A tall d'exemple, respecte al llenguatge C++, el llenguatge Java elimina:

- L'aritmètica de punters.
- Les referències.
- Els registres (*struct*), tot substituint-los per les classes.
- La definició de tipus (*typedef*), tot substituint-la per les classes.
- Les macros (*#define*).
- La necessitat d'alliberar la memòria dinàmica assignada (no existeix cap operació equivalent a les operacions *free* i *delete* de C++).
- L'herència múltiple.
- Sentència *goto*.
- Variables globals.

#### El mot *servlet*

El Centre de Terminologia Termcat per a la llengua catalana tradueix *servlet* per 'miniaplicació de servidor'.

Nosaltres considerem que aquesta traducció no cospa el veritable significat del mot *servlet*, motiu pel qual en aquest material es mantindrà aquest mot.

La comparació de Java amb el llenguatge C++ és de molta utilitat per als programadors que coneixen el llenguatge C++.



I incorpora característiques molt útils, com:

- El *garbage collector* (recuperador de memòria), que és un procés que s'executa periòdicament de manera automàtica i s'encarrega d'alliberar la memòria dinàmica assignada prèviament que ha deixat de ser utilitzada. És el responsable de la inexistència d'operacions equivalents a les operacions `free` i `delete` de C++.
- Resolució dinàmica de mètodes, que permet en temps d'execució connectar la classe a què pertany un objecte determinat i, per tant, aplicar-hi els mètodes de la classe a què pertany.

#### Aclariment referent a la resolució dinàmica de mètodes en els llenguatges Java i C++

Suposem que `X` és una classe d'objectes i que `Y` és una classe derivada de la classe `X`, és a dir, classe heretada de la classe `X`.

Suposem, en el llenguatge C++, que hi ha un punter `px` a la classe `X`, o en el llenguatge Java, una variable `px` de la classe `X`. És a dir:

En C++:        `X *px;`  
En Java:        `X px;`

En tot cas, `px` no és cap objecte sinó que és una variable per apuntar a un objecte quan aquest sigui creat, cosa que tant en el llenguatge C++ com en el llenguatge Java s'aconsegueix amb l'operador `new`:

```
px = new X(); /* px apunta a un objecte de la classe X
px = new Y(); /* px apunta a un objecte de la classe Y
```

Fixem-nos que `px`, declarat per apuntar objectes de la classe `X`, pot apuntar objectes de qualsevol classe derivada de la classe `X` (efectes de l'herència).

Suposem ara que la classe `X` té definit un mètode anomenat `metode()` i que la classe `Y` també té definit un mètode amb el mateix nom, que no és producte de l'herència (és a dir, s'ha definit de nou a la classe `Y`). Què passa quan en temps d'execució es troba una crida del mètode sobre l'objecte apuntat per `px` tenint en compte que `px` pot apuntar un objecte de la classe `X` o un objecte de la classe `Y`?

```
En C++:px->metode();
En Java:px.metode();
```

Bé, se suposa que esteu esperant que s'executi `metode()` de la classe a què correspon l'objecte apuntat per `px` en el moment d'executar la instrucció. És a dir, si `px` apunta un objecte de la classe `Y`, que s'executi `metode()` de la classe `Y`, malgrat que `px` hagi estat declarat com a "punter a" (llenguatge C++) o "variable de" (llenguatge Java) la classe `X`.

El llenguatge Java té la funcionalitat esperada, mentre que en el llenguatge C++ impera el tipus per al qual ha estat declarat el punter i executa `metode()` de la classe `X` malgrat que el punter estigui apuntant un objecte d'una classe derivada. En el llenguatge C++ aquest funcionament lògic es pot aconseguir definint mètodes virtuals... En el llenguatge Java és molt més simple: és el funcionament per defecte! No cal introduir el concepte de mètodes virtuals!

Per acabar, volem comentar que la sintaxi del llenguatge Java és molt similar al llenguatge C++ i, per tant, el seu aprenentatge és molt senzill per als programadors en C++. I per als programadors que no coneixin el llenguatge C++, volem comentar que és un llenguatge més

#### Garbage collector

El Centre de Terminologia Termcat, responsable de la revisió dels termes catalans i la normalització dels neologismes, ha establert la forma "recuperador de memòria" per referir-se al *garbage collector*.

És molt possible que aquesta forma s'acabi imposant, però en aquest moment, la bibliografia existent en llengua catalana tradueix *garbage collector* per a formes més properes a *recol·lector de deixalles*.

senzill que qualsevol altre entorn de programació. Ara bé, l'obstacle que es pot presentar és aconseguir comprendre la programació orientada a objectes, fet que és independent del llenguatge.

Java és un llenguatge distribuït.

Java proporciona una col·lecció de classes per utilitzar en aplicacions de xarxa que permeten obrir sòcols (*sockets*) i establir i acceptar connexions amb servidors o clients remots, la qual cosa facilita la creació d'aplicacions distribuïdes.

Java és un llenguatge orientat a objectes.

El llenguatge Java va ser dissenyat com un llenguatge orientat a objectes des del principi i implementa la tecnologia bàsica de C++ amb algunes millores i elimina alguns aspectes per mantenir l'objectiu de la simplicitat del llenguatge. No cal dir que dóna suport a les tres característiques bàsiques de l'orientació a objectes: encapsulació, herència i polimorfisme.

És un llenguatge orientat a objectes pur en el sentit que no hi ha cap variable, funció o constant que no estigui dins d'una classe i s'accedeix als membres de les classes (dades i mètodes) per mitjà dels objectes. Per raons d'eficiència s'han conservat vuit tipus de dades bàsics (*byte*, *short*, *int*, *long*, *float*, *double*, *char* i *boolean*) com a tipus primitius, tot i que també es proporcionen les classes corresponents (*classes embolcalls*) per si es vol que totes les dades es gestionin mitjançant objectes de classes.

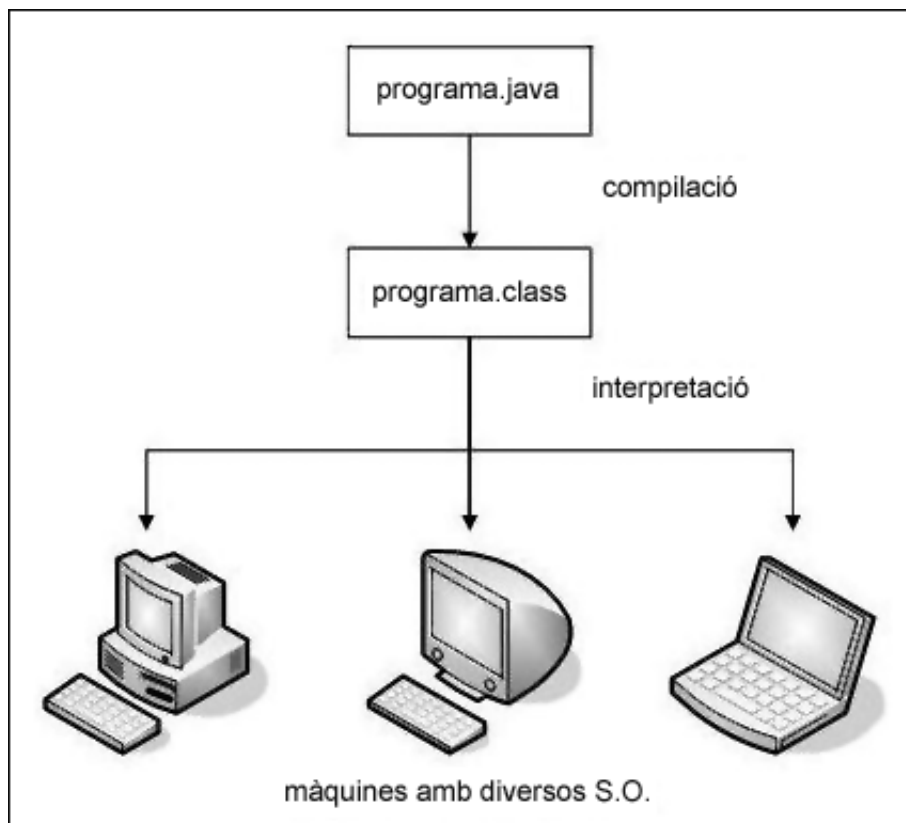
Java és un llenguatge compilat i interpretat.

El codi font dels programes Java resideix en fitxers d'extensió `.java` sobre els quals s'aplica un procés de compilació i s'obtenen els fitxers d'extensió corresponents `.class`. Els fitxers compilats contenen un conjunt d'instruccions anomenades *bytecodes* que són independents del tipus d'ordinador en què es vulgui executar el programa. Com es mostra en la figura 1, l'execució del mateix compilat en diverses plataformes (*write once, run anywhere*) és possible gràcies a l'existència d'interprets específics per a cada plataforma (màquina virtual Java), que són els encarregats de l'execució.

#### Sòcol

Un sòcol és un espai virtual d'un node d'una xarxa reservat per executar una única aplicació o un únic procés i que serveix per mantenir una comunicació.

Figura 1. Fases de compilació i interpretació en la tecnologia Java



Java és un llenguatge robust.

Java es va dissenyar per crear programari molt fiable. Per això proporciona moltes comprovacions en fase de compilació i en fase d'execució. Destaquem:

- **Les característiques de gestió de memòria** que alliberen els programadors de tots els errors relatius a l'aritmètica de punters, ja que s'ha prescindit totalment dels punters i el recuperador de memòria elimina la necessitat de gestionar l'alliberament explícit de la memòria.
- **La comprovació dels límits de les taules.** Recordem que el llenguatge C no comprova mai si l'accés a una posició d'una taula és correcte, de manera que l'accés a una posició inexistente es converteix, en realitat, en un accés a una posició de memòria ocupada per altres dades.
- **L'obligatorietat d'inicialitzar les variables.** En altres llenguatges de programació, els valors de les variables no inicialitzades són indeterminats, motiu pel qual si hi accedim abans d'inicialitzar-les podem obtenir resultats impredecibles. Això en Java no és així: les variables locals s'han d'inicialitzar obligatòriament, de manera que si un programa intenta utilitzar una variable local abans d'assignar-li un valor, el compilador genera un error.

- **La conversió segura de tipus.** Només es permet conversió de tipus entre certs tipus primitius del llenguatge Java (`int` i `long`) i entre objectes de la mateixa branca de l'arbre d'herència (un que hereta de l'altre però no a l'inrevés).

### Conversió de tipus (*typecasting*) en C/C++ i Java

La conversió de tipus és una característica que possibiliten molts llenguatges de programació i que permet considerar una entitat d'un tipus determinat com si fos d'un altre tipus.

Així, per exemple, en el llenguatge C, considerem:

---

```
int i=10;
float r=10.2;

if (r>i) printf ("%g > %d ? Cert\n", r, i);
else printf ("%g > %d ? Fals\n", r, i);
```

---

L'execució d'aquest tros de programa dóna el resultat:

---

```
10.2 > 10 ? Cert
```

---

Com ha pogut el llenguatge comparar un valor `float` amb un valor `int`? Doncs perquè abans de fer la comparació s'ha produït una conversió automàtica de tipus i el valor `int` l'ha considerat com a `float`. En aquest cas s'ha produït una conversió implícita de tipus.

Considerem, en el llenguatge C++, l'exemple següent:

---

```
class X1
{
    int a1;
    float b1;
};

class X2
{
    float a2;
    int b2;
};

void main()
{
    int i=10;
    float r=10.2;
    X1 *x1; /* Punter per apuntes un objecte de la classe X1 */
    X2 *x2; /* Punter per apuntes un objecte de la classe X2 */

    x2=new X2(); /* Nou objecte de la classe X2, apuntat per x2 */

    x1=x2; /* (1) */
    x1=(X1 *)x2; /* (2) */
}
```

---

En les dues darreres línies de l'exemple es vol utilitzar el punter `x1` (pensat per apuntar objectes de la classe `X1`) per apuntar l'objecte prèviament creat de la classe `X2` que està apuntat per `x2`. El compilador de C++ no permetrà mai efectuar l'assignació (1) i en tal cas ens ho comunicaria amb un error similar a "Cannot convert 'X2 \*' to 'X1 \*'". Però, en canvi, si el programador li indica l'opció (2), el compilador en fa cas. Evidentment, els resultats que es puguin produir en operar amb l'objecte de la classe `X2` mitjançant el punter `x1` són imprevisibles.

El llenguatge Java, mai no permetrà efectuar l'assignació de cap de les dues opcions (1) i (2), tret que la classe `X2` sigui una classe heretada de la classe `X1`. En aquesta situació sí que és permès que una variable de la classe `X1` faci referència a un objecte de la classe `X2`.

- La gestió d'excepcions per detectar els errors que es puguin produir en temps d'execució i, sigui quin sigui el punt en què es produeixin, gestionar-los allí mateix o propagar-los al nivell superior perquè hi siguin gestionats.
- La possibilitat de prohibir l'herència sobre certes classes i/o la sobreescritura de certs mètodes, tot declarant-los de tipus *final*.

Java és un llenguatge segur.

Java es va dissenyar, des del principi, pensant en la seguretat. Això és especialment important atesa la naturalesa distribuïda de Java. Sense certes garanties ningú no voldria baixar codi d'Internet i executar-lo en l'ordinador, que és el que fan tots els dies milions de persones amb els *applets* de Java.

Java no és un llenguatge totalment segur, cap llenguatge ho és, però es diu que és un llenguatge segur perquè en el seu desenvolupament s'han tingut en compte els mecanismes que es poden utilitzar per corrompre el programari, de manera que s'han anat aixecant diverses barreres per impossibilitar-ho. A continuació presentem aquestes barreres en un ordre que les faci comprensibles i després ja veurem en quin ordre estan situades en el procés d'execució d'una aplicació Java.

**1)** La primera barrera la proporciona la **robustesa del mateix llenguatge**, ja explicitada (gestió de memòria, comprovació dels límits de les taules, inicialització de variables, conversió segura de tipus, gestió d'excepcions, prohibició d'heretar d'una classe final i de sobreesciure un mètode final).

Però, tot i la robustesa del llenguatge, què passaria si es desenvolupés un compilador capaç de generar *bytecodes* de Java passant per alt totes les proteccions subministrades pel llenguatge i pel compilador de Java? Ens cal, doncs, verificar la correcció dels *bytecodes* a executar i això provoca l'aparició d'una nova barrera.

**2)** La segona barrera la proporciona el **verificador dels arxius de classes** (*class file verifier*) que incorpora la JVM i que té per funció validar els *bytecodes* a interpretar i executar.

Encara que aquesta tasca pot semblar absurda, no ho és, ja que no tots els *bytecodes* han de ser correctes, ja que es poden generar a mà o utilitzant compiladors de Java modificats.

El sistema distingeix entre el codi en què es confia (generalment, les classes del sistema i les validades per l'usuari) i el codi en què no es con-

fia. Les classes considerades segures ja no es validen, però la resta sí. Com que el verificador d'arxius de classes forma part de la JVM, no hi ha perill que sigui reemplaçat per l'aplicació que executarem, ja que només es pot reemplaçar substituint la JVM.

En les classes que valida, hi cerca intents de fabricar punters, executar instruccions de codi natiu, cridar mètodes (funcions) amb paràmetres no vàlids, usar variables abans de ser inicialitzades, declarar classes heretades de classes catalogades com a `final`...

Però, tot i el verificador de classes, què passaria si, per la xarxa (per exemple, en la descàrrega d'un *applet*), arriba una classe que reemplaça una classe crítica del sistema, per exemple, la classe `SecurityManager`? Ens cal, doncs, verificar les classes a carregar i això provoca l'aparició d'una nova barrera.

La classe `SecurityManager` és la classe que controla els accessos als recursos en temps d'execució.

**3)** La tercera barrera la proporciona el **carregador de classes** (*class loader*) i s'encarrega de separar les classes que carrega per evitar atacs.

En Java 2 es defineixen tres grups de classes associades a diferents camins de cerca:

- Classes del sistema (associades al camí d'arrencada o *boot class path*).
- Classes d'extensió del sistema (associades a la ruta d'extensió o *extension class path*).
- Classes de l'usuari o aplicació (associades a la ruta de classes de l'usuari o aplicació *user class path*).

L'ordre de cerca de les classes és el següent: sistema, extensió i usuari. Quan es troba una classe, s'atura la cerca.

Per tant, una aplicació només podria reemplaçar una classe del sistema modificant la ruta d'arrencada i això no és possible sense tenir accés total al sistema.

El carregador de classes és una classe Java i es pot estendre (redefinir) per definir carregadors de classes especials, però només per a les aplicacions; si un *applet* pogués definir el seu propi carregador podria modificar el carregador del sistema i, potencialment, apoderar-se de la màquina en què s'executa el navegador que ha descarregat l'*applet*.

Però tot i el carregador de classes, seria possible que algun recurs del sistema (sistema de fitxers, per exemple) fos fàcilment accessible per qualsevol classe? Per evitar-ho, apareix una nova barrera.

**4)** La quarta barrera és el **gestor de seguretat** (`SecurityManager`). El gestor de seguretat és una classe del sistema que s'encarrega de comprovar l'accés als recursos en temps d'execució i a l'igual del carregador de classes, pot ser estesa (redefinida) per les aplicacions.

Els recursos sobre els quals té control són múltiples: E/S de xarxa i fitxers, creació de carregadors de classes, manipulació de fils d'execució, execució de programes nadius, aturada de la JVM, càrrega de codi natiu en la màquina virtual, realització de determinades operacions en l'entorn de finestres o càrrega de certs tipus de classes.

El gestor de seguretat ha estat la barrera que més ha evolucionat en les diverses versions de la tecnologia Java. Des de la versió Java 2, la màquina virtual crida mètodes del gestor de seguretat per determinar la política de seguretat activa (definible en el sistema) i realitzar verificacions de control d'accés.

Per defecte, quan s'executen *applets*, sempre es carrega una implementació del gestor de seguretat que és totalment restrictiva, és a dir, no permet que els *applets* tinguin accés a cap recurs del sistema tret que haguem explicitat els permisos en el sistema client.

En canvi, per a les aplicacions, no és així, i per defecte tenen accés a tots els recursos del sistema. L'usuari que posa en marxa l'aplicació pot indicar que vol cridar un gestor de seguretat i, a partir d'aquest moment, la màquina virtual aplicarà la política de seguretat activa.

Un aspecte molt important del gestor de seguretat és que una vegada carregat no es pot reemplaçar, de manera que ni els *applets* ni les aplicacions en execució poden instal·lar el seu quan l'usuari (en el cas de les aplicacions) o el sistema (en el cas dels *applets*) ja n'han carregat un.

#### **Seguretat en la tecnologia Java de la versió JDK 1.0**

El model de seguretat original de la tecnologia Java és el conegut com el model del calaix de sorra (*sandbox model*), que proporcionava un entorn molt restringit en el qual es podia executar codi no fiable obtingut de la xarxa. El nom del model ve dels calaixos de sorra en què es deixa jugant els nens petits, de manera que poden fer tot el que vulguin en el seu interior però no poden sortir a l'exterior.

En aquest model treballem amb dos nivells d'accés als recursos: total, per als programes locals, i molt restringit, per als programes remots.

La pega fonamental d'aquest model és que és molt restrictiu, ja que no permet que els programes remots facin gairebé res d'útil, perquè estan restringits al model del calaix de sorra.

#### **Seguretat en la tecnologia Java de la versió JDK 1.1**

Com el model del JDK 1.0 era massa restrictiu, es va introduir el concepte de codi remot signat, que segueix garantint la seguretat dels clients, però permet que el codi obtingut remotament surti del

calaix i tingui accés a tots els recursos, sempre que estigui firmat per una entitat en què el client confia.

Encara que això millora una mica la situació, continua essent un control de dos nivells: total per al codi local, o remot signat i restringit per al codi remot no signat o amb signatures no validades pel client.

A més del codi signat, el JDK 1.1 va introduir altres millores de seguretat:

- **Un parell d'eines de seguretat:** el programa `jar` que és un programa arxivador que permet reunir un conjunt de classes i altres fitxers (com, per exemple, imatges o text) en un sol arxiu, emmagatzemat normalment amb l'extensió `.jar`, que fa possible la transferència d'aplicacions d'un mode compacte en una sola connexió, i signant tots els programes de manera conjunta; i el programa `javakey`, que permetia efectuar la signatura de classes en els fitxers `.jar`.
- **Una API per a programació segura,** que introduïa paquets de classes que proporcionen funcions criptogràfiques als programadors i eines per generar firmes digitals i la seva gestió.

## Seguretat en Java 2

En la versió JDK 1.2 es van introduir noves característiques que milloren el suport i el control de la seguretat:

- **Control d'accés de grànul fi.** Un dels problemes fonamentals de la seguretat JDK 1.1 és que el model només tenia en compte dos nivells de permisos: accés total o calaix de sorra (restricció total). Per solucionar aquest problema s'introdueix un sistema de control de permisos de grànul fi que permet donar permisos específics a trossos de codi específics per accedir a recursos específics en el client, depenent de la signatura del codi i/o de l'URL d'origen.
- **Control d'accés aplicat a tot el codi.** El concepte de codi signat ara és aplicable a tot el codi, independentment de la seva procedència (local o remota).
- **Facilitat de configuració de polítiques de seguretat.** La nova arquitectura de seguretat permet ajustar de manera senzilla els permisos d'accés utilitzant un fitxer de polítiques (*policy file*) en què es defineixen els permisos per accedir als recursos del sistema per a tot el codi (local o remot, signat o sense signar). Gràcies a això, l'usuari es pot baixar aplicacions de la xarxa, instal·lar-les i executar-les tot assignant només els permisos que necessiten.
- **Estructura de control d'accés extensible.** En versions anteriors de JDK, quan es volia crear un nou tipus de permís d'accés, era necessari afegir un nou mètode *check* en la classe del gestor de seguretat. La versió JDK 1.2 permet definir permisos tipus que representen un accés als recursos del sistema i el control automàtic de tots els permisos d'un tipus correcte, la qual cosa repercuteix en el fet que, en la majoria de casos, és innecessari afegir mètodes al gestor de seguretat.

Com va succeir amb el JDK 1.1, en el JDK 1.2 van aparèixer o millorar les eines i l'API de seguretat:

Respecte a les eines, en el llenguatge Java 2 es disposa del següent: el programa `jar`, similar al JDK 1.1; el programa `keytool`, per a la generació de claus i certificats; el programa `jarsigner` per signar fitxers `.jar` i verificar les signatures de fitxers `.jar` ja signats (substituint l'eina anterior `javakey`), i la utilitat `policytool`, per crear i modificar els fitxers de configuració de polítiques de seguretat del client.

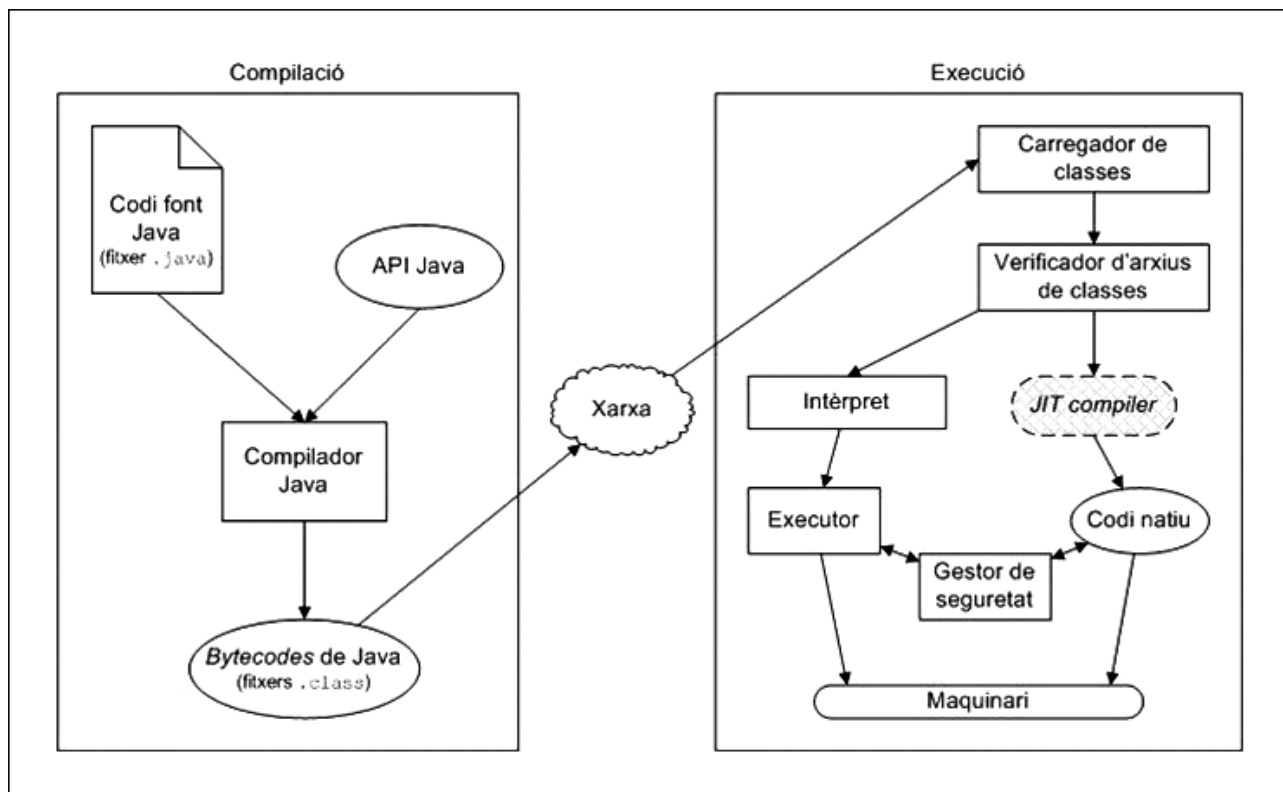
L'API de seguretat es va veure incrementada amb nous paquets que milloren el suport de certificats.

Una vegada coneguts els actors principals que fan de Java un entorn segur (llenguatge, verificador dels arxius de classes, carregador de classes i gestor de seguretat), estem en condicions de veure una aproximació de com interactuen.

La figura 2 ens mostra una aproximació de la ubicació dels diversos actors en les fases de compilació i execució dels programes Java.



Figura 2. Actors en les fases de compilació i execució de les aplicacions Java



Java és un llenguatge portable entre diferents plataformes.

Més enllà de la portabilitat bàsica donada pel caràcter d'arquitectura independent de la tecnologia Java, Java implementa altres estàndards de portabilitat per facilitar el desenvolupament. Així, els valors enters de tipus `int` sempre són nombres enters representats en 32 bits en complement de dos, independentment de la plataforma d'execució. A més, construeix les seves interfícies d'usuari per mitjà d'un sistema abstracte de finestres (AWT) de manera que les finestres es puguin implementar en qualsevol entorn (Linux, Windows o Mac).

Java és un llenguatge multifil.

Avui en dia les aplicacions que només poden executar una acció a la vegada es consideren molt limitades. El llenguatge Java suporta sincronització de múltiples fils d'execució (*multithreading*) per al llenguatge, especialment útils en la creació d'aplicacions de xarxa distribuïdes. Així, mentre un fil s'encarrega de la comunicació, un altre pot interactuar amb l'usuari mentre un altre presenta una animació en pantalla i un altre realitza càlculs.

El Java és un llenguatge dinàmic.

El llenguatge Java i el seu sistema d'execució en temps real són dinàmics en la fase d'enllaç, és a dir, les classes només s'enllacen en el moment en què es necessiten. Això permet que es puguin enllaçar nous mòduls de codi sota demanda, procedents de fonts molt variades, fins i tot de la xarxa.

### 1.3. Eines bàsiques per desenvolupar programes Java

En el món del desenvolupament en Java hi podem trobar infinitat d'entorns de desenvolupament integrats (IDE) per facilitar la tasca dels desenvolupadors. Així podem destacar:

- *Netbeans*, projecte multiplataforma (Windows, Linux, Mac, Solaris) de codi obert fundat per Sun Microsystems el juny de 2000, desenvolupat en Java.
- *Eclipse*, projecte multiplataforma (Windows, Linux, Mac) de codi obert fundat per IBM el novembre de 2001, desenvolupat en Java.
- *Anjuta DevStudio*, per al GNU/Linux creat per Naba Kumar el 1999.
- *JBuilder*, eina multiplataforma (Windows, Linux, Mac) propietat de l'empresa Borland, apareguda el 1995. La versió 2008 incorpora tres edicions (*Enterprise* –de pagament–, *Professional* –de pagament– i *Turbo* –gratuïta–).
- *JDeveloper*, eina multiplataforma (Windows, Linux, Mac) gratuïta propietat de l'empresa Oracle, apareguda el 1998 inicialment basada en *JBuilder* però desenvolupada des de 2001 en Java.

Les eines aquí enumerades potser són les més representatives en el món Java, però cal tenir en compte que n'hi ha moltes més.

La utilització d'un IDE desconegut per iniciar l'aprenentatge d'un llenguatge no sempre és aconsellada, ja que s'ajunten les dificultats inherents a l'aprenentatge del llenguatge amb les dificultats inherents a l'aprenentatge de la utilització de l'IDE.

Per a l'aprenentatge inicial de Java no ens és necessari utilitzar cap IDE. En tenim prou amb el següent:

- Un editor de textos. Pot servir qualsevol, fins i tot el Notepad del Windows o el *vi* del Linux. Ara bé, cal conèixer l'existència d'editors de textos pensats per donar suport a llenguatges de programació, com el Notepad++, i seria una ximpleria no utilitzar-los.
- El compilador (`javac`) i l'interpret (`java`) proporcionats per Sun Microsystems en el JDK, igual que altres eines incloses en el JDK com

#### IDE

Els entorns de desenvolupament integrat (*integrated development environment*, IDE) són programes que proporcionen un conjunt d'eines al programador i que poden estar ideats per a un únic llenguatge de programació o per a més.

Entre els seus components bàsics trobem un editor de textos sensible a la sintaxi del llenguatge de desenvolupament, un compilador, un intèrpret (en cas que el llenguatge sigui interpretat), un entorn d'execució i un depurador de codi. Molts IDE actuals també incorporen control de versions i facilitats per a la construcció d'interfícies gràfiques d'usuaris.

#### Notepad ++

És un editor de codi font lliure que admet molts llenguatges de programació i s'executa en Windows, tot i que ja fa temps es pot utilitzar en GNU/Linux mitjançant la tecnologia *Wine*.

l'ordre `javap` per obtenir informació bàsica de les classes i l'ordre `jar` per obtenir arxius `.jar` pensats per distribuir aplicacions Java.

Una manera ràpida per saber si tenim aquestes eines instal·lades correctament és executar aquestes ordres (`java`, `javac`, `javap`, `jar`...) des d'una consola de sistema. En cas de no estar instal·lades correctament, el sistema operatiu es queixarà indicant-nos que no les troba. Si l'eina està instal·lada correctament, acostuma a donar per resposta la informació de les diverses opcions amb les quals es pot executar. Fent cas de la informació subministrada per `javac`, si volem conèixer la versió de compilador de Java instal·lada podem fer:

```
C:\>javac -version
javac 1.6.0_11

C:\>java -version
java version "1.6.0_11"
Java(TM) SE Runtime Environment (build 1.6.0_11-b03)
Java HotSpot(TM) Client VM (build 11.0-b16, mixed mode,
sharing)
```

En la resposta veiem que tenim instal·lada la versió 1.6 (Java SE 6), en el seu *release* 11.

- La documentació del llenguatge Java, proporcionada per Sun Microsystems, molt més extensa que la subministrada per l'eina `javap`.

## 1.4. El primer programa Java

L'inici de l'aprenentatge d'un llenguatge de programació s'acostuma a realitzar visualitzant un missatge per pantalla, i nosaltres no trencarem aquesta tradició. Com probablement sabeu, el missatge acostuma a ser "Hola Món".

### 1.4.1. El codi font

El codi font del programa Java que mostra el missatge "Hola Món" és:

```
/* Fitxer: HolaMon.java
   Descripció: Programa que visualitza el missatge "Hola Món" per la consola
   Autor: Isidre Guixà
*/
class HolaMon
{
    public static void main (String args[])
    {
        System.out.println ("Hola Món");
    }
}
```



Per a instal·lar l'entorn de desenvolupament de Java (JDK) subministrat per Sun Microsystems, vegeu el contingut "Instal·lació de l'entorn de desenvolupament Java en l'MSWindows".



Trobareu el fitxer `HolaMon.java` en la secció "Recursos de contingut" del web.

Analitzem els diferents elements que configuren aquest programa.

- Les quatre primeres línies, tancades entre els símbols `/*` i `*/` corresponen a comentaris.
- La cinquena línia declara la classe: `class HolaMon`.
  - Tot el codi Java ha d'anar dins d'una declaració de classe. Com a mínim hi ha d'haver una classe, però n'hi pot haver més.
  - A continuació de la declaració de la classe hi ha d'haver el símbol `{`, que indica l'inici del contingut de la classe i que s'ha de correspondre amb un símbol `}` en la finalització del contingut de la classe. Hi ha programadors que situen aquest símbol en la mateixa línia de la declaració i d'altres que el situen en la línia següent. Aquesta darrera possibilitat potser permet fer un seguiment visual millor del contingut de la classe i utilitzem diferents nivells de sagnat en els continguts de la classe. En el nostre programa, el símbol `{` d'inici de la classe es troba en la sisena línia, i el símbol `}` de finalització de la classe es troba en l'onzena línia.
- La setena línia, `public static void main (String args[])`, conté el punt d'inici d'execució del programa.

Perquè una classe inclosa en un arxiu `.java` es pugui cridar com a programa (i no ser únicament una classe que puguin cridar altres classes) ha de contenir un mètode anomenat exactament com indica la setena línia o, si no, l'interpret Java en rebutjarà l'execució.

Vegem una breu introducció sobre els diferents elements que apareixen en aquesta setena línia.

- El mot `main` indica que és el mètode per on començarà l'execució si la classe es crida com a programa.
- El mot `public` permet que el mètode sigui executat des de l'exterior de la classe, com quan la classe vol ser cridada com a programa des d'una consola del sistema.
- El mot `static` permet que el mètode es cridi de manera genèrica, sense haver d'actuar sobre cap objecte concret de la classe a què pertany el mètode.
- El mot `void` indica que el mètode `main` no retornarà cap resultat.
- L'únic argument `args` que té la funció `main` és una taula de cadenes de caràcters (`String`) en què es recullen els paràmetres que puguin acompanyar la crida del programa i en el mateix ordre emprat en la crida.

#### Tots els llenguatges de programació...

...tenen alguna funció especial per indicar per on ha de començar l'execució d'un programa. En Java, com en els llenguatges C/C++, es tracta d'un mètode anomenat `main()`.

#### Els mètodes d'una classe

En programació orientada a objectes, els mètodes d'una classe es criden sobre un objecte de la classe, exceptuant els mètodes estàtics, que es criden de manera genèrica sense incidir sobre cap objecte concret de la classe.

- A continuació de la declaració de qualsevol mètode hi ha d'haver el símbol `{` que indica l'inici del contingut del mètode i que s'haurà de correspondre amb un símbol `}` en la finalització del contingut del mètode. Hi ha programadors que situen aquest símbol en la mateixa línia de la declaració i d'altres que el situen en la línia següent. Aquesta darrera possibilitat potser permet fer un seguiment visual millor del contingut de la classe i utilitzem diferents nivells de sagnat en els continguts de la classe. En el nostre programa, el símbol `{` d'inici del mètode `main` es troba en la vuitena línia, i el símbol `}` de finalització del mètode es troba en la desena línia.
- La novena línia, `System.out.println("Hola Món");`, correspon a una instrucció de programa i, com a tal, finalitza amb el símbol `;`. En Java, totes les instruccions de programa finalitzen amb el símbol `;`.

Aquesta línia està cridant el mètode `println` que efectua la impressió del text "Hola Món" passat per paràmetre, sobre el canal estàndard de sortida, anomenat `out`, que resideix en la classe `System` proporcionada per l'entorn Java.

### 1.4.2. La fase de compilació

Recordem que el compilador del llenguatge Java és l'eina `javac`, que admet diverses opcions d'execució.

Per cada classe existent dins el fitxer `.java` el compilador de Java generarà un fitxer `.class` amb el nom de la classe. Així, en el cas que ens ocupa, en compilar el fitxer `HolaMon.java` obtindrem un fitxer anomenat `HolaMon.class`.

A continuació podem veure el procés de compilació del fitxer acompanyat del contingut del directori abans i després de la compilació:

```
E:>dir
04/02/2009  07:53    <DIR>        .
04/02/2009  07:53    <DIR>        ..
04/02/2009  07:53                240 HolaMon.java

E:\>javac HolaMon.java

E:\>dir
04/02/2009  07:53    <DIR>        .
04/02/2009  07:53    <DIR>        ..
04/02/2009  07:53                417 HolaMon.class
04/02/2009  07:53                240 HolaMon.java
```

Veiem que la instrucció per compilar el programa és, simplement:

```
javac HolaMon.java
```

Per compilar un arxiu `.java` cal indicar-ne obligatòriament l'extensió.

La seva execució, si és correcta, no dóna cap tipus de missatge. En cas d'errors, apareixeria una llista dels errors amb indicacions sobre la tipologia de l'error i la línia en què es troben.

L'arxiu que estem compilant s'anomena `HolaMon.java` i coincideix (deixant de banda l'extensió `.java`) amb el nom de l'única classe declarada en el seu interior. D'altra banda, hem comentat que un fitxer `.java` pot contenir diverses classes. És clar que el fitxer només pot tenir un nom. Hi ha alguna norma i/o obligació respecte als noms dels fitxers `.java`?

En principi, el nom d'un fitxer `.java` pot ser qualsevol, evitant els símbols especials (accents, caràcters idiomàtics...), però en desenvolupar aplicacions Java es treballa amb molts fitxers `.java` contenidors de classes i, per poder accedir des d'una classe a classes declarades en altres fitxers, ens veurem amb la necessitat de declarar com a públiques les classes a les quals volem accedir. En aquest cas apareix una limitació important: un fitxer `.java` només pot contenir una classe pública, i el nom del fitxer ha de coincidir obligatòriament amb el nom de la classe pública.

Per declarar una classe Java com a pública, cal emprar el mot `public` davant la seva declaració:

```
public class HolaMon
```

Si al davant de la declaració de la classe `HolaMon` afegim el mot `public` i enregistrem el fitxer `.java` amb un nom diferent a `HolaMon`, per exemple, `HolaMonPublic`, veiem l'error que es produeix en intentar-ne la compilació:

```
E:\>javac HolaMonPublic.java
HolaMonPublic.java:6: class HolaMon is public, should be declared
in a file named HolaMon.java
1 error
```

També cal tenir en compte que el llenguatge Java és sensible a majúscules/minúscules i, per tant, el nom de la classe pública en l'interior del seu arxiu `.java` ha de ser exactament igual que el nom de l'arxiu `.java`.

### 1.4.3. La fase d'execució

Recordem que l'interpret del llenguatge Java és l'eina `java`, que admet diverses opcions d'execució.

Una vegada coneguts els diversos elements que formen el programa i generat el seu *bytecode* (arxiu `.class` obtingut en la fase de compilació) podem procedir a la seva execució. La manera més fàcil d'executar-lo és cridant l'interpret `java` acompanyat del nom de l'arxiu `.class` corresponent (sense indicar-ne l'extensió) des d'una consola de sistema:

Per executar programes Java només necessitem disposar dels arxius `.class` generats en la fase de compilació.

```
E:\>java HolaMon
Hola M³n
```

Veiem que l'execució té lloc però advertim que els caràcters especials com els accents no es visualitzen correctament perquè la consola DOS treballa amb el codi de pàgina OEM. De fet, si des de la consola DOS visualitzem amb l'ordre `type` el contingut del font `HolaMon.java` que hem editat amb un editor del Windows (com el Notepad++) advertirem el mateix problema en el codi font:

---

```
G:\>type HolaMon.java
/* Fitxer: HolaMon.java
   Descripció: Programa que visualitza el missatge "Hola Món" per la consola
   Autor: Isidre Guixó
*/
class HolaMon
{
    public static void main (String args[])
    {
        System.out.println ("Hola Món");
    }
}
```

---

Això no passaria si haguéssim emprat un editor del DOS (com el programa *edit* del DOS encara incorporat en les versions actuals del Windows), però això no ens ha de passar pel cap, ja que provocariem el funcionament anòmal en totes les visualitzacions de dades en entorns gràfics. La solució és indicar, en el moment d'execució, que l'interpret utilitzi un codi de pàgina diferent:

---

```
E:\>java -Dfile.encoding=cp850 HolaMon
Hola Món
```

---

Ara sí que la visualització ha estat correcta per la consola de sistema.

Per acabar, cal comentar quelcom important referent a la ubicació dels arxius `.class` i el punt des d'on s'executa l'interpret `java`. L'interpret `java` cerca les classes en el camí indicat pel següent:

- L'opció `-cp` o `-classpath` de la línia de crida de l'interpret.
- Si no se li ha indicat l'opció `-cp` o `-classpath`, cerca la variable d'entorn `CLASSPATH` definida en el sistema operatiu.
- Si tampoc no troba definida la variable `CLASSPATH`, cerca les classes en el directori des d'on es crida l'interpret.

## 1.5. Components bàsics del llenguatge

Els elements bàsics a tenir presents en el desenvolupament de programes en llenguatge Java per a un programador coneixedor de la programació estructurada i modular són els següents: comentaris, identificadors, literals, separadors, operadors, tipus de dades, variables i constants.

## 1) Comentaris. En Java disposem de tres tipus de comentaris:

- **Comentaris a l'estil del llenguatge C**, també anomenats *comentaris multilínia*, ja que poden abraçar diverses línies i comencen amb el símbol `/*` i finalitzen amb el símbol `*/`.
- **Comentaris a l'estil del llenguatge C++**, que s'indiquen amb el símbol `//` i comprenen tot el text que segueix el símbol i fins al final de la línia. Abracen, com a màxim, una línia de text.
- **Comentaris especials de Javadoc**, que poden abraçar diverses línies i comencen amb el símbol `/**`, finalitzen amb el símbol `*/` i són utilitzats per l'eina *Javadoc* per ajudar a documentar les aplicacions Java.

### Javadoc

L'eina *Javadoc* és una eina que permet generar la documentació html per a les classes desenvolupades, en el mateix format que la documentació oficial de les classes existents en el llenguatge Java facilitades per Sun Microsystems.

## 2) Identificadors. Els identificadors, en qualsevol llenguatge, són els noms que s'utilitzen per anomenar variables, funcions, classes, mètodes, objectes... En definitiva, qualsevol cosa que el programador necessiti identificar.

En Java, els identificadors comencen per una lletra, un subratllat (`_`) o un símbol de dòlar (`$`). Els caràcters següents poden ser lletres o dígitos. Es distingeixen les majúscules de les minúscules i no hi ha longitud màxima.

El codi font de Java utilitza l'estàndard Unicode de 16 bits que suporta text escrit en diferents llenguatges humans, fet que permet utilitzar en els programes Java diversos alfabetes com el japonès, el grec, el rus o l'hebreu.

Els noms dels identificadors poden contenir paraules reservades del llenguatge, però no poden coincidir amb cap d'elles. Per exemple, `superMan` és un identificador vàlid, però `super` no, ja que `super` és una paraula reservada del llenguatge. Com a paraules reservades del llenguatge Java cal distingir-ne tres tipus: les paraules clau (referides en la taula 2), els literals booleans (`true` i `false`) i el literal nul (`null`).



Podeu accedir les especificacions de Java consultant la secció "Adreces d'interès" del web.

Taula 2. Paraules clau de Java segons la darrera especificació (gener del 2005)

<code>abstract</code>	<code>continue</code>	<code>for</code>	<code>new</code>	<code>switch</code>
<code>assert</code>	<code>default</code>	<code>if</code>	<code>package</code>	<code>synchronized</code>
<code>boolean</code>	<code>do</code>	<code>goto</code>	<code>private</code>	<code>this</code>
<code>break</code>	<code>double</code>	<code>implements</code>	<code>protected</code>	<code>throw</code>
<code>byte</code>	<code>else</code>	<code>import</code>	<code>public</code>	<code>throws</code>
<code>case</code>	<code>enum</code>	<code>instanceof</code>	<code>return</code>	<code>transient</code>
<code>catch</code>	<code>extends</code>	<code>int</code>	<code>short</code>	<code>try</code>
<code>char</code>	<code>final</code>	<code>interface</code>	<code>static</code>	<code>void</code>
<code>class</code>	<code>finally</code>	<code>long</code>	<code>strictfp</code>	<code>volatile</code>
<code>const</code>	<code>float</code>	<code>native</code>	<code>super</code>	<code>while</code>



Les paraules clau `const` i `goto` estan reservades però actualment no tenen cap utilització.

Els identificadors que contenen el signe \$ no són molt habituals i, per tant, és millor no utilitzar-los, excepte si hi ha una raó de pes per incloure el símbol en el nom de l'identificador.

En la comunitat de desenvolupadors en Java, hi ha un cert conveni a l'hora d'identificar classes, variables, constants, objectes i mètodes:

- La inicial del nom d'una classe seria amb majúscula i la resta amb minúscules.
- Els noms de variables, objectes i mètodes, serien amb minúscules.
- En el cas de noms formats per més d'una paraula, no s'utilitzaria el subratllat per diferenciar les paraules, sinó que les paraules s'ajuntarien i es posaria amb majúscula la inicial de cada paraula (`superMan`, `afegirPerInici`, `hiHaEstoc...`).
- Els noms de constants s'escriurien amb majúscules, separant cada paraula amb el símbol de subratllat `_` (`MAXIMA_GRANDARIA`, `TOP`, ...).

**3) Literals.** Un valor constant en Java es crea utilitzant-ne una representació literal. Java utilitza sis tipus de literals: enters, reals en punt flotant, booleans, caràcters, cadenes i valor nul, que es poden posar en qualsevol lloc del codi font de Java. Cadascun d'aquests tipus de literal està associat a un tipus de dada present en el llenguatge Java i es comporta segons estableix el tipus de dada.

Ens cal saber com representar els diversos tipus de literals possibles.

- **Literals enters.** Les constants enters es poden representar amb notació decimal (seqüència de dígitos que no pot començar amb el dígit zero), octal (seqüència de dígitos que comença amb el dígit zero) o hexadecimal (seqüència de dígitos i caràcters A, B, C, D, E i F començada per `0x`).

Les constants enters són valors enters de tipus `int` (enters de 32 bits en complement a 2) excepte si la seva representació finalitza amb la lletra `L` (majúscula o minúscula), fet que indica valor enter de tipus `long` (enter de 64 bits en complement a 2). No és recomanable la utilització de la lletra `L` minúscula, ja que es pot confondre amb el dígit `u` (1).

La taula 3 mostra alguns exemples de representacions de literals enters.

#### Tipus de dades enters en Java

El llenguatge Java incorpora quatre tipus de dades enters:

- `byte`, de 8 bits
- `short`, de 16 bits
- `int`, de 32 bits
- `long`, de 64 bits

Taula 3. Exemples de literals enters

Representació	Comentari
4235	Correspon al valor <code>int</code> positiu 4.235.
-4235	Correspon al valor <code>int</code> negatiu -4.235.
123412341234	Com que no finalitza amb la lletra <code>L</code> (majúscula o minúscula), el compilador la interpreta com un valor <code>int</code> i, en sobrepassar el màxim valor permès per a un valor <code>int</code> , el compilador dóna un error similar a <code>integer number too large</code> .
123412341234L	Correspon al valor <code>long</code> positiu 123.412.341.234.
-123412341234L	Correspon al valor <code>long</code> negatiu -123.412.341.234.
04235	Representació octal del valor <code>int</code> positiu 2.205.
-04235	Representació octal del valor <code>int</code> negatiu -2.205.
0123412341234L	Representació octal del valor <code>long</code> positiu 11.209.917.084.
-0123412341234L	Representació octal del valor <code>long</code> negatiu -11.209.917.084.
0x4235	Representació hexadecimal del valor <code>int</code> positiu 16.949.
-0x4235	Representació hexadecimal del valor <code>int</code> negatiu -16.949.
0x123412341L	Representació hexadecimal del valor <code>long</code> positiu 4.886.438.721
-0x123412341L	Representació hexadecimal del valor <code>long</code> negatiu -4.886.438.721.

- **Literals reals.** Les constants reals es poden representar amb una seqüència de dígits que ha d'incloure un punt decimal o una part exponencial (indicada amb la lletra `E` o `e`) o ha de finalitzar amb la lletra `F` o `f` (per a tipus `float`) o amb la lletra `D` o `d` (per a tipus `double`). La finalització sense lletra provoca que el compilador Java interpreti la constant real (si conté el punt decimal o la part exponencial) com a una constant `double`.

La taula 4 mostra alguns exemples de representacions de literals reals.

Taula 4. Exemples de literals reals

Representació	Comentari
4.235	Correspon al valor <code>double</code> positiu 4,235.
4.235d	Correspon al valor <code>double</code> positiu 4,235 i la lletra <code>d</code> és redundant.
-4.235	Correspon al valor <code>double</code> negatiu -4,235.
4.235f	Correspon al valor <code>float</code> positiu 4,235.
-4.235f	Correspon al valor <code>float</code> negatiu -4,235.
4.235E-2	Correspon al valor <code>double</code> positiu 0,04235.
4.235E-2d	Correspon al valor <code>double</code> positiu 0,04235 i la lletra <code>d</code> és redundant.
-4.235E-2	Correspon al valor <code>double</code> negatiu -0,04235.
4.235E-2f	Correspon al valor <code>float</code> positiu 0,04235.
-4.235E-2f	Correspon al valor <code>float</code> negatiu -0,04235.

#### Tipus de dades reals en Java

El llenguatge Java incorpora dos tipus de dades reals:

- `float`, de 32 bits
- `double`, de 64 bits

- **Literals booleans.** Hi ha dos literals booleans, representats per les paraules reservades `true` i `false`.
- **Literals caràcter.** Les constants caràcter es representen per un caràcter simple o una seqüència d'escapament situats entre cometes simples.

Un caràcter simple pot ser qualsevol caràcter exceptuant el caràcter `\` i la cometa simple: `'a'`, `'b'`, `'c'...`

Una seqüència d'escapament sempre va precedida del caràcter `\` i les seqüències permeses són:

- `\b` per indicar un retrocés (*backspace*)
- `\t` per indicar un salt de tabulador
- `\n` per indicar un salt de línia (*linefeed*)
- `\f` per indicar un salt de pàgina (*formfeed*)
- `\r` per indicar un retorn (*carriage return*)
- `\\` per indicar el caràcter `\`
- `\'` per indicar el caràcter cometa simple
- `\"` per indicar el caràcter cometa doble
- `\u####` per indicar un valor Unicode

- **Literals cadena.** Les constants cadena són valors que poden prendre els objectes de la classe `String` proporcionats pel llenguatge Java i es representen entre cometes dobles i en el seu interior poden incorporar qualsevol seqüència d'escapament.

Exemples de constants cadena vàlides són:

```
"La lluna i la pruna"  
"L'hospital de la vall"  
"L'error \"integer number too large\" indica..."
```

- **Literal nul.** El literal nul és la paraula reservada `null` que s'utilitza en casos com els següents:
  - Per inicialitzar una referència a una classe que no faci referència a cap objecte de la classe.
  - Per preguntar si una referència a una classe fa referència a un objecte de la classe.

**4) Separadors.** La taula 5 mostra els separadors permesos en el llenguatge Java.

Taula 5. Símbols separadors permesos en el llenguatge Java

Separador	Comentari
()	Parèntesis, emprats per: <ul style="list-style-type: none"> <li>• contenir llistes de paràmetre en la definició i crida a mètodes</li> <li>• definir precedència en expressions</li> <li>• contenir expressions en les sentències de control de flux</li> <li>• rodejar les conversions de tipus</li> </ul>
{ }	Claus, emprades per: <ul style="list-style-type: none"> <li>• contenir els valors de taules inicialitzades automàticament</li> <li>• definir blocs de codi, classes, mètodes i àmbits locals</li> </ul>
[ ]	Claudàtors, emprats per: <ul style="list-style-type: none"> <li>• declarar variables de tipus taula</li> <li>• fer referència a valors d'una taula</li> </ul>
;	Punt i coma, per separar sentències
,	Coma, emprada per: <ul style="list-style-type: none"> <li>• separar identificadors consecutius en una declaració de variables</li> <li>• encadenar sentències dins una sentència <code>for</code></li> </ul>
.	Punt, emprat per: <ul style="list-style-type: none"> <li>• separar els noms de paquets dels noms dels subpaquets i classes</li> <li>• separar un nom de variable o mètode d'una referència</li> </ul>

**5) Operadors i expressions.** Els operadors del llenguatge Java són molt similars, en estil i funcionament, als operadors dels llenguatges C/C++ i, com en aquests llenguatges, n'hi ha de binaris i d'unaris.

La taula 6 mostra els operadors del llenguatge Java per ordre de precedència, de manera que tots els operadors situats en una mateixa fila són més prioritaris que els de les files que es troben per sota i, dins una mateixa fila, la prioritats disminueix d'esquerra cap a dreta. La descripció detallada de cada operador és disponible en l'especificació del llenguatge Java.

Els programes en Java, com en la resta de llenguatges, es componen de sentències, que al seu torn estan formades a partir d'expressions.

Taula 6. Operadors del llenguatge Java per ordre de precedència

Precedència	+						-					
+	.	[ ]	()									
	++	--										
	!	~	instanceof									
	*	/	%									
	+	-										
	<<	>>	>>>									
	<	>	<=		>=		==		!=			
	&	^										
	&&											
	? :											
	-	=	op=	,								

L'operador `op=` fa referència a la combinació de qualsevol dels operadors `*`, `/`, `%`, `+`, `-`, `<<`, `>>`, `>>>`, `&`, `^` i `|` seguits de l'operador `=`.



Podeu accedir les especificacions de Java consultant la secció "Adreces d'interès" del web.

Una expressió és una combinació determinada d'operadors i operands que s'avaluen per obtenir un resultat particular. En el llenguatge Java, els operands poden ser variables, constants o crides a mètodes definits en les classes cridades en el programa.

Per confeccionar expressions vàlides ens cal conèixer el funcionament dels operadors, llur precedència i els resultats proporcionats pels mètodes cridats dins les expressions.

El llenguatge Java, a diferència del C++, no permet la sobrecàrrega d'operadors i, per tant, no és possible definir l'actuació d'un operador sobre objectes d'una classe definida pel programador. **!!**

Un cas interessant, excepció de l'afirmació anterior, és l'operador + que es pot emprar per efectuar una suma de valors numèrics i també per concatenar cadenes. És a dir, el llenguatge Java el té sobrecarregat internament, de manera que si un dels dos operands sobre els que actua l'operador + és una cadena, el compilador Java interpreta que ha d'activar l'operador + per concatenar cadenes i concatena la cadena amb l'altre operand, que en cas de no ser cadena, és convertit automàticament en una cadena de caràcters per poder efectuar la concatenació.

#### Exemple d'utilització de la sobrecàrrega de l'operador + en el llenguatge Java

```
/* Fitxer: UsosOperadorSuma.java
   Descripció: Programa que mostra la utilització de l'operador + com a
               suma de valors numèrics i com a concatenació de cadenes.
   Autor: Isidre Guixà
*/
class UsosOperadorSuma
{
    public static void main (String args[])
    {
        int x=10;
        int y=20;
        System.out.println ("La suma de " + x + " i " + y + " és " + (x+y));
    }
}
```

En la darrera línia del programa anterior es pot veure com l'operador + actua concatenant la cadena "La suma de " amb una representació cadena del valor enter 10 emmagatzemat per la variable **x**. La cadena resultant ("La suma de 10") es concatena amb la cadena " i " i de la mateixa manera es produeix una concatenació successiva amb el valor enter 20 emmagatzemat per la variable **y**, la cadena " és " i el valor 30 resultat de la suma de les variables **x** i **y**, en què es veu que l'operador + ha actuat com a suma dels valors numèrics emmagatzemats per les variables **x** i **y**. L'execució del programa dona:

```
E:\>java -Dfile.encoding=cp850 UsosOperadorSuma
La suma de 10 i 20 és 30
```



Trobareu el fitxer  
UsosOperadorSuma.java en la  
secció "Recursos de contingut" del  
web.

Fixem-nos en la importància de situar l'expressió  $x+y$  entre parèntesis. Si no ho haguéssim fet, s'hauria continuat avaluant l'operador  $+$  com a concatenació amb la cadena de l'esquerra i hauriem obtingut:

---

```
E:\>java -Dfile.encoding=cp850 UsosOperadorSuma
La suma de 10 i 20 és 1020
```

---

En el procés descrit, el compilador de Java és capaç d'obtenir una representació cadena d'un valor que no és cadena, i això és capaç de fer-ho amb tots els objectes, tant si són de classes facilitades pel mateix llenguatge Java o de classes desenvolupades per nosaltres. En aquest darrer cas, si volem que la representació cadena sigui coherent, cal haver donat instruccions sobre com es pot generar la representació cadena.

### Exemple de sobrecàrrega d'operadors en els llenguatges C++ i Java

Suposem que el programador està implementant una classe anomenada `Data` per gestionar dates. En aquest cas és molt típic voler dotar aquesta classe de mètodes que permetin obtenir una data futura/passada a partir de la suma/resta d'un nombre determinat de dies en una data concreta o calcular el nombre de dies que hi ha entre dues dates i, per tant, seria molt útil poder emprar els símbols  $+$  i  $-$  per a aquest propòsit.

És a dir, estaria molt bé poder utilitzar els operadors  $+$  i  $-$  per fer càlculs com:

---

```
Data d1 = new Data(1,1,1980); /* d1 conté 1-1-1980 */
Data d2 = null;
int x;

d2 = d1+40;           // d2 passaria a contenir 10-2-1980
d2 = d1-40;           // d2 passaria a contenir 22-11-1979
x = d1-d2;            // x passaria a contenir 40, doncs és el nombre de
                      // dies que hi ha entre d1 i d2.
```

---

El comportament anterior dels operadors  $+$  i  $-$  es podria definir en el llenguatge C++ gràcies a la sobrecàrrega d'operadors, però en el llenguatge Java no és factible i, per tant, cal definir-lo mitjançant mètodes com:

- **`sumaDies(int)`**, mètode que aplicat a un objecte `Data` retorni una nova data futura/passada a partir de sumar el nombre de dies indicat en el paràmetre (positiu/negatiu).
- **`intervalDies(Data)`**, mètode que aplicat a un objecte `Data` retorni el nombre de dies que el separen de l'objecte `Data` indicat en el paràmetre.

Així, el codi de més amunt, caldria refer-lo així:

---

```
Data d1 = new Data(1,1,1980); /* d1 conté 1-1-1980 */
Data d2 = null;
int x;

d2 = d1.sumaDies(40); // d2 passaria a contenir 10-2-1980
d2 = d1.sumaDies(-40); // d2 passaria a contenir 22-11-1979
x = d1.intervalDies(d2); // x passaria a contenir 40
```

---

**6) Tipus de dades.** Els llenguatges de programació proporcionen tipus de dades primitius (també anomenats *simples* o *elementals*) i tipus de dades complexos, construïts a partir dels tipus de dades primitius. Així mateix, proporcionen mecanismes perquè els programadors puguin definir els seus propis tipus de dades complexos.

El llenguatge Java proporciona vuit tipus primitius, presentats en la taula 7, i que s'agrupen en quatre categories: lògics, caràcters, enters i reals.

Taula 7. Tipus de dades primitius del llenguatge Java

Categoria	Tipus	Valors possibles	Grandària en bits	Observacions
Lògics	boolean	Els literals booleans <code>true</code> i <code>false</code> .		
Caràcters	char	Qualsevol literal caràcter, des de <code>\u0000</code> fins a <code>\uffff</code>		
Enters	byte	$-2^7 \dots 2^7-1$	8	A diferència dels llenguatges C/C++, no hi ha tipus enters sense signe. La implementació sempre és en complement a 2 en qualsevol plataforma.
	short	$-2^{15} \dots 2^{15}-1$	16	
	int	$-2^{31} \dots 2^{31}-1$	32	
	long	$-2^{63} \dots 2^{63}-1$	64	
Reals	float		32	La implementació segueix la norma IEEE 754 i és idèntica en totes les plataformes.
	double		64	

I respecte als tipus de dades complexos, en el llenguatge Java, com a llenguatge pur orientat a objectes que és, no hi ha altre mecanisme per definir tipus complexos que la definició de classes.

**7) Variables i constants.** El llenguatge Java permet treballar amb dos tipus de variables i de constants en funció de si han de servir per emmagatzemar valors de tipus primitius o per fer referència a objectes de classes.

- **Variables i constants de tipus primitius.** Donat un tipus primitiu de nom `T`, per declarar-ne variables i/o constants cal seguir la sintaxi següent, en què es veu que en el moment de la declaració es pot donar valor a la variable i/o constant però no és obligatori en cap cas:

---

```
T nomVariable [= <valor_del_tipus_T>];
final T NOM_CONSTANT [= <valor_del_tipus_T>];
```

---

L'apartat `<valor_del_tipus_T>` pot ser un literal corresponent al tipus `T` o una expressió que té com a resultat un valor del tipus `T` o una variable del tipus `T` inicialitzada.

La paraula reservada `final` al davant del tipus indica la definició d'una constant, i la seva absència indica la definició d'una variable. Cal tenir en compte que:

- No és obligatori, a diferència d'altres llenguatges, donar valor a les constants en el moment de la declaració.
- Abans d'utilitzar per primera vegada constants i/o variables, cal que s'hagin inicialitzat amb l'operador d'assignació (`=`).

- Una constant només pot prendre valor una vegada. El compilador interceptarà qualsevol intent de canviar-ne el valor una vegada inicialitzada.

#### Exemple de declaracions de variables i constants de tipus primitius

```

/* Fitxer: VariablesConstantsTipusPrimitius.java
   Descripció: Exemples de declaracions de variables i constants de tipus primitius
   Autor: Isidre Guixà
*/
class VariablesConstantsTipusPrimitius
{
    public static void main (String args[])
    {
        int i1=10, i2;
        char c1, c2='Z';
        boolean b1=true, b2=false;
        final int NRE_MESOS;
        long n = 123412341234L;
        float f = 42314.4231F;
        double d = 4231442142.4231;
        i2=20;
        c1='A';
        NRE_MESOS=12;    // Una constant es pot inicialitzar en un moment
                        // posterior a la seva declaració
//    NRE_MESOS=24;    No podem donar valor a una constant inicialitzada
        System.out.println ("i1 = " + i1);
        System.out.println ("i2 = " + i2);
        System.out.println ("c1 = " + c1);
        System.out.println ("c2 = " + c2);
        System.out.println ("b1 = " + b1);
        System.out.println ("b2 = " + b2);
        System.out.println ("NRE_MESOS = " + NRE_MESOS);
        System.out.println ("n = " + n);
        System.out.println ("f = " + f);
        System.out.println ("d = " + d);
    }
}

```

En una assignació de valor a una variable es pot perdre informació si els tipus són diferents i, com a mesura preventiva, el compilador requereix del programador la confirmació de l'assignació per mitjà d'una conversió. Així, per exemple, davant el codi següent:

```

long valorGran = 99L;
int valorPetit = valorGran;

```

El compilador ens donaria l'error següent:

```

possible loss of precision
found: long
required: int
                int valorPetit = valorGran;
                ^
1 error

```



Trobareu el fitxer  
VariablesConstantsTipusPrimitius.java en la secció  
"Recursos de contingut" del web.




El programador que ha escrit el codi anterior sap perfectament que no es produeix cap pèrdua de precisió, ja que el valor enter 99 es pot representar en una variable de tipus `int`. Però el compilador exigeix al programador que li marqui la conversió, de manera que cal escriure:

---

```
long valorGran = 99L;
int valorPetit = (int)valorGran;
```

---

En la darrera línia de codi, el programador ha indicat que el valor de la variable `valorGran` de tipus `long` es converteixi a un valor de tipus `int` per tal de poder-la assignar a la variable `valorPetit` de tipus `int`.

Per efectuar una conversió, el tipus al qual es vol fer la conversió cal indicar-lo entre parèntesis i com a prefix de l'expressió a convertir. No és necessari posar tota l'expressió a convertir entre parèntesis, però és aconsellable per evitar problemes. 

Els tipus de dades es converteixen automàticament quan no hi ha possibilitat de pèrdua d'informació en la conversió. En general, es pot prendre una expressió com una assignació compatible, si el tipus de variable que s'utilitza és, com a mínim, tan gran (en nombre de bits) com el tipus de l'expressió.

Cal anar en compte amb les operacions entre dades de tipus primitius numèrics enters, ja que el resultat de les operacions entre elles sempre és, com a mínim, `int` i sempre coincideix amb el tipus més gran dels dos operands. Per tant, com s'exemplifica en la taula 8, aquestes operacions poden provocar desbordament o pèrdua de precisió.

Taula 8. Exemplificació de problemes que poden sorgir en operacions entre tipus de dades numèrics primitius.

<code>a = 1</code>	<code>b = 2</code>	<code>c = a op b</code>	Comportament del compilador	Solució
<code>short a</code>	<code>short b</code>	<code>short c</code>	Error, ja que el resultat de l'operació és <code>int</code> (malgrat tots dos operands siguin <code>short</code> ) i la variable a la qual es vol assignar és <code>short</code> .	<code>c = (short) (a op b);</code>
<code>int a</code>	<code>int b</code>	<code>int c</code>	Cap problema.	
<code>short a</code>	<code>long b</code>	<code>int c</code>	Error, perquè el resultat de l'operació és <code>long</code> , ja que és el tipus de dada més gran d'entre els dos operands, i la variable a la qual es vol assignar és <code>int</code> .	<code>c = (int) (a op b);</code>

També cal saber que el llenguatge Java manté el mateix comportament del llenguatge C respecte a les operacions amb operands enters quan un resultat sobrepassa la capacitat màxima del tipus resultant. En molts llenguatges es produeix un error en temps d'execució que provoca la fi immediata del programa. En argot informàtic es diu que “el programa ha avortat”. Quan això passa, el programa acostuma a informar d'un error d'*overflow* (valor per damunt de la capacitat permesa, tant en el vessant positiu com en el negatiu).

Els nostres amics Java i C no actuen així. No provoquen cap error i, per tant, no avorten l'execució del programa. Llavors, com actuen? És difícil d'explicar amb paraules. Potser un exemple és el més convenient.

### Exemple de comportament cíclic dels tipus primitius enters en superar la capacitat

L'exemple següent ens mostra un problema que podem trobar en utilitzar els tipus primitius enters en cas d'efectuar operacions que superin la capacitat del tipus de dada corresponent al resultat.



Trobareu el fitxer `ComportamentCiclicValorsEnters.java` en la secció "Recursos de contingut" del web.

```
/* Fitxer: ComportamentCiclicValorsEnters.java
   Descripció: Exemple del comportament cíclic dels tipus primitius quan una operació
               supera el màxim valor permès
   Autor: Isidre Guixà
*/

class ComportamentCiclicValorsEnters
{
    public static void main(String args[])
    {
        int i = 60000;
        System.out.println("i = " + i);
        System.out.println("i * i = " + (i*i));
    }
}
```

Suposem que la sortida esperada és:

```
G:\>java ComportamentCiclicValorsEnters
i = 60000
i * i = 3600000000
```

Doncs, ens sap greu decebre-us. La sortida és:

```
G:\>java ComportamentCiclicValorsEnters
i = 60000
i * i = -694967296
```

Segurament us preguntareu d'on surt el valor -694.967.296?

Imagineu-vos el conjunt de valors representables pel tipus `int` (-2.147.483.648 ... 2.147.483.647) com un conjunt cíclic: així, de la mateixa manera que després del 60.000 ve el 60.001, succeeix que després del 2.147.483.647 ve el -2.147.483.648 i comencem a pujar en direcció al zero. Per tant, com que el resultat matemàtic de  $60.000 * 60.000$  (3.600.000.000) excedeix el valor màxim pel tipus `int`, 2.147.483.647, en 1.452.516.353 unitats, tornant a començar pel valor mínim, que és -2.147.483.648, resulta que arribem a -694.967.296, resultat que proporciona la màquina virtual.

Aquest funcionament té lloc en tots els tipus de dades numèriques enteres. En conclusió, cal tenir cura de no sobrepassar les capacitats màximes permeses en els tipus de dades resultants de les operacions amb valors enters. (❗)

- **Variables i constants per fer referència a objectes de classes.** Donada una classe de nom `C`, per declarar-ne variables i/o constants cal seguir la sintaxi següent, en la qual es veu que en el moment de la declaració es pot donar valor a la variable i/o constant però en cap cas no és obligatori.

```
C nomVariable [= <referència>];
final C NOM_CONSTANT [= <referència>];
```

L'apartat <referència> ha de ser obligatòriament una referència a un objecte de la classe `C` o el valor `null`. En particular, es pot utilitzar l'operador `new` acompanyat d'un mètode constructor de la classe per construir l'objecte en el moment d'efectuar la declaració de la variable i/o constant.

La paraula reservada `final` a l'esquerra del nom de la classe indica la definició d'una constant, i la seva absència indica la definició d'una variable. Cal tenir en compte que:

- No és obligatori, a diferència d'altres llenguatges, donar valor a les constants en el moment de la declaració.
- Abans d'utilitzar per primera vegada constants i/o variables, cal que s'hagin inicialitzat amb l'operador d'assignació (=).
- Una constant només pot fer referència a un objecte, de manera que quan s'hagi inicialitzat, ja mai no podrà ser utilitzada per fer referència a un altre objecte.

#### Exemple de declaracions de variables i constants de referències

```
/* Fitxer: VariablesConstantsReferencias.java
   Descripció: Exemples de declaracions de variables i constants de referències
   Autor: Isidre Guixà
*/
class X
{
}

class VariablesConstantsReferencias
{
    public static void main (String args[])
    {
        X obj1 = null;
        X obj2 = new X();
        X obj3 = obj2;
        final X obj4;
        obj4 = new X(); // Una constant es pot inicialitzar en un moment
                        // posterior a la seva declaració
        // obj4 = obj2; // No podem donar valor a una constant inicialitzada
    }
}
```

Hem creat les variables `obj1`, `obj2` i `obj3` i la constant `obj4` per fer referència a objectes de la classe `X` (creada sense cap contingut). La variable `obj1` ha estat inicialitzada amb el valor `null` (no fa referència a cap objecte). La variable `obj2` ha estat inicialitzada amb la referència a un objecte que s'ha creat en la mateixa sentència utilitzant l'operador `new`. La variable `obj3` ha estat inicialitzada amb la referència al mateix objecte a què fa referència `obj2` (per tant, `obj2` i `obj3` fan referència al mateix objecte). La constant `obj4` en el moment de la seva declaració no ha estat inicialitzada. Posteriorment ha estat inicialitzada fent referència a un objecte de nova creació.

**8) Ubicació i àmbit d'existència de les variables.** El llenguatge Java permet declarar variables en diversos llocs:

- En definir els membres d'una classe.
- En els mètodes (accions i procediments) definits dins les classes.

#### Constructor

Els constructors són els mètodes que proporcionen les classes per construir objectes. Tota classe ha de proporcionar, com a mínim, un constructor.



Trobareu el fitxer `VariablesConstantsReferencias.java` en la secció "Recursos de contingut" del web.

Les variables definides dins un mètode verifiquen el següent:

- Es poden declarar en qualsevol punt del mètode, tot i que seguint els principis de la programació estructurada i modular és aconsellable declarar totes les variables a l'inici de cada mètode i abans de qualsevol instrucció que no sigui de declaració de variables.
- Són variables locals en el mètode, que es creen en el moment en què el flux d'execució passa per la instrucció de declaració de la variable i es destrueixen quan el flux d'execució abandona el mètode. Per tant, una vegada creades, són visibles en qualsevol punt del mètode i mai no poden ser emmascarades per variables (amb el mateix nom) declarades dins de blocs de codi interns al mètode.
- Si es declaren dins un bloc de codi intern al mètode (aconsellable que sigui a l'inici del bloc), mai no poden tenir el mateix nom que una variable present en un bloc més extern; es creen en el moment en què el flux d'execució passa per la instrucció de declaració de la variable i es destrueixen quan el flux d'execució abandona definitivament el bloc.



Podeu veure com s'efectua la declaració de variables en definir els membres d'una classe en l'apartat "Definició de classe: dades i mètodes" del nucli d'activitat "Programació orientada en Java".

#### Blocs de codi

Sovint la programació en Java necessita agrupar un conjunt d'instruccions en què s'anomena un bloc de codi, emmarcant-les entre claus: { i }.



Trobareu el fitxer `AmbitExistenciaVariables.java` en la secció "Recursos de contingut" del web.

#### Exemple de l'àmbit d'existència de variables

El programa següent exemplifica diverses situacions de declaracions de variables i llur accés. Les línies de codi comentades corresponen a situacions errònies, i el comentari incorpora el motiu de l'error.

```
/* Fitxer: AmbitExistenciaVariables.java
   Descripció: Programa per comprovar l'àmbit d'existència de les variables
   Autor: Isidre Guixà
*/
class AmbitExistenciaVariables
{
    public static void main(String[] args)
    {
        int n = 10;
        System.out.println ("Punt 1. n = " + n);
        {
            // int n = 20;    No es pot crear una variable dins el bloc doncs ja
            //                existeix en un bloc superior
            n = 20;
            char c = 'A';
            System.out.println ("Punt 2. n = " + n);
            System.out.println ("Punt 2. c = " + c);
            {
                n = 30;
                c = 'B';
                float f = 23.45F;
                System.out.println ("Punt 3. n = " + n);
                System.out.println ("Punt 3. c = " + c);
                System.out.println ("Punt 3. f = " + f);
            }
            System.out.println ("Punt 4. n = " + n);
            System.out.println ("Punt 4. c = " + c);
            // System.out.println ("Punt 4. f = " + f);    La variable f que havia existit dins un
            //                                           bloc més intern, ja no existeix en haver
            //                                           sortit del bloc on havia estat declarada
        }
        System.out.println ("Punt 5. n = " + n);
        // System.out.println ("Punt 5. c = " + c);    La variable c que havia existit dins un
        //                                           bloc més intern, ja no existeix en haver
        //                                           sortit del bloc on havia estat declarada
    }
}
```

El resultat de l'execució és:

---

```
G:\>java AmbitExistenciaVariables
Punt 1. n = 10
Punt 2. n = 20
Punt 2. c = A
Punt 3. n = 30
Punt 3. c = B
Punt 3. f = 23.45
Punt 4. n = 30
Punt 4. c = B
Punt 5. n = 30
```

---

## 1.6. Sentències de control de flux

Java, com a llenguatge que compleix amb els principis de la programació estructurada, proporciona sentències condicionals i repetitives per al control de flux.

### 1.6.1. Sentències condicionals

Java proporciona dues sentències condicionals: `if/then` i `switch`.

**1) Sentència `if/then`.** La sintaxi és:

---

```
if (expressió_booleana)
{
    <conjunt_de_sentències_si_la_condició_s'avalua_com_a_cert>;
}
[ else
{
    <conjunt_de_sentències_si_la_condició_s'avalua_com_a_fals>;
} ]
```

---

Aquesta sentència provoca que l'execució del programa executi un conjunt o un altre de sentències en funció de l'avaluació de l'expressió booleana, la qual pot ser una variable simple declarada com a `boolean` o una expressió que utilitzi operadors relacionals per generar el resultat d'una comparació. Com es veu, la clàusula `else` és opcional.

En cas que `<conjunt de sentències>` es redueixi a una única sentència, podem no tancar-la entre claus, és a dir:

---

```
if (expressió_booleana)
    <sentència_única_si_la_condició_s'avalua_com_a_cert>;
[ else
    <sentència_única_si_la_condició_s'avalua_com_a_fals>;
]
```

---

## 2) Sentència **switch**. La sintaxi és:

---

```
switch (expressió)
{
    case valor1:
        <conjunt_de_sentències>;
        [break;]
    case valor2:
        <conjunt_de_sentències>;
        [break;]
    ...
    [default:
        <conjunt_de_sentències>;]
}
```

---

La sentència `switch` permet efectuar la comparació del valor resultant d'expressió amb un nombre finit de valors indicats per les clàusules `case` segons el funcionament següent: el valor resultant d'expressió es compara amb cadascun dels valors de les clàusules `case`, de manera que en el moment en què coincideix amb algun, s'executa el codi que hi ha a continuació fins que trobi una sentència `break`; i ja no s'efectua la comparació amb els valors de la resta de clàusules `case`. Si no coincideix amb cap dels valors de les clàusules `case` i existeix la clàusula `default`, s'executa el codi que hi ha a continuació d'aquesta clàusula, i si no hi ha clàusula `default` no executa res.

L'expressió de la clàusula `switch` ha de retornar un enter de grandària igual o inferior al tipus `int`, és a dir, `byte` o `short` o `char` o `int`. Els valors constants indicats en les diverses clàusules `case` han de ser compatibles amb el tipus de l'expressió de la clàusula `switch`. Els valors de les constants de les clàusules `case` han de ser únics.

El compilador de Java inspecciona cadascun dels valors que pugui prendre l'expressió de la clàusula `switch` a partir de les clàusules `case` que es proporcionen i crea una taula eficient que utilitza per ramificar el control de flux al `case` adequat en funció del valor que prengui l'expressió. Per tant, si es necessita triar entre un gran nombre de valors, una sentència `switch` s'executarà molt més ràpid que la lògica equivalent codificada utilitzant sentències `if/else` imbricades.

La sentència `break` s'utilitza per indicar que l'execució surti de la sentència `switch` saltant a la sentència següent, de manera similar als llenguatges C/C++. A l'hora de programar, és un error habitual oblidar la sentència `break` al final del codi corresponent a una clàusula `case`, ja que el compilador no avisa d'aquestes omissions. Però de vegades pot interessar que en finalitzar el codi propi d'una clàusula `case` també es vulgui l'execució del codi de la clàusula `case` següent, situació en la qual no s'ha d'incloure la sentència `break`. En aquests casos és recomanable incloure un comentari similar a `Continua` en el lloc en què habitualment es posaria el `break`, amb vista a revisions de codi posteriors.

### 1.6.2. Sentències repetitives

Java proporciona tres sentències repetitives (`for`, `while` i `do`) que permeten l'execució repetida d'un bloc de sentències segons l'avaluació d'una condició. Les sentències `for` i `while` avaluen la condició abans d'executar el bloc repetitiu mentre que la sentència `do` avalua la condició després d'executar el bloc repetitiu. Per tant, les sentències `for` i `while` poden no executar el bloc repetitiu ni una sola vegada (en cas que la condició ja s'avalui com a falsa des del principi), mentre que la sentència `do` executa el bloc repetitiu una vegada com a mínim.

#### 1) Sentència `for`. La sintaxi és:

---

```
for ([part_ini]; [expressió_booleana]; [part_ite])
{
    <conjunt_de_sentències_si_la_condició_s'avalua_com_a_cert>;
}
```

---

Aquesta sentència repetitiva és molt similar a la mateixa sentència en els llenguatges C/C++, i en la seva execució involucra tres accions, corresponents als tres apartats que hi ha dins els parèntesis que acompanyen la clàusula `for`:

- **Part\_ini** acostuma a contenir un conjunt d'inicialitzacions de variables que han d'estar declarades i no han de ser necessàriament enteres o de càlculs inicials. Aquesta part s'executa una única vegada quan es comença l'execució de la sentència `for`.
- **Expressió\_booleana** que s'avalua després de l'execució de `part_ini` i després de cada execució de `part_ite`. Si el resultat és `cert`, s'executa el `<conjunt_de_sentències>` de l'interior del `for`. Si el resultat és `fals` es finalitza l'execució del `for`.
- **Part\_ite** acostuma a contenir el conjunt d'instruccions que fan evolucionar el(s) valor(s) de les variables perquè l'`expressió_booleana` deixi de ser certa i, per tant, s'acabi l'execució de l'estructura iterativa. S'executa després de cada execució del `<conjunt_de_sentències>`.

`Part_ini` i `part_ite` poden estar formades per diferents instruccions; en aquest cas, van separades per comes.

`Part_ite` es podria obviar i constituir part del `<conjunt_de_sentències>`, de la mateixa manera que a vegades el `<conjunt_de_sentències>` es podria obviar i situar-se com a component de `part_ite`. Però això no és lògic. Cal acostumar-se a situar les instruccions que provoquen el canvi de valor de

l'expressió\_booleana en part\_ite i el conjunt d'instruccions repetitiu en el <conjunt\_de\_sentències>.

És obligatori situar els dos “;” que separen les tres parts de la capçalera del `for` encara que alguna part no s'hi defineixi.

Els tres apartats de la capçalera del `for` són optatius, però no té gaire sentit deixar-los buits. En aquesta situació possiblement s'hauria de recórrer a un altre tipus de sentència enlloc d'utilitzar la sentència `for`.

En cas que <conjunt de sentències> es redueixi a una única sentència, podem no tancar-la entre claus, és a dir:

---

```
for ([part_ini]; [expressió_booleana]; [part_ite])
    <sentència_única_si_la_condició_s'avalua_com_a_cert>;
```

---

#### Exemple d'utilització de la sentència `for`

Volem desenvolupar un programa que ens construeixi la taula de sumar del numero dos. Una solució, utilitzant la sentència `for`, la tenim en el programa següent:

---

```
/* Fitxer: TaulaSumar2ViaFor.java
   Descripció: Programa que genera la taula de sumar del nombre 2
               utilitzant la sentència for.
   Autor: Isidre Guixà
*/
class TaulaSumar2ViaFor
{
    public static void main (String args[])
    {
        int i;
        for (i=0;i<=10;i++)
            System.out.println ("2 + " + i + " = " + (2+i));
    }
}
```

---

No cal mostrar aquí el resultat de l'execució... Ja el sabem, oi?

## 2) Sentència `while`. La sintaxi és:

---

```
while (expressió_booleana)
{
    <conjunt_de_sentències_si_la_condició_s'avalua_com_a_cert>;
}
```

---

Aquesta sentència repetitiva és molt similar a la mateixa sentència en els llenguatges C/C++. El programa, en temps d'execució, avalua expressió\_booleana. Si és avaluada com a cert, s'executa <conjunt\_de\_sentències> i es torna a repetir el procés d'avaluació



Trobareu el fitxer  
TaulaSumar2ViaFor.java en  
la secció "Recursos de contingut"  
del web.



d'expressió\_booleana. Quan és avaluada com a fals, el programa finalitza l'execució de la sentència while.

En cas que <conjunt de sentències> es redueixi a una única sentència, podem no tancar-la entre claus, és a dir:

---

```
while (expressió_booleana)
    <sentència_única_si_la_condició_s'avalua_com_a_cert>;
```

---

#### Exemple d'utilització de la sentència while

Volem desenvolupar un programa que ens construeixi la taula de sumar del número dos. Una solució, utilitzant la sentència while, la tenim en el programa següent:

---

```
/* Fitxer: TaulaSumar2ViaWhile.java
   Descripció: Programa que genera la taula de sumar del nombre 2
               utilitzant la sentència while.
   Autor: Isidre Guixà
*/
class TaulaSumar2ViaWhile
{
    public static void main (String args[])
    {
        int i=0;
        while (i<=10)
        {
            System.out.println ("2 + " + i + " = " + (2+i));
            i++;
        }
    }
}
```

---



Trobareu el fitxer  
TaulaSumar2ViaWhile.java  
en la secció "Recursos de  
contingut" del web.

### 3) Sentència do. La sintaxi és:

---

```
do
{
    <conjunt_de_sentències>;
}
while (expressió_booleana)
```

---

Aquesta sentència repetitiva és molt similar a la mateixa sentència en els llenguatges C/C++. El programa, en temps d'execució, executa una vegada <conjunt\_de\_sentències> i, posteriorment, avalua expressió\_booleana. Si és avaluada com a cert, torna a executar <conjunt\_de\_sentències> i es torna a repetir el procés d'avaluació d'expressió\_booleana. Quan és avaluada com a fals, el programa finalitza l'execució de la sentència do.

En cas que <conjunt de sentències> es redueixi a una única sentència, podem no tancar-la entre claus, és a dir:

---

```
do
    <sentència_única>;
while (expressió_booleana)
```

---

### Exemple d'utilització de la sentència do

Volem desenvolupar un programa que ens construeixi la taula de sumar del número dos. Una solució, utilitzant la sentència do, la tenim en el programa següent:

```
/* Fitxer: TaulaSumar2ViaDo.java
   Descripció: Programa que genera la taula de sumar del nombre 2
               utilitzant la sentència do.
   Autor: Isidre Guixà
*/
class TaulaSumar2ViaDo
{
    public static void main (String args[])
    {
        int i=0;
        do
        {
            System.out.println ("2 + " + i + " = " + (2+i));
            i++;
        }
        while (i<=10);
    }
}
```



Trobareu el fitxer  
TaulaSumar2ViaDo.java en la  
secció "Recursos de contingut" del  
web.

### 1.6.3. Trencament d'estructures repetitives

Les estructures repetitives consisteixen en la repetició de l'execució d'un conjunt d'instruccions sota la supervisió d'una condició lògica que s'avalua abans o després de l'execució del conjunt d'instruccions i determina la continuació o l'avortament del procés repetitiu. Aquest és el funcionament desitjable en programació, ja que assegura que la condició lògica és l'únic que governa el procés.

Alguns llenguatges, i Java no és una excepció, incorporen en les sentències repetitives la possibilitat de trencar el procés repetitiu sense necessitat d'esperar la condició lògica. No és una bona pràctica acostumar-se a fer servir aquestes possibilitats, ja que treu llegibilitat i claredat al programa, però en situacions excepcionals poden ser d'utilitat.

El llenguatge Java proporciona dues sentències per controlar el flux: `break` i `continue`.

1) La sentència `break` té la sintaxi següent:

```
break [etiqueta];
```

Aquesta sentència ja ens és familiar perquè s'utilitza en l'estructura condicional `switch` per finalitzar l'execució del codi corresponent a una opció.

La utilització de `break` sense etiqueta en un bloc repetitiu provoca la sortida immediata del bloc saltant a la sentència que segueix a l'estructura repe-

#### Llicència per trencar estructures repetitives

El programador té llicència per trencar el flux normal d'execució de les estructures iteratives quan es vol controlar alguna circumstància excepcional (una situació anòmla, per exemple) que es pot produir durant l'execució del bloc repetitiu i no es vol complicar la condició que controla el flux normal d'execució.

titiva. En cas que hi hagi imbricació d'estructures repetitives, només afecta l'estructura en què s'inclou, i si es vol que surti d'una estructura repetitiva d'àmbit superior cal batejar l'estructura repetitiva d'àmbit superior de la qual es vol sortir amb una etiqueta al davant, seguint la sintaxi següent:

---

```
etiqueta: <sentència>;
```

---

i llavors cridar `break etiqueta;`.

#### Exemple d'utilització de la sentència `break` per trencar el flux en blocs repetitius

El programa següent mostra un parell d'estructures repetitives imbricades i etiquetades i, dins la més interna, un parell de situacions en què s'executa una sentència `break`.



Trobareu el fitxer `UtilitzacioBreakEnBucles.java` en la secció "Recursos de contingut" del web.

---

```
/* Fitxer: UtilitzacioBreakEnBucles.java
   Descripció: Programa per exemplificar la utilització de "break" en bucles.
   Autor: Isidre Guixà
*/
class UtilitzacioBreakEnBucles
{
    public static void main(String[] args)
    {
        bloc1: for (int i=1 ;; i++)
        {
            System.out.println("Principi de bloc1: i=" + i);
            bloc2: for (int j=1;; j++)
            {
                System.out.println("Principi de bloc2: j="+ j);
                if (i%2==0)
                {
                    System.out.println("Surto de bloc1");
                    break bloc1;
                }
                if (i%2!=0)
                {
                    System.out.println("Surto de bloc2");
                    break;
                }
                System.out.println("Darrera instrucció de bloc2");
            }
        }
    }
}
```

---

Veiem que la darrera instrucció del `for` intern no s'executa mai, ja que abans tenim dues sentències condicionals que són complementàries de manera que sempre una de les dues és certa i, per tant, en qualsevol cas s'executa la sentència `break` que incorporen. Com a bons programadors que sou, segurament opineu que la seqüència de les dues sentències és un mal exemple de programació, ja que en ser complementàries, la segona no té motiu de ser i el seu codi hauria de formar part de la clàusula `else` de la primera sentència. Però en aquest cas, el compilador de Java detecta que la darrera instrucció del `for` intern mai seria executable i no compila el programa. Per tant, mantenim les dues sentències condicionals per poder comprovar el funcionament de les sentències `break` que incorporen.

Veiem que tots dos bucles `for` són etiquetats. L'etiqueta `bloc2` del bucle intern és innecessària perquè en cap lloc del programa es desvia el flux cap aquesta sentència. D'altra banda, la sentència `break` de la segona sentència condicional no incorpora etiqueta i, per tant, provoca la sortida del `for` etiquetat amb `bloc2`, que és el bucle a què pertany, però també es podria posar `break bloc2` per aconseguir el mateix efecte.

El resultat de l'execució és:

```
E:\>java -Dfile.encoding=cp850 UtilitzacioBreakEnBucles
Principi de bloc1: i=1
Principi de bloc2: j=1
Surto de bloc2
Principi de bloc1: i=2
Principi de bloc2: j=1
Surto de bloc1
```

## 2) La sentència continue té la sintaxi següent:

```
continue [etiqueta];
```

La utilització de `continue` sense etiqueta en un bloc repetitiu provoca la sortida immediata del bloc saltant a la capçalera de la sentència repetitiva. En cas que hi hagi imbricació d'estructures repetitives, només afecta l'estructura en la qual s'inclou, i si es vol que salti a la capçalera d'una estructura repetitiva d'àmbit superior, cal batejar l'estructura repetitiva d'àmbit superior a la qual es vol saltar, amb una etiqueta al davant, seguint la sintaxi següent:

```
etiqueta: <sentència>;
```

i llavors cridar `continue etiqueta;`.

### Exemple d'utilització de la sentència `continue` per trencar el flux en blocs repetitius

El programa següent mostra un parell d'estructures repetitives imbricades i etiquetades i, dins la més interna, un parell de situacions en què s'executa una sentència `continue`.



Trobareu el fitxer `UtilitzacioContinueEnBucles.java` en la secció "Recursos de contingut" del web.

```
/* Fitxer: UtilitzacioContinueEnBucles.java
   Descripció: Programa per exemplificar la utilització de "continue" en bucles.
   Autor: Isidre Guixà
*/
class UtilitzacioContinueEnBucles
{
    public static void main(String[] args)
    {
        bloc1: for (int i=1; i<=2; i++)
        {
            System.out.println("Principi de bloc1: i=" + i);
            bloc2: for (int j=1 ;j<=2; j++)
            {
                System.out.println("Principi de bloc2: j="+ j);
                if (i%2==0)
                {
                    System.out.println("Salto a bloc1");
                    continue bloc1;
                }
                if (i%2!=0)
                {
                    System.out.println("Salto a bloc2");
                    continue;
                }
                System.out.println("Darrera instrucció de bloc2");
            }
        }
    }
}
```

Veiem que la darrera instrucció del `for` intern no s'executa mai, ja que abans tenim dues sentències condicionals que són complementàries de manera que sempre una de les dues és certa i, per tant, en qualsevol cas, s'executa la sentència `continue` que incorporen. Com a bons programadors que sou, segurament opineu que la seqüència de totes dues sentències és un mal exemple de progra-

mació, ja que en ser complementàries, la segona no té motiu de ser i el seu codi hauria de formar part de la clàusula `else` de la primera sentència. Però en aquest cas, el compilador de Java detecta que la darrera instrucció del `for` intern mai no seria executable i no compila el programa. Per tant, mantenim totes dues sentències condicionals per poder comprovar el funcionament de les sentències `continue` que incorporen.

Veiem que tots dos bucles `for` són etiquetats. L'etiqueta `bloc2` del bucle intern és innecessària perquè en cap lloc del programa es desvia el flux cap aquesta sentència. D'altra banda, la sentència `continue` de la segona sentència condicional no incorpora etiqueta i, per tant, provoca el salt al `for` etiquetat amb `bloc2`, que és el bucle al qual pertany, però també es podria posar `continue bloc2` per aconseguir el mateix efecte.

El resultat de l'execució és:

---

```
E:\>java -Dfile.encoding=cp850 UtilitzacioContinueEnBucles
Principi de bloc1: i=1
Principi de bloc2: j=1
Salto a bloc2
Principi de bloc2: j=2
Salto a bloc2
Principi de bloc1: i=2
Principi de bloc2: j=1
Salto a bloc1
```

---

## 1.7. Taules

Com en altres llenguatges de programació, Java permet declarar taules (també conegudes com a vectors, matrius o *arrays*) per agrupar dades d'una mateixa tipologia i poder-s'hi adreçar sota un nom únic i comú.

Per gestionar taules en Java ens cal conèixer la distinció que hi ha entre declaració i creació, com s'efectua la inicialització d'una taula, com s'accedeix a les seves cel·les, com es gestionen taules multidimensionals i la possibilitat de copiar taules entre elles.

**1) Declaració de taules enfront de creació de taules.** Java permet declarar taules de tipus de dades primitius i taules d'objectes de classes existents. En qualsevol cas, la declaració ha de seguir la sintaxi següent:

---

```
<tipus> nom [];// Nomenclatura com els llenguatges C/C++
```

---

o

---

```
<tipus> [] nom;
```

---

en què `tipus` pot ser el nom d'un tipus primitiu o el nom d'una classe.

En Java, una taula és un objecte encara que s'hagi construït a partir de tipus primitius. Per tant, com en la resta de classes, la declaració d'una taula no crea l'objecte, sinó únicament una referència que es pot emprar per referir-se a una taula. Per crear la taula, cal utilitzar l'operador `new` acompanyat del tipus de dada de la taula (primitiu o classe) i de la grandària de la taula, seguint la sintaxi següent:

---

```
nom = new <tipus>[dimensió];
```

---

En aquest cas, la utilització de l'operador `new` provoca l'execució del constructor de taules que proporciona el llenguatge Java.

La creació de la taula també es pot efectuar en la seva declaració, seguint qualsevol de les dues sintaxis de declaració:

---

```
<tipus> nom [] = new <tipus>[dimensió];
```

---

o

---

```
<tipus> [] nom = new <tipus>[dimensió];
```

---

### Exemples de declaració i creació de taules

Les sentències següents són vàlides i faciliten la declaració i creació de quatre taules d'enters (tipus primitiu).

---

```
int [] t10 = new int[10];  
int t20 [] = new int[20];  
int [] t30;  
int t40 [];  
t30 = new int[30];  
t40 = new int[40];
```

---

Però cal seguir el mateix procediment si es volen declarar taules d'objectes d'una classe X:

---

```
X [] x10 = new X[10];  
X x20 [] = new X[20];  
X [] x30;  
X x40 [];  
x30 = new X[30];  
x40 = new X[40];
```

---

Hi ha, però, una diferència important a tenir en compte en la creació d'una taula de dades d'un tipus primitiu i d'una taula d'objectes: la inicialització implícita de les cel·les.

Una taula de dades d'un tipus primitiu, com les taules `t10`, `t20`, `t30` i `t40` de l'exemple de declaració i creació de taules, en el moment de ser creada, conté tantes cel·les com la seva dimensió indiqui, inicialitzades amb valor zero pels tipus enters, reals i caràcter, i amb valor `false` pel tipus lògic, cosa que no passa quan es declara una variable individual d'un tipus primitiu.

En canvi, una taula d'objectes, com les taules `x10`, `x20`, `x30` i `x40` de l'exemple anterior, en el moment de ser creada, conté tantes cel·les com la seva dimensió indiqui, inicialitzades amb el valor `null`, ja que cada cel·la està destinada a contenir una referència a un objecte de la classe que correspongui, el qual s'haurà d'haver creat amb la sentència `new`.

**2) Inicialització explícita de taules.** El llenguatge Java permet crear taules inicialitzant-les en el moment de la seva declaració, seguint la sintaxi següent:

---

```
<tipus> [] nom = { valor1, valor2, valor3... }
```

---

en què `valor1, valor2, valor3...` han de ser del tipus per al qual es declara la taula. Així, si `<tipus>` és un tipus primitiu, `valor1, valor2, valor3...` han de ser valors del tipus primitiu corresponent, però si `<tipus>` és una classe, llavors `valor1, valor2, valor3...` han de ser referències a objectes existents de la classe corresponent, que s'hauran creat amb la sentència `new`.

#### Exemples de creació de taules inicialitzades

Les sentències següents són vàlides i faciliten la declaració i creació de taules ja inicialitzades.

```
int [] tpos = {10, 20, 30, 40};
int tneg [] = {-10, -20, -30};
// Suposant que X és una classe
X td [] = { new X(), new X(), new X() };
X [] te = { new X(), new X() };
```

**3) Accés a les cel·les d'una taula.** En Java, les diverses cel·les d'una taula estan enumerades a partir de la posició 0, com en els llenguatges C/C++.

Per accedir al valor d'una cel·la d'una taula se segueix la sintaxi que es proporciona en la majoria de llenguatges, amb la utilització de claudàtors:

```
<nom_taula> [posició]
```

A diferència de la majoria de llenguatges en què el programador ha de tenir constància de la dimensió de la taula per a la seva gestió, el llenguatge Java emmagatzema el nombre d'elements de la taula com a part de la mateixa taula, en un atribut intern anomenat `length`. Per accedir a la dimensió d'una taula `t`, cal emprar la notació `t.length`.

#### Exemple d'utilització de l'atribut `length` en taules

En el programa següent es veu la utilització de l'atribut `length` sobre diverses taules per efectuar un recorregut per les seves cel·les.

```
/* Fitxer: RecorregutTaules.java
   Descripció: Programa per comprovar el funcionament de l'atribut
               length en taules
   Autor: Isidre Guixà
*/

class X
{
}

class RecorregutTaules
{
    public static void main(String[] args)
    {
        int tpri[] = {10, 20, 30};
        X tobj[] = new X[2];
        for (int i=0; i < tpri.length; i++)
            System.out.println("tpri [" + i + "] = " + tpri[i]);
        for (int i=0; i < tobj.length; i++)
            System.out.println("tobj [" + i + "] = " + tobj[i]);
    }
}
```

En l'exemple es veu com es creen dues taules: la taula `tpri` de dades de tipus primitius, inicialitzada explícitament amb els valors 10, 20 i 30, i la taula `tobj` d'objectes de la classe `X` (creada sense cap contingut), inicialitzada implícitament amb valors `null`.



Trobareu el fitxer `RecorregutTaules.java` en la secció "Recursos de contingut" del web.

En els recorreguts de totes dues taules s'utilitza l'atribut `length` per obtenir la dimensió de cada taula. Com era d'esperar, el resultat de l'execució és:

```
G:\>java RecorregutTaules
tpri [0] = 10
tpri [1] = 20
tpri [2] = 30
tobj [0] = null
tobj [1] = null
```

**4) Taules multidimensionals.** Java no proporciona una sintaxi específica per definir taules multidimensionals com fan molts altres llenguatges, però això no vol pas dir que no sigui possible gestionar taules multidimensionals.

Atès que es pot declarar una taula de qualsevol tipus base, es poden declarar i crear taules de taules (i taules de taules de taules, i successivament). Així, per crear una taula bidimensional, cal fer:

```
<tipus> taula [][] = new <tipus> [dimensió][];
taula [0] = new <tipus> [dimensió_fila_0];
taula [1] = new <tipus> [dimensió_fila_1];

taula [dimensió-1] = new <tipus> [dimensió_fila_dimensió-1];
```

Veiem que aquesta manera de fer permet tenir taules bidimensionals rectangulars (totes les files de mateixa dimensió) i taules bidimensionals no rectangulars (files de dimensió diferent).

Per crear taules bidimensionals rectangulars hi ha una manera més senzilla que l'anterior:

```
<tipus> taula [][] = new <tipus> [dimensió1][dimensió2];
```

**5) Còpia de taules.** El llenguatge Java proporciona un mètode especial de la classe `System` per copiar taules, anomenat `arraycopy()`. Consultant la documentació de Java, trobarem la sintaxi d'aquest mètode:

```
public static void arraycopy(Object src,
                             int srcPos,
                             Object dest,
                             int destPos,
                             int length)
```

La documentació ens diu que aquest mètode copia un tros de taula de llargada especificada per l'argument `length`, de la taula origen especificada per l'argument `src` a partir de la posició indicada per l'argument `srcPos`, dins la taula especificada per l'argument `dest` a partir de la posició indicada per l'argument `destPos`.

Per al bon funcionament sense cap error d'aquest mètode cal:

- Que les referències `src` i `dest` facin referència a taules i siguin taules de tipus compatibles.

#### La classe `Object`

En Java, tot objecte d'una classe és, a la vegada, objecte de la classe `Object` facilitada pel mateix llenguatge.

Així, la sentència `Object x;` declara `x` com una referència que pot apuntar a un objecte de qualsevol classe.

#### Excepcions en Java

La màquina virtual Java, quan es troba amb un error d'execució, genera una excepció que, si no és gestionada pel programa, provoca el final no controlat del programa i informa de l'error a la interfície des d'on s'executa el programa.



- Que en cap moment no s'intenti accedir a posicions inexistents en cap de les taules. Si no, en temps d'execució, la màquina virtual Java generarà una excepció.

Cal tenir en compte que, en utilitzar el mètode `arraycopy` en taules d'objectes (no de tipus primitius), es copien les referències de les cel·les de la taula origen a les cel·les de la taula destinació i, per tant, les referències de les cel·les copiades fan referència als mateixos objectes. **!!**

#### Exemple d'utilització correcta del mètode `arraycopy`

En el programa següent es veu la utilització correcta del mètode `arraycopy` en dues taules, una de dades de tipus primitiu i una altra d'objectes.

```
/* Fitxer: CopiaTaules.java
   Descripció: Programa per comprovar el funcionament del mètode
               arraycopy()
   Autor: Isidre Guixà
*/

import java.util.Date;

class CopiaTaules
{
    public static void main(String[] args)
    {
        int tori[] = {10, 20, 30, 40, 50};
        int tdes[] = new int[6];
        System.arraycopy(tori, 2, tdes, 3, 3);
        for (int i=0; i < tdes.length; i++)
            System.out.println("tdes [" + i + "] = " + tdes[i]);

        Date dori[] = new Date[5];
        Date ddes[] = new Date[4];
        for (int i=0; i<dori.length; i++)
            dori[i] = new Date(109,0,i+1);
        System.arraycopy(dori, 3, ddes, 2, 2);
        for (int i=0; i < ddes.length; i++)
            System.out.println("ddes [" + i + "] = " + ddes[i]);
    }
}
```

Aquest programa presenta dues situacions en què utilitzem el mètode `arraycopy`.

El primer cas exemplifica la utilització del mètode `arraycopy` entre taules de dades del tipus primitiu `int`: es copien tres valors de la taula `tori` a partir de la posició 2 (30, 40 i 50) cap a la taula `tdes`, inicialitzada amb sis zeros, a partir de la posició 3.

El segon cas exemplifica la utilització del mètode `arraycopy` entre taules d'objectes de la classe `Date` proporcionada pel llenguatge Java. Per poder-ne comprovar el funcionament, hem hagut de crear alguns objectes `Date` i guardar-ne les referències a la taula `dori`. Per aconseguir-ho hem cridat un dels constructors que ens proporciona la classe `Date`, tenint en compte la informació que ens proporciona la documentació de Java:

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

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
```

Segons aquesta informació i analitzant el codi s'arriba a la conclusió que la taula `dori` s'ha emplenat amb les dates dels cinc primers dies de l'any 2009. La utilització posterior del mètode `arraycopy`



Trobareu el fitxer  
`CopiaTaules.java` en la secció  
"Recursos de contingut" del web.

copia dues referències de la taula `dori` a partir de la posició 3 (referències a les dates 4 i 5 de gener del 2009) cap a la taula `ddes`, inicialitzada amb quatre valors `null`, a partir de la posició 2.

Veiem que perquè es pugui dur a terme la compilació hem d'afegir, a l'inici del fitxer, una instrucció per indicar la ubicació de la classe `Date` que utilitzem:

---

```
import java.util.Date;
```

---

Fixem-nos en el resultat de la compilació:

---

```
G:\>javac CopiaTaules.java
Note: CopiaTaules.java uses or overrides a deprecated API.
Note: Recompile with -Xlint:deprecation for details.
```

---

El compilador ens avisa del fet que aquest programa utilitza mètodes obsolets (*deprecated*) i tenim l'opció de compilar amb l'opció `-Xlint` per obtenir més informació:

---

```
G:\>javac -Xlint CopiaTaules.java
CopiaTaules.java:21: warning: [deprecation] Date(int,int,int) in
java.util.Date has been deprecated
    dori[i] = new Date(109,0,i+1);
                  ^
1 warning
```

---

Certament, si observem la documentació de Java referent al constructor `Date` utilitzat, ens diu:

---

```
Deprecated. Instead use the constructor Date(long date)
```

---

El missatge del compilador ens convida a utilitzar un altre constructor. Però ara mateix ens és més fàcil construir una data indicant el dia, el mes i l'any que no pas indicant un valor que, segons la documentació de Java, ha de ser el nombre de mil·lisegons transcorreguts des de l'1 de gener del 1970.

Si procedim a l'execució comprovarem que el resultat és:

---

```
G:\>java CopiaTaules
tdes [0] = 0
tdes [1] = 0
tdes [2] = 0
tdes [3] = 30
tdes [4] = 40
tdes [5] = 50
ddes [0] = null
ddes [1] = null
ddes [2] = Sun Jan 04 00:00:00 CET 2009
ddes [3] = Mon Jan 05 00:00:00 CET 2009
```

---

## 1.8. Cadenes

En Java, el concepte de cadena (seqüència de dades de tipus caràcter amb operacions específiques per a la seva gestió conjunta) s'implementa amb la classe `String`. Així, doncs, per declarar referències a cadenes, farem:

---

```
String str1, str2;
```

---

`str1` i `str2` són referències per fer referència a objectes cadena que encara no existeixen.

A diferència dels llenguatges C/C++, les cadenes no finalitzen amb cap marca similar al `'\0'`.

Podem crear cadenes, com en objectes de qualsevol classe, utilitzant qualsevol dels molts constructors que proporciona el llenguatge Java i que trobarem en la seva documentació.

D'altra banda, ja sabem que Java té en compte els literals cadena (constants cadena), els quals es creen com una seqüència de caràcters tancats entre cometes dobles: "Hola Món", "La lluna i la pruna"... Quan es troba un literal cadena, el compilador crea automàticament un objecte `String` que conté la seqüència de caràcters del literal.

Per tant, atès que un literal cadena és un objecte de la classe `String`, podem utilitzar-lo per crear objectes `String` als quals fer referència des de variables `String`. La taula 9 ens mostra exemples de creació de cadenes.

Taula 9. Exemples de declaració i creació de cadenes

Codi Java	Explicació
<code>String str1 = "Hola";</code>	<code>str</code> és una referència que apunta un objecte cadena amb contingut "Hola" creat a partir d'un literal.
<code>String str; str = "La lluna";</code>	Inicialment, <code>str</code> és una referència pendent d'inicialitzar. Posteriorment, <code>str</code> fa referència a l'objecte "La lluna" creat a partir d'un literal.
<code>char t[]={'a','b','c'}; String str1 = new String(t);</code>	<code>str1</code> és una referència que apunta un objecte cadena creat amb un constructor que rep una taula de caràcters per paràmetre.
<code>String t[]; t = new String [10]; for (int i=0; i&lt;t.length; i++)     t[i]= String.valueOf(i*100);</code>	Inicialment, <code>t</code> és una referència per fer referència a una taula (encara inexistent) de referències a objectes <code>String</code> (inexistents). En segon lloc, <code>t</code> ja fa referència a una taula de deu referències per fer referència a objectes <code>String</code> (inexistents). En tercer lloc, cada cel·la de la taula <code>t</code> fa referència a una cadena que s'ha creat amb el mètode <code>valueOf</code> de la classe <code>String</code> (vegeu la documentació de Java) que crea una cadena a partir d'un valor enter passat per paràmetre.

Els objectes `String` són immutables, és a dir, una vegada creats no es poden canviar. !!

El llenguatge Java proporciona la classe `StringBuffer` per gestionar cadenes mutables.

Observem el codi següent:


```
String str1="Cadena 1";
String str2=str1;
```

En aquest cas, la màquina virtual Java té un únic objecte `String`, amb valor "Cadena 1", al qual es fa referència per mitjà de dues variables `String`: `str1` i `str2`.

Suposem que posteriorment executem el codi:

```
str2 = "Cadena 2";
```

Ara, la màquina virtual Java té dos objectes `String`, un amb valor "Cadena 1" al qual es fa referència mitjançant `str1`, i un altre amb valor "Cadena 2" al

qual es fa referència mitjançant `str2`. És a dir, no s'ha de caure en l'error de pensar que l'assignació de "Cadena 2" a `str2` quan aquesta variable feia referència a "Cadena 1" provoqui la substitució de "Cadena 1" per "Cadena 2". Els objectes `String` són immutables. 

## 2. Programació orientada a objectes en Java

En l'aprenentatge del llenguatge Java, una vegada coneguts els fonaments de programació estructurada que conté, pertoca endinsar-se en el coneixement dels conceptes de programació orientada a objectes que incorpora.

Hi ha dues possibles maneres d'enfocar l'aprenentatge de la programació orientada a objectes en Java: una, adreçada a programadors experimentats en programació orientada a objectes en altres llenguatges com el C++; l'altra, adreçada a programadors en llenguatges de tercera generació que mai no han programat en llenguatges orientats a objectes. Aquest darrer és l'enfocament d'aquest material i, per tant, anirem introduint els conceptes de programació orientada a objectes i els posarem en pràctica en el llenguatge Java, fent petites al·lusions, quan sigui adequat, a altres llenguatges orientats a objectes com C++.

### 2.1. Classes i objectes

En el desenvolupament de programes amb llenguatges de tercera generació, quan el programador es troba amb la necessitat de definir tipus de dades compostes, si és un bon programador, acostuma a definir el nou tipus de dada com a agregació d'altres tipus de dades existents (tant si són proporcionats pel llenguatge com construïts prèviament pel propi programador) i acompanya la definició del nou tipus amb la definició d'un conjunt complet de funcions per gestionar les variables que defineixi del nou tipus.

#### Exemple de definició d'un tipus agrupat de dades en llenguatge C

Davant la necessitat de gestionar dades de persones, és molt possible que una vegada analitzada la situació, el programador arribi a la conclusió que necessita agrupar, sota un nou tipus de dada anomenat `tPersona`, les dades a gestionar de les persones (suposem, per simplificar, les dades `dni`, `nom` i `edat`). En un llenguatge de tercera generació, com és el llenguatge C, el programador declararia, possiblement:

---

```
struct tPersona
{
    char dni[10];
    char *nom;
    unsigned short edat;
};
```

---

I, per gestionar persones, arribaria a la conclusió que li cal definir un conjunt de funcions que podrà utilitzar en diversos programes, com:

---

```
void inicialitzarPersona (struct tPersona *p);
/* Posa els punters per gestió dinàmica a NULL */

void netejarPersona (struct tPersona *p);
/* Neteja els camps "dinàmics" de persona tot deixant a NULL els
   punters*/

int llegirPersona (struct tPersona *p);
/* Demana les dades d'una persona per pantalla
   i omple la persona que s'ha passat per variable.
   Retorna: 0-OK
           1-Manca memòria pel nom
*/

void visualitzarPersona (struct tPersona p);
/* Mostra les dades d'una persona per pantalla */

void llegirDni (char *missatge, char *s);
/* Llegeix per pantalla un DNI mostrant el missatge a la seva
   esquerra i omplint la cadena s que s'ha passat per variable */

int llegirPersonaMenysDni (struct tPersona *p);
/* Demana les dades d'una persona, menys el dni, per pantalla
   i omple la persona que s'ha passat per variable.
   Retorna: 0-OK
           1-Manca memòria pel nom
*/

void freePersona (struct tPersona **p);
/* Allibera una persona creada dinàmicament alliberant tots els
   seus camps dinàmics i deixa a NULL el punter que hi apunta */

void visualitzarPersonaHor (struct tPersona p);
/* Mostra les dades d'una persona en una fila per pantalla */

int setDni (struct tPersona *p, char *nouDni);
/* Permet modificar el dni d'una persona
   Retorna: 0-OK
           1-El dni que s'ha passat és erroni
           (No s'ha efectuat el canvi)
*/

int setNom (struct tPersona *p, char *nouNom);
/* Permet modificar el nom d'una persona
   Retorna: 0-OK
           1-Manca memòria pel nou nom
*/

int setEdat (struct tPersona *p, int edat);
/* Permet modificar l'edat d'una persona
   Retorna: 0-OK
           1-El valor edat que s'ha passat és erroni
           (No s'ha efectuat el canvi)
*/

char *getDni (struct tPersona p);
/* Retorna el dni d'una persona */

char *getNom (struct tPersona p);
/* Retorna el nom d'una persona */

int getEdat (struct tPersona p);
/* Retorna l'edat d'una persona */
```

---

Aquesta relació de funcions únicament és a tall d'exemple. Com que donem per fet que coneixeu la gestió de dades dinàmiques, segur que totes les funcions referides us seran familiars.

A més, el bon programador haurà inclòs totes aquestes definicions (l'estructura i els prototipus) en un fitxer de capçalera `persona.h`, i el codi de les funcions l'haurà definit en un fitxer font `persona.c` incorporant el fitxer objecte `persona.obj` corresponent en totes les aplicacions en què calgui gestionar persones.

Evidentment, és aconsellable utilitzar les funcions desenvolupades per gestionar les persones, però no s'està obligat. Un programador pot accedir directament a una variable de tipus `tPersona` i canviar el `dni` existent per qualsevol tira de caràcters, tant si és vàlida com si no és vàlida com a `dni`.

Els llenguatges de tercera generació (com el llenguatge C), davant la declaració de tipus complexos i de funcions per a la seva gestió, no aporten la possibilitat de lligar la declaració del tipus amb les funcions definides de manera que sigui obligatòria la utilització de les funcions per gestionar les variables del tipus corresponent i sigui impossible accedir directament a les dades membre del tipus definit. Això representa un forat per a la robustesa del codi desenvolupat.!

La programació orientada a objectes, entre altres millores, ens soluciona aquesta problemàtica, ja que ens proporciona el concepte de classe que **encapsula** en un sol ens la definició del tipus de dada i la definició del conjunt complet de funcions per gestionar les variables (anomenades *objectes*) que es defineixin del nou tipus de dada (classe) i permet definir l'obligació d'utilitzar les funcions definides per accedir a les dades contingudes en els objectes.

Una **classe** és un tipus definit que especifica com són els atributs (**dades**) que configuren el tipus i quin és el conjunt d'operacions (**mètodes**) que permeten gestionar els atributs.

Un **objecte** és una instància (variable) de la classe i està format per tots els atributs (dades) indicats en la definició de la classe.

El fet que la programació orientada a objectes permeti definir l'obligació d'utilitzar els mètodes proporcionats per la classe, per accedir a les dades dels objectes de la classe, és una primera raó de pes per fer el salt de la programació estructurada i modular a la programació orientada a objectes. (❗)

Així, doncs, ja podem començar a descobrir de quina manera, en un llenguatge orientat a objectes determinat (com Java, en el nostre cas), hem de dissenyar classes (dades i mètodes), crear objectes, utilitzar els mètodes per gestionar els objectes...

Abans, però, convé aclarir que el programador fa dues funcions molt diferents:

- **En la utilització d'una classe** en el desenvolupament d'una aplicació, el programador veu la classe des de l'exterior i no li ha de preocupar gens

ni mica com és la implementació interna de la classe. Únicament li ha de preocupar saber quin és el coneixement que permet gestionar la classe (no pas com són els atributs interns que permeten enregistrar el coneixement) i els mètodes que la classe li proporciona per gestionar aquest coneixement.

Així, en la utilització de la classe `Date` proporcionada pel llenguatge Java i pensada per a guardar informació de moments temporals (data-hora-minut-segon), no ens interessa gens ni mica saber com s'està guardant internament el dia, el mes, l'any, l'hora... En tenim prou de saber que s'hi està guardant i donem per fet que el dissenyador i el programador de la classe s'han trencat la closca per fer-ne una gestió òptima. Únicament necessitem saber quin és el coneixement que està encapsulat en la classe (data-hora-minut-segon) i els mètodes que ens proporciona la classe per poder-ne gestionar objectes, tot introduint el coneixement, modificant-lo, consultant-lo...

- **En el disseny d'una classe**, el programador veu la classe per dins i ha de tenir tota la informació sobre les dades que incorpora (de quin tipus és i per què s'ha escollit aquest tipus, quins valors pot prendre, s'ha de deixar accés directe o no –normalment és que no–...) i sobre els mètodes que proporciona per gestionar les dades (què ha de fer cada mètode, quins arguments ha de tenir, quin resultat ha de retornar...). A més, ha de ser capaç de situar-se en el paper del programador usuari de la classe per detectar si el conjunt de mètodes que la classe incorpora és suficient per gestionar el coneixement encapsulat en aquesta.

De vegades, sobre una classe, el programador fa els dos papers, ja que és qui implementa la classe i qui la utilitza posteriorment, però ha de ser capaç de separar els dos papers i pensar que la classe dissenyada ha de poder ser utilitzada, amb posterioritat, per altres programadors i, per tant, qui la utilitzi esperarà trobar-hi un conjunt complet de mètodes per gestionar els objectes de la classe amb la documentació corresponent.

## 2.2. Gestió d'objectes d'una classe

En l'aprenentatge del llenguatge Java, abans d'introduir-nos en com es dissenya una classe, és millor conèixer com es gestionen els objectes d'una classe, ja que ho podem experimentar amb qualsevol de les classes proporcionades pel mateix llenguatge i ens ajudarà a introduir conceptes que haurem de tenir en compte en el disseny de les classes.

Per gestionar objectes ens cal saber el següent: com crear-los, com accedir-hi, com inicialitzar-los, com manipular-los i com eliminar-los.



### 2.2.1. Com es creen els objectes?

La creació d'un objecte la realitza sempre un mètode especial de la classe, anomenat *mètode constructor*, que es distingeix perquè té el mateix nom que la classe.

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 3).

Figura 3. Documentació proporcionada pel Java referent als constructors de la classe `Date`

Constructor Summary	
<code>Date(int year, int month, int day)</code>	<b>Deprecated.</b> <i>instead use the constructor <code>Date(long date)</code></i>
<code>Date(long date)</code>	Constructs a <code>Date</code> object using the given milliseconds time value.

En la figura 3 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 4).

Figura 4. Documentació detallada dels constructors de la classe `Date`

Constructor Detail	
<b>Date</b> <pre>public Date(int year,             int month,             int day)</pre> <p><b>Deprecated.</b> <i>instead use the constructor <code>Date(long date)</code></i></p> <p>Constructs a <code>Date</code> object initialized with the given year, month, and day.</p> <p>The result is undefined if a given argument is out of bounds.</p> <p><b>Parameters:</b></p> <p>year - the year minus 1900; must be 0 to 8099. (Note that 8099 is 9999 minus 1900.)</p> <p>month - 0 to 11</p> <p>day - 1 to 31</p>	
<b>Date</b> <pre>public Date(long date)</pre> <p>Constructs a <code>Date</code> 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.</p> <p><b>Parameters:</b></p> <p>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.</p>	

#### Mètodes constructors

El nom dels constructors trenca el conveni d'anomenar les dades i els mètodes de les classes amb lletres minúscules, ja que els mètodes constructors, en haver-se d'anomenar com la classe, tenen la inicial amb majúscula.

#### Mètodes obsolets en Java

El llenguatge Java té una forta evolució i, de vegades, decideix crear noves classes i/o nous mètodes per substituir les classes i/o mètodes existents, ja que es necessitaven més prestacions i/o funcionalitats que les proporcionades per les classes i/o mètodes substituïts.

Per qüestions de compatibilitat del programari ja existent amb les noves versions de Java, els mètodes substituïts de moment no s'eliminen, però s'avisava del fet que són obsolets (*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.

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

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

L'operador `new` crea un objecte assignant la memòria necessària. En el llenguatge Java, els objectes sempre són dinàmics.

#### Objectes estàtics i dinàmics en el llenguatge C++

El llenguatge C++, a diferència del llenguatge Java, permet la creació d'objectes estàtics i d'objectes dinàmics.

Així, suposant que tenim la classe `X` amb un constructor `X` sense paràmetres, en el llenguatge C++ podem fer:

```
X obj; // obj un objecte estàtic de la classe X
X *px = new X(); // s'ha creat un objecte dinàmic de la classe X
// que està apuntat pel punter px
```

#### Atès que els objectes són instàncies de la classe...

...en lloc de dir que l'operador `new` crea un objecte, també es diu que l'operador `new` instancia la classe, ja que executa la creació d'una instància de la classe.

### 2.2.2. 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.

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

```
X obj;
```

I per aconseguir que `obj` faci referència a un objecte, hem d'assignar a `obj` 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 ma-

#### Declarar un objecte

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.

teix moment en què s'efectua la declaració o amb posterioritat, tal com es veu en els exemples següents:

---

```
X obj1;           // Declaració de variable de referència no inicialitzada
obj1 = new X(...); // Creació d'objecte al que es podrà accedir via la variable de
                  // referència obj1
X obj2 = new X(...); // Declaració de variable de referència i creació d'objecte al
                  // que es podrà accedir via la variable de referència obj2
X obj3;           // Declaració de variable de referència no inicialitzada
obj3 = obj1;       // La variable obj3 fa referència al mateix objecte que fa
                  // referència la variable obj1
X obj4 = obj2;     // Declaració de variable de referència que fa referència al
                  // mateix objecte que fa referència la variable obj2
```

---

### Accés als objectes en el llenguatge C++

L'accés als objectes en el llenguatge C++ té dues possibilitats segons si l'objecte es crea estàticament o dinàmicament. Suposem:

---

```
X obj;           // obj un objecte estàtic de la classe X
X *px = new X(); // s'ha creat un objecte dinàmic de la classe X
                  // que està apuntat pel punter px
```

---

L'objecte estàtic `obj` és accessible directament pel nom. És a dir, allà on haguem de fer alguna manipulació amb aquest objecte, hi farem referència utilitzant el nom `obj`.

L'objecte dinàmic és accessible per qualsevol punter que hi apunti, com és el cas de `px`. Allà on haguem de fer alguna manipulació amb aquest objecte hi farem referència utilitzant qualsevol punter que hi estigui apuntant.

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.

Així, per exemple, suposem que per cridar un mètode d'una classe hem de passar per paràmetre un objecte que potser encara no existeix. És clar que l'hem de crear abans de cridar el mètode. Segurament pensem en fer quelcom similar a:

---

```
C obj = new C(...)
<nomMètodeInvocat> (... , obj , ...);
```

---

Si no ens interessa guardar la referència a l'objecte creat, emmagatzemada en la variable `obj`, és absurd haver creat la variable `obj` i hauríem pogut cridar directament l'operador `new` en la crida al mètode:

---

```
<nomMètodeInvocat> (... , new C(...), ...);
```

---

### 2.2.3. 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

#### Inicialització dels objectes

En certs llenguatges, la construcció dels objectes és responsabilitat exclusiva del mètode 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".

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).

Com que els constructors són mètodes que 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:

---

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

---

#### 2.2.4. Com es manipulen els objectes?

La manipulació dels objectes d'una classe s'ha de fer per mitjà dels mètodes no constructors que proporciona la pròpia classe, amb una sintaxi molt simple:

---

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

---

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

---

```
Date d = new Date (109,0,1);    // Objecte amb valor 1-1-2009
d.setYear (100);                // Objecte amb valor 1-1-2000
d.setMonth (5);                 // Objecte amb valor 1-6-2000
d.setDate (40);                 // Objecte amb valor 10-7-2000
```

---

La manera lògica de manipular els objectes d'una classe és utilitzar els mètodes 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 utilització dels mètodes i no permeten l'accés directe a les dades contingudes en els objectes.

Però, per si el dissenyador de la classe deixa la porta oberta al fet que s'accedeixi directament a les dades, interessa conèixer la sintaxi a utilitzar, que és similar a la que utilitzem per cridar un mètode:

---

```
<variableQueFaReferènciaObjecte>.<nomDada>;
```

---

Així, donada una classe `X` amb dues dades membre `a` i `b` de tipus `int` no protegides, per modificar el valor de les dades membre podríem fer:

---

```
X obj = new X();  
obj.a = 10;  
obj.b = 20;
```

---

La manipulació directa de les dades d'un objecte és una causa d'errors que poden col·locar l'objecte en un estat d'inconsistència i, per tant, la majoria de les classes privatitzen les dades dels objectes i obliguen a utilitzar els mètodes proporcionats per a la seva manipulació. **!!**

### Com es criden els mètodes en el llenguatge C++

La crida de mètodes en el llenguatge C++ té dues possibilitats segons si l'objecte es crea estàticament o dinàmicament. Suposem:

---

```
X obj;           // obj un objecte estàtic de la classe X  
X *px = new X(); // s'ha creat un objecte dinàmic de la classe X  
                // que està apuntat pel punter px
```

---

Per cridar un mètode `xxx()` sobre un objecte estàtic com `obj` o sobre un objecte dinàmic com l'apuntat pel punter `px`, utilitzarem la sintaxi següent, en què els punts suspensius indiquen la possible existència de paràmetres:

---

```
obj.xxx(...);  
px->xxx(...);
```

---

### 2.2.5. 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. Doncs bé, Java ens estalvia aquesta feina, de manera que ens permet crear tants objectes com es vulgui

(únicament limitats per les pròpies limitacions 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 un mètode 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.

## Com es destrueixen els objectes en el llenguatge C++

La destrucció d'objectes en C++ l'efectua un mètode específic categoritzat com a destructor i que es distingeix perquè s'anomena igual que la classe però amb el símbol `~` davant del nom.

El llenguatge proporciona automàticament aquest mètode, però en totes les classes que continguin dades dinàmiques és necessari definir-lo per indicar quines dades s'han d'alliberar de la memòria després de la destrucció de l'objecte. Aquest mètode també pot ser necessari utilitzar-lo per aconseguir finalitzats similars a les del mètode `finalize()` del llenguatge Java.

És a dir, suposem una classe `Persona` que conté una dada `nom` com a dada dinàmica (`char *nom`). En temps d'execució, cada objecte creat de la classe `Persona` té, a l'interior, el punter `nom` apuntant a una zona de memòria assignada dinàmicament en què hi haurà el nom de la persona corresponent. Doncs bé, quan l'objecte hagi de desaparèixer, el destructor que proporciona automàticament el llenguatge, només es preocupa de destruir el punter `nom` inclòs en la definició de la classe i no allibera la memòria assignada apuntada pel punter. En poc temps, això provocaria el col·lapse de la memòria. Per tant, en dissenyar la classe, el dissenyador de la classe ha de tenir en compte la incorporació del mètode `~Persona` que el llenguatge utilitzarà quan hagi d'eliminar un objecte de la classe enlloc d'utilitzar el destructor proporcionat pel llenguatge.

Però això no és tot, perquè recordem que en el llenguatge C++ podem tenir objectes creats estàticament i dinàmicament. Suposem:

```
X obj;           // obj un objecte estàtic de la classe X
X *px = new X(); // s'ha creat un objecte dinàmic de la classe X
                // que està apuntat pel punter px
```

L'objecte `obj` és estàtic, sabem que existeix dins el bloc (conjunt d'instruccions emmarcat entre claus `{ }`) en què ha estat declarat quan el flux d'execució ha entrat en el bloc, i que en abandonar-lo definitivament, `obj` deixa d'existir. Doncs bé, en el moment en què deixa d'existir s'executa el destructor de la classe corresponent (el que s'hagi dissenyat o, si no, el que proporciona el llenguatge). Per tant, si `obj` conté dades dinàmiques i s'ha dissenyat adequadament el destructor `~X()`, les dades dinàmiques seran correctament alliberades.

Però, aplicant el mateix raonament al punter `px`, aquest punter deixa d'existir quan el flux d'execució abandona definitivament el bloc en què s'ha declarat. Es destrueix el punter, però... Què succeeix amb l'objecte creat dinàmicament i que era apuntat per `px`? És responsabilitat del programador controlar si l'objecte ha de continuar existint, cosa que només tindrà sentit si hi ha algun altre punter d'un àmbit més extern que continua apuntant a l'objecte, ja que si l'adreça de memòria en què es troba es perd serà impossible alliberar-lo. I, per tant, el programador ha de decidir si l'objecte ha de continuar existint o ha d'eliminar-lo. Si el deixa viure, li haurà d'anar seguint la pista, ja que en algun moment l'haurà d'eliminar. Per eliminar un objecte dinàmic, el llenguatge C++ proporciona el mètode `delete` que es crida acompanyat del punter que apunta l'objecte dinàmic a eliminar, com:

```
delete px;
```

L'execució d'aquesta instrucció, si `px` apunta un objecte (no té valor `null`), provoca que l'entorn d'execució cridi el mètode destructor de la classe corresponent (el que s'hagi dissenyat o, si no, el que proporciona el llenguatge).

Ens sembla que queda clar que el recuperador de memòria del llenguatge Java ens facilita moltíssim la feina, oi? Qui voldrà programar en llenguatge C++ tenint el llenguatge Java?

## 2.3. Disseny de classes

Ara que ja sabem com crear i manipular objectes de classes existents, cal iniciar-nos en el disseny de les classes. Concretament hem de saber com s'efectua la definició d'una classe amb tots els membres (dades i mètodes); com s'aconsegueix l'ocultació de les dades i, potser, d'alguns mètodes; com es pot aconseguir el polimorfisme en els mètodes d'una mateixa

classe (sobrecàrrega de mètodes) i com es defineixen els mètodes constructors.

### 2.3.1. Definició de classes: dades, iniciadors i mètodes

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

```
DeclaracióDeLaClasse
{
    CosDeLaClasse
}
```

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.

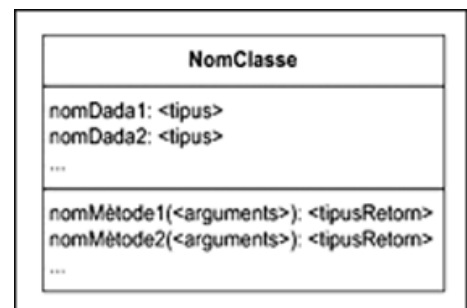
En principi, totes les classes que hem dissenyat han tingut, com a declaració, la sintaxi següent:

```
[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 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 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.



Aproximació a la notació UML per a una classe en què s'aprecien dues zones: per a les dades i per als mètodes.

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

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>
}
```

#### Classe public

Un fitxer de codi Java pot incorporar diverses classes, però només una pot estar declarada com a `public`, i en aquesta situació el nom de l'arxiu ha de coincidir amb el nom de la classe `public`.

## 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. Un d'ells és el conegut `final` que permet declarar la dada com a constant.

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

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.

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

## 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.

## Definició dels mètodes

La seqüència de definicions de mètodes consisteix en la definició (prototipus i contingut) dels diversos mètodes amb la sintaxi de Java, que és molt similar a la sintaxi de la declaració de funcions en els llenguatges C/C++. 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. Per crear els primer mètodes, però, no els necessitem.

En desenvolupar programes en 3GL estem acostumats a dissenyar accions (no retornen resultat) i funcions (retornen resultat). En el llenguatge Java, tota acció i funció es tradueix en un mètode d'una classe i en les accions, per indicar que 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 per valor, i això 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. **!!**

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 accessoras i, potser, un mètode `visualitzar()` per mostrar tot el contingut d'una persona.



Les classes anònimes són classes sense nom que es veuen en l'apartat "Classes internes anònimes" d'aquest nucli d'activitat.

### Definició dels mètodes

En entorns purs orientats a objectes, com és el llenguatge Java, tota acció i funció ha d'estar implementada com un mètode en alguna classe.

En canvi, en el llenguatge C++ conviuen els mètodes de les classes amb funcions externes a les classes.

En els llenguatges C/C++ i Java s'accepta el tipus `void` en aquelles situacions en les que cal indicar un tipus buit.

### Mètodes getter i setter

Totes les classes acostumen a proporcionar uns mètodes de lectura (*get*) i escriptura (*set*) sobre els atributs de la classe: són les anomenades *operacions accessoras*.

Una possible solució és:

```
/* Fitxer: Persona01.java
   Descripció: Disseny de la classe "Persona".
               Inclou un petit programa que comprova el funcionament dels mètodes desenvolupats.
   Autor: Isidre Guixà
*/

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

    int setDni(String nouDni)
    // Retorna: 0 si s'ha pogut canviar el dni
    //           1 si el nou dni no és correcte - No s'efectua el canvi
    {
        // 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;
    }

    int setEdat(int novaEdat)
    // Retorna: 0 si s'ha pogut canviar l'edat
    //           1 : Error per passar una edat negativa
    //           2 : Error per passar una edat "enorme"
    {
        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();
    }
}
```



**Trobareu el fitxer**  
**Persona01.java** en la secció  
"Recursos de contingut" del web.

L'execució d'aquest programa dona el resultat:

```
G:\>java -Dfile.encoding=cp850 Persona
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
```

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 usats 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` (classe embolcall del tipus primitiu) per comprovar si el valor enter de l'argument `novaEdat` del mètode `setEdat()` és massa gran per la dada `edat`.

#### Classes embolcall

El llenguatge Java proporciona per a cada tipus primitiu (`byte`, `short`, `int`, `long`, `float`, `double`, `char` i `boolean`) vuit classes corresponents (`Byte`, `Short`, `Integer`, `Long`, `Float`, `Double`, `Character` i `Boolean`), anomenades *classes embolcall* (*wrapper*, en anglès), que proporcionen dades i mètodes per a la gestió dels tipus de dades corresponents.


### 2.3.2. Encapsulació. 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.

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 classe `Persona` dissenyada fins ara (versió inclosa en el fitxer `Persona01.java`), 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?

### Comprovació sobre el grau d'ocultació de les dades d'una classe

Considerem el programa següent en què es creen objectes de la classe `Persona` dissenyada en l'arxiu `Persona01.java`.

```
/* Fitxer: CridaPersona.java
   Descripció: Programa per comprovar l'accés a objectes de la
               classe Persona des d'altres classes
   Autor: Isidre Guixà
*/
class CridaPersona
{
    public static void main(String args[])
    {
        Persona p = new Persona();
        p.dni = "--$%#@--";
        p.nom = "";
        p.edat = -23;
        System.out.println("Visualització de la persona p:");
        p.visualitzar();
    }
}
```

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 `Persona01.java` en el mateix directori) i l'execució dóna el resultat:

```
G:\>java -Dfile.encoding=cp850 CridaPersona
Visualització de la persona p:
Dni.....:--$%#@--
Nom.....:
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

### 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.



Trobareu el fitxer `CridaPersona.java` en la secció "Recursos de contingut" del web.

### Classes derivades

Les classes derivades d'una classe `X` són classes que s'obtenen a partir de la classe `X` heretant-ne tots els membres (dades i mètodes).

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:

```
[<modificadorAccés>] [<altresModificadors>] <tipusDada> <nomDada>;

[<modificadorAccés>] [<altresModificadors>] <tipusRetorn> <nomMètode> (<llistaArgs>)
{...}
```

El modificador d'accés pot prendre quatre valors:

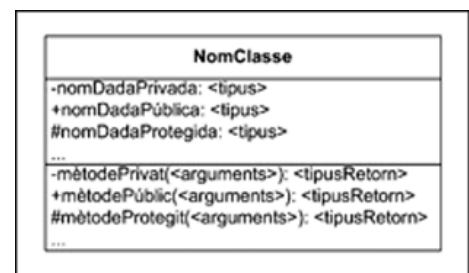
- **public**, que dona accés a tothom;
- **private**, que prohibeix l'accés a tothom menys pels mètodes de la pròpia classe;
- **protected**, que es comporta com a **public** per a les classes derivades de la classe i com a **private** per a la resta de classes;
- **sense modificador**, que es comporta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

#### 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.

Les classes `CridaPersona` i `Persona` (del fitxer `Persona01.java`), 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`.



#### Notació UML

Notació UML per a una classe, en què 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

#### 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: Persona02.java
   Descripció: Disseny de la classe "Persona" amb modificadors d'accés
               Inclou un petit programa que comprova el funcionament dels mètodes desenvolupats.
   Autor: Isidre Guixà
*/

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

    public int setDni(String nouDni)
    // Retorna: 0 si s'ha pogut canviar el dni
    //           1 si el nou dni no és correcte - No s'efectua el canvi
    {
        // 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;
    }

    public int setEdat(int novaEdat)
    // Retorna: 0 si s'ha pogut canviar l'edat
    //           1 : Error per passar una edat negativa
    //           2 : Error per passar una edat "enorme"
    {
        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();
    }
}
```



**Trobareu el fitxer**  
Persona02.java en la secció  
"Recursos de contingut" del web.

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:

```
G:\>javac CridaPersona.java
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.

### 2.3.3. Sobrecàrrega de mètodes. Polimorfisme?

De vegades, en els programes, cal dissenyar diverses versions d'accions i/o funcions 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, en els llenguatges de tercera generació no hi havia més remei que dissenyar dues funcions que havien de tenir obligatòriament noms diferents. Ho podem exemplificar en pseudocodi:

```
funció sumaEnter (n1: enter, n2: enter) retorna enter;
funció sumaReal (r1: real, r2: real) retorna real;
```

Les dues funcions 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, seria d'agrair poder-les batejar amb el mateix nom:

```
funció suma (n1: enter, n2: enter) retorna enter;
funció suma (r1: real, r2: real) retorna real;
```

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

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

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

El llenguatge C++, en què es poden definir funcions externes a les classes, ofereix la possibilitat de sobrecarregar tot tipus de funcions, tant les externes a les classes com els mètodes membres de les classes.




En el llenguatge Java, en el qual no es poden definir funcions externes a les classes perquè és un llenguatge pur orientat a objectes i tot codi ha d'estar obligatòriament dins una classe, únicament es parla de sobrecàrrega de mètodes.

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 els constructors `String()` i altres mètodes com `format()`, `getBytes()`, `indexOf()`,...

Per finalitzar amb la sobrecàrrega de mètodes, cal fer una reflexió entorn del mot **polimorfisme** tan utilitzat en la programació orientada a objectes: sobrecàrrega de mètodes = polimorfisme? 

Atès que el concepte polimorfisme indica múltiples formes, seria lògic utilitzar aquest mot en parlar de funcions sobrecarregades i, per tant, anomenar-les *funcions polimòrfiques*. Si ens cenyim a l'àmbit de la programació orientada a objectes, podríem dir que la característica tan anomenada del polimorfisme ve donada per la sobrecàrrega de mètodes. I així va ser originàriament, però avui en dia, en parlar de polimorfisme en orientació a objectes, s'és més ambiciós, i fa referència a la sobreescritura de mètodes en dissenyar classes utilitzant l'herència.



La sobreescritura de mètodes i llur equivalència amb el concepte de polimorfisme es veu en l'apartat "Sobreescritura de membres" d'aquest nucli d'activitat.

#### 2.3.4. Construcció i inicialització d'objectes

Sabem que la construcció d'un objecte s'efectua amb la utilització de l'operador `new` acompanyada d'un constructor de la classe. 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 lògic, i amb valor `null` per les variables de referència.

**2) S'executen les inicialitzacions explícites.** Les dades membres d'una classe es poden inicialitzar explícitament tot assignant expressions en la declaració dels membres.

El llenguatge C++ no facilita la inicialització explícita de les dades membre d'una classe.

#### Exemple d'inicialització explícita de dades membres en una classe

```
/* Fitxer: InicialitzacioExplicita.java
   Descripció: Programa per comprovar la inicialització explícita
               de dades membres
   Autor: Isidre Guixà
*/
import java.util.Date;

class InicialitzacioExplicita
{
    private int x = 20;
    private int y;
    private Date d = new Date (100,0,1);
    private String s;

    public static void main(String args[])
    {
        InicialitzacioExplicita obj = new InicialitzacioExplicita();
        System.out.println("x = " + obj.x);
        System.out.println("y = " + obj.y);
        System.out.println("d = " + obj.d);
        System.out.println("s = " + obj.s);
    }
}
```



Trobareu el fitxer  
InicialitzacioExplicita.java  
en la secció "Recursos de  
contingut" del web.

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:

```
G:\>java InicialitzacioExplicita
x = 20
y = 0
d = Sat Jan 01 00:00:00 CET 2000
s = null
```

**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`.

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 en concret per implementar la inicialització, ja que pot ser necessari:

- 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`.

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 `void`).
- 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`:

```
public Persona () {}

public Persona (String sDni, String sNom, int nEdat)
{
    dni = sDni;
    nom = sNom;
    if (nEdat>=0 && nEdat<=Short.MAX_VALUE)
        edat = (short)nEdat;
}
```

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

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



El fitxer `Persona03.java`, que trobareu en la secció "Recursos de contingut" del web d'aquest crèdit, conté el disseny de la classe `Persona` fins al punt actual.

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 `Persona` no hauria estat possible.

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

---

```
G:\>java -Dfile.encoding=cp850 Persona
Visualització de persona p1:
Dni.....:00000000
Nom.....:Pepe Gotera
Edat.....:33
Visualització de persona p2:
Dni.....:null
Nom.....:null
Edat.....:0
```

---

### 2.3.5. La paraula reservada `this`

El llenguatge Java proporciona la paraula reservada `this` amb dues finalitats:

**1) 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)`.

**2) 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 mit-

jaçant `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:

```
G:\>java -Dfile.encoding=cp850 Persona
Visualització de persona p2:
Dni.....:00000000
Nom.....:Pepe Gotera
Edat.....:33
Visualització de persona p3:
Dni.....:00000000
Nom.....:Pepe Gotera
Edat.....:33
```



El fitxer `Persona04.java` que trobareu en la secció "Recursos de contingut" del web conté el disseny de la classe `Persona` fins al punt actual.

### 2.3.6. La paraula reservada `static`

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:

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

---

```
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`.
- 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 lògic.

**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 tota acció/funció s'ha d'implementar forçosament com a mètode en alguna classe i, ja que aquest mètode permet aconseguir un objecte `String`, sembla lògic que resideixi dins la classe `String`.

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 sobre escriure (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.

**4)** 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 {...}
```

---

#### **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`.



Trobareu el fitxer

`ExempleDadaStatic.java` en la secció "Recursos de contingut" del web.

```
/* Fitxer: ExempleUsosStatic.java
   Descripció: Programa per comprovar les diverses utilitzacions de la
               paraula reservada "static"
   Autor: Isidre Guixà
*/
class ExempleUsosStatic
{
    private static int comptador = 0;
    private int numeroSerie;

    static { System.out.println ("Iniciador \"static\" que s'executa en carregar la classe"); }

    { System.out.println ("Iniciador que s'executa en la creació de cada objecte"); }

    public ExempleUsosStatic ()
    {
        comptador++;
        numeroSerie = comptador;
        System.out.println ("S'acaba de crear l'objecte número " + numeroSerie);
    }

    public static int nombreObjectesCreats ()
    {
        return comptador;
    }

    public static void main(String args[])
    {
        ExempleUsosStatic d1 = new ExempleUsosStatic();
        ExempleUsosStatic d2;
        d2 = new ExempleUsosStatic();
        System.out.println("Número de sèrie de d1 = " + d1.numeroSerie);
        System.out.println("Número de sèrie de d2 = " + d2.numeroSerie);
        System.out.println("Objectes creats: " + nombreObjectesCreats());
    }
}
```

L'execució del programa dóna el resultat:

```
G:\>java -Dfile.encoding=cp850 ExempleUsosStatic
Iniciador "static" que s'executa en carregar la classe
Iniciador que s'executa en la creació de cada objecte
S'acaba de crear l'objecte número 1
Iniciador que s'executa en la creació de cada objecte
S'acaba de crear l'objecte número 2
Número de sèrie de d1 = 1
Número de sèrie de d2 = 2
Objectes creats: 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`.



Trobareu el fitxer

`CarregaClasse.java` en la secció "Recursos de contingut" del web.



---

```
/* Fitxer: CarregaClasse.java
   Descripció: Programa per comprovar quan es carrega una classe
   Autor: Isidre Guixà
*/

class CarregaClasse
{
    public static void main (String args[])
    {
        System.out.println ("Punt 1. Abans de declarar la variable obj");
        ExempleUsosStatic obj;
        System.out.println ("Punt 2. Després de declarar la variable obj");
        System.out.println ("          i abans d'invocar el mètode static");
        System.out.println ("Anem a invocar el mètode static: " +
                            ExempleUsosStatic.nombreObjectesCrea());
    }
}
```

---

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`.

---

```
G:\>java -Dfile.encoding=cp850 CarregaClasse
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
```

---

### 2.3.7. Paquets de classes

El llenguatge Java proporciona un mecanisme, anomenat `package`, per poder agrupar classes.

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 (directori actual) està constituït per totes les classes dissenyades en fitxers font que no incorporen la sentència `package`.

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 `pathC1/xxx/yyy/zzz` i que el fitxer `.class` corresponent a `C2` residís en `pathC2/xxx/yyy/zzz`.

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:

---

```
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:

---

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

---

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

---

```
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.

Per acabar, cal tenir en compte com compilar les classes que pertanyen a un paquet. Tenim dues possibilitats:

- Si l'estructura de directoris corresponent al paquet ja està creada i el fitxer font resideix dins el subdirectorí corresponent a la classe, la compilació es pot efectuar amb l'ordre:

---

```
G:\> javac <pathDirectoris>/<NomClasse>.java
```

---

- Independentment de disposar de l'estructura de directoris corresponent al paquet, si volem que el procés de compilació creï l'estructura de directoris a partir d'una ubicació determinada, podem executar l'ordre:

---

```
G:\> javac -d <camí> <NomClasse>.java
```

---

Aquesta ordre crea, si no existia, l'estructura de directoris corresponent al paquet a partir de la ubicació indicada per `camí` i deixa els fitxers `.class` en el subdirectorí corresponent. En particular, si es vol que l'estructura de directoris es creï a partir de la ubicació actual, cal indicar un `.` com a `camí`.

Normalment s'utilitza la segona opció perquè permet tenir tots els font en una mateixa ubicació i anar generant els `.class` on correspongui sense haver de crear, prèviament, l'estructura de directoris.

#### Exemple de definició de paquets de classes i accés corresponent

Considerem les classes dissenyades en el fitxer següent:

---

```
/* Fitxer: ClasseC1.java
   Descripció: Classes per fer proves amb gestió de paquets
   Autor: Isidre Guixà
*/
package xxx.yyy.zzz;

public class ClasseC1
{
    int mc1=10;
}
class ClasseC1Bis
{
    int mc1=20;
}
```

---

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`:

---

```
/* Fitxer: ClasseC2.java
   Descripció: Classes per fer proves amb gestió de paquets
   Autor: Isidre Guixà
*/
package xxx.yyy.zzz;

public class ClasseC2
{
    int mc2=10;
}

class ClasseC2Bis
{
    int mc2=20;
}
```

---



Trobareu el fitxer `ClasseC1.java` en la secció "Recursos de contingut" del web.



Trobareu el fitxer `ClasseC2.java` en la secció "Recursos de contingut" del web.

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:

```
/* Fitxer: AccésIntern.java
   Descripció: Classes per fer proves amb gestió de paquets
   Autor: Isidre Guixà
*/
package xxx.yyy.zzz;

class AccésIntern
{
    public static void main (String args[])
    {
        ClasseC1 c1 = new ClasseC1();
        ClasseC1Bis c1b = new ClasseC1Bis();
        ClasseC2 c2 = new ClasseC2();
        ClasseC2Bis c2b = new ClasseC2Bis();
        System.out.println ("c1.mc1 = " + c1.mc1);
        System.out.println ("c1b.mc1 = " + c1b.mc1);
        System.out.println ("c2.mc2 = " + c2.mc2);
        System.out.println ("c2b.mc2 = " + c2b.mc2);
    }
}
```



**Trobareu el fitxer**  
**AccésIntern.java** en la secció  
**"Recursos de contingut"** del web.

Procedim a compilar els fitxers `ClasseC1.java` i `ClasseC2.java` i a generar l'estructura de directoris en diferents ubicacions, les quals han d'existir prèviament.

```
G:\>javac -d C:\ProvaC1 ClasseC1.java
```

```
G:\>javac -d G:\ProvaC2 ClasseC2.java
```

Veiem que la compilació no dóna cap error i podem comprovar l'estructura de directoris que hem generat amb aquestes compilacions:

```
G:\>dir /S /B C:\ProvaC1\*. *
C:\ProvaC1\xxx
C:\ProvaC1\xxx\yyy
C:\ProvaC1\xxx\yyy\zzz
C:\ProvaC1\xxx\yyy\zzz\ClasseC1.class
C:\ProvaC1\xxx\yyy\zzz\ClasseC1Bis.class
```

```
G:\>dir /S /B G:\ProvaC2\*. *
G:\ProvaC2\xxx
G:\ProvaC2\xxx\yyy
G:\ProvaC2\xxx\yyy\zzz
G:\ProvaC2\xxx\yyy\zzz\ClasseC2.class
G:\ProvaC2\xxx\yyy\zzz\ClasseC2Bis.class
```

Ara procedim a compilar el fitxer `AccésIntern.java`, el qual necessita tenir accés, en la seva compilació, als fitxers `.class` de les classes que s'hi criden. El compilador Java, com l'interpret Java, cerca les classes en els llocs següents:

- L'opció `-cp` o `-classpath` de la línia de crida del compilador.
- Si no se li ha indicat l'opció `-cp` o `-classpath`, cerca la variable d'entorn `CLASSPATH` definida en el sistema operatiu.
- Si tampoc troba definida la variable `CLASSPATH`, cerca les classes en el directori des d'on es crida el compilador.

Per tant, a causa de les ubicacions on hem generat els `.class` resultat de la compilació dels fitxers `ClasseC1.java` i `ClasseC2.java` i suposant que el compilat del fitxer `AccésIntern.java` (amb la seva estructura de directoris) el volem a partir de la ubicació actual, l'ordre per compilar `AccésIntern.java` és:

```
G:\>javac -cp C:\ProvaC1;G:\ProvaC2 -d . AccésIntern.java
```

Podem comprovar l'estructura de directoris generada:

```
G:\>dir /S /B xxx\*. *
G:\xxx\yyy
G:\xxx\yyy\zzz
G:\xxx\yyy\zzz\AccésIntern.class
```

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

```
G:\>java xxx.yyy.zzz.AccesIntern
c1.mc1 = 10
c1b.mc1 = 20
c2.mc2 = 10
c2b.mc2 = 20

G:\>java xxx/yyy/zzz/AccesIntern
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
   Descripció: Classes per fer proves amb gestió de paquets
   Autor: Isidre Guixà
*/
import xxx.yyy.zzz.*;

class AccesExtern
{
    public static void main (String args[])
    {
        ClasseC1 c1 = new ClasseC1();
        // ClasseC1Bis clb = 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
    }
}
```



Trobareu el fitxer `AccesExtern.java` en la secció "Recursos de contingut" del web.

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`.

Recordem que la compilació i execució d'aquest fitxer s'ha de fer amb les ordres:

```
G:\>javac -cp C:\ProvaC1;G:\ProvaC2 AccesExtern.java

G:\>java AccesExtern
```

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`.

#### Documentació resumida d'una classe mitjançant l'eina `javap`

El paquet JDK de Java incorpora l'eina `javap`, que permet obtenir informació resumida sobre el contingut d'una classe. Per a les classes proporcionades pel llenguatge Java, només cal saber

el paquet al qual pertany, mentre que per a les classes pròpies o de tercers cal saber-ne el paquet i tenir accés al fitxer `.class`, que és d'on l'eina `javap` obté la informació.

L'execució `javap -help` informa de les opcions d'execució de l'eina:

---

```
G:\>javap -help
Usage: javap <options> <classes>...

where options include:
  -c                      Disassemble the code
  -classpath <pathlist>   Specify where to find user class files
  -extdirs <dirs>         Override location of installed extensions
  -help                  Print this usage message
  -J<flag>                Pass <flag> directly to the runtime system
  -l                      Print line number and local variable tables
  -public                 Show only public classes and members
  -protected              Show protected/public classes and members
  -package                Show package/protected/public classes
                          and members (default)
  -private                Show all classes and members
  -s                      Print internal type signatures
  -bootclasspath <pathlist> Override location of class files loaded
                          by the bootstrap class loader
  -verbose                Print stack size, number of locals and args for methods
                          If verifying, print reasons for failure
```

---

Així, per obtenir informació de les classes `ClasseC1`, `ClasseC2`, `AccesIntern` i `AccesExtern` dissenyades prèviament:

---

```
G:\>javap -private -classpath C:\ProvaC1 xxx.yyy.zzz.ClasseC1
Compiled from "ClasseC1.java"
public class xxx.yyy.zzz.ClasseC1 extends java.lang.Object{
    int mc1;
    public xxx.yyy.zzz.ClasseC1();
}

G:\>javap -private -classpath G:\ProvaC2 xxx.yyy.zzz.ClasseC2
Compiled from "ClasseC2.java"
public class xxx.yyy.zzz.ClasseC2 extends java.lang.Object{
    int mc2;
    public xxx.yyy.zzz.ClasseC2();
}

G:\>javap -private xxx.yyy.zzz.AccesIntern
Compiled from "AccesIntern.java"
class xxx.yyy.zzz.AccesIntern extends java.lang.Object{
    xxx.yyy.zzz.AccesIntern();
    public static void main(java.lang.String[]);
}

G:\>javap -private AccesExtern
Compiled from "AccesExtern.java"
class AccesExtern extends java.lang.Object{
    AccesExtern();
    public static void main(java.lang.String[]);
}
```

---

### 2.3.8. 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` 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}[vfmOMe] [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 punto 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...
```

---

D'aquesta manera, podem executar directament l'aplicació fent:

---

```
java -jar nomArxiu.jar
```

---

L'opció `-jar` indica a la màquina virtual que cerqui dins el fitxer `.jar` el fitxer de manifest i executi la classe que allí s'hi indica.

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

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

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:

---

```
E:\>java -cp paquet.jar AccesExtern
```

---

```
E:\>java -cp paquet.jar xxx.yyy.zzz.AccesIntern
c1.mc1 = 10
c1b.mc1 = 20
c2.mc2 = 10
c2b.mc2 = 20
```

---

L'execució del programa `AccesExtern` no dóna cap sortida, atès que no visualitzava cap informació, però s'ha executat sense problemes, ja que la màquina virtual no s'ha queixat.

Fixem-nos que un arxiu `.jar` pot contenir diverses classes que continguin un mètode `main()` i podem executar la que ens interressi. En una aplicació Java hi haurà una classe amb un mètode `main()` que engegui l'aplicació i aquesta és la que indicariem en el fitxer de manifest. Així, supo-



sem que en el nostre cas la classe `AccesIntern` és la que conté el mètode `main()` que engega l'aplicació. En aquesta situació, generem un fitxer de nom `manifest.txt` amb el contingut:

---

```
Main-Class: xxx.yyy.zzz.AccesIntern
```

---

Ara ja podem generar el fitxer `.jar` i, posteriorment, havent mogut el fitxer a una altra ubicació, comprovar-ne l'execució correcta, i veure que podem continuar executant classes del paquet que continguin un mètode `main()`:

---

```
G:\> jar cmf manifest.txt paquet.jar AccesExtern.class xxx/yyy/zzz
```

```
...
```

```
E:\>java -cp paquet.jar AccesExtern
```

```
E:\>java -jar paquet.jar
```

```
c1.mc1 = 10
```

```
c1b.mc1 = 20
```

```
c2.mc2 = 10
```

```
c2b.mc2 = 20
```

---

## 2.4. Herència

L'**herència** és un mecanisme proporcionat pels llenguatges orientats a objectes que permet dissenyar classes com a especialitzacions de classes ja existents.

Les classes obtingudes com a especialitzacions d'altres classes s'anomenen **subclasses**, **classes derivades**, **classes filles** o **classes heretades**.

Les classes a partir de les quals es dissenyen classes especialitzades s'anomenen **classes bases**, **classes pare** o **superclasses**.

A partir de la classe `Ocell` (classe base) es poden crear les classes derivades `Cadenera`, `Periquito` i `Pardal`, ja que són especialitzacions de la classe `Ocell`.

Cal no confondre el conceptes *ser un/una* (corresponent a una relació de tipus especialització o generalització) amb *ser una característica/part de* (corresponent a una relació d'agregació o composició). (❗)

### Distinció entre *ser un/una* i *ser una característica/part de*

Considerem la classe `Persona` dissenyada amb les dades membres `dni`, `nom` i `edat`. Veiem que les seves dades membres no són altra cosa que característiques ( propietats ) dels objectes de la classe `Persona`.

Suposem que ara ens trobem amb la necessitat de gestionar informàticament les entitats `alumne` i `professor` i ens adonem que ambdues entitats són persones (ja que tenen un `dni`, `nom` i `edat` que també hem de gestionar) però amb unes característiques pròpies (diferenciades) que no tenen totes les persones. Així, per als alumnes interessa gestionar el nivell d'estudis en què estan matriculats, mentre que per als professors interessa gestionar el sou.

La solució en un llenguatge no orientat a objectes passaria per declarar uns nous tipus de dades a partir de l'agregació del tipus `tPersona` ja existent més les noves característiques. Així, en pseudocodi tindríem una definició dels nous tipus similar a:

```
tipus tAlumne = tuple
    persona : tPersona;
    nivell : caràcter;
fituple
tipus tProfessor = tuple
    persona: tPersona;
    sou: real;
fituple
```

La implementació en llenguatge C podria ser:

```
typedef struct tAlumne
{
    tPersona persona; /* Suposant declarat el typedef tPersona */
    char nivell;
};
typedef struct tProfessor
{
    tPersona persona;
    double sou;
};
```

La implementació que proporcionen els llenguatges de tercera generació no orientats a objectes barreja els conceptes *ser un/una* i *ser una característica/part de*, ja que:

- `tAlumne` s'implementa com l'agregació dels conceptes `persona` i `nivell`.

Tothom entén que “un alumne és una persona” (especialització) i ningú no diu “una persona és una característica/part d'un alumne”, mentre que tothom entén que “un alumne té/seguieix/cursa un nivell d'estudis” (característica) i ningú no diu “un alumne és un nivell”.

- `tProfessor` s'implementa com l'agregació dels conceptes `persona` i `sou`.

Tothom entén que “un professor és una persona” (especialització) i ningú no diu “una persona és una característica/part d'un professor”, mentre que tothom entén que “un professor té un sou” (característica) i ningú no diu “un professor és un sou”.

Així, doncs, la implementació proporcionada pels llenguatges de tercera generació no orientats a objectes no és prou propera a la realitat. En canvi, l'existència de l'herència en els llenguatges orientats a objectes ens permet, a partir de la classe `Persona` ja existent, declarar noves classes de manera similar a:

```
classe Alumne (derivada de Persona):
    nivell: caràcter;
ficlasse

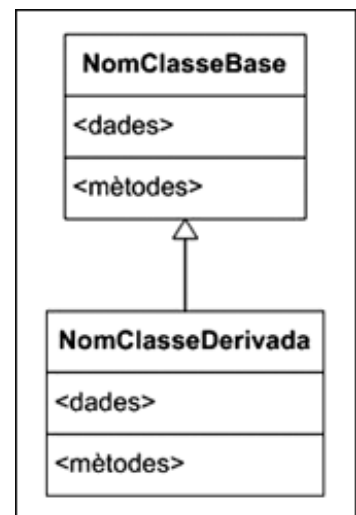
classe Professor (derivada de Persona):
    sou: real;
ficlasse
```

Aquesta implementació permet distingir la relació *ser un/una* de la relació *ser una característica/part de*:

- Els objectes de la classe `Alumne` són objectes de la classe `Persona` que, a més, cursen un nivell d'estudis.
- Els objectes de la classe `Professor` són objectes de la classe `Persona` que, a més, tenen un sou.

La utilització de l'herència per al disseny de classes especialitzades a partir d'altres classes presenta les característiques següents:

- Una classe derivada pot, als seu torn, ser una classe base per a altres classes, i donar lloc a una jerarquia de classes.



Notació UML per indicar que una classe deriva d'una altra classe, consistent en una fletxa amb punta tancada i línia contínua amb origen en la classe derivada i final en la classe base.

- Una classe derivada hereta tots els membres (dades i mètodes) de la classe base excepte els constructors.
- En el llenguatge Java hi ha la classe `java.lang.Object` que és, automàticament, superclasse per a totes les altres classes. Per tant, totes les classes (proporcionades pel llenguatge i dissenyades pels programadors) es poden considerar sota una gran jerarquia de classes que té com a arrel la classe `java.lang.Object`. En conseqüència, tots els membres de la classe `Object` són heretats per totes les classes.
- Una classe derivada pot afegir, als membres heretats, els membres propis (dades i mètodes) i pot sobreescriure els mètodes heretats.
- Els llenguatges OO acostumen a proporcionar mecanismes per prohibir la possibilitat de dissenyar classes derivades d'una determinada classe. En el llenguatge Java això s'aconsegueix declarant com a `final` la classe per a la qual es vol prohibir la derivació, seguint la sintaxi següent:

```
[final] [public] NomClasse ...
```

Així, la classe `String` proporcionada per Java és una classe `final`.

- Cal distingir entre herència simple (classe derivada d'una única classe base) i herència múltiple (classe derivada de diverses classes base). No tots els llenguatges permeten l'herència múltiple a causa de la complexitat que provoca i, atès que Java es va dissenyar amb la idea que fos un llenguatge senzill, se li va denegar l'herència múltiple.

### Complexitat de l'herència múltiple

En l'herència múltiple apareixen situacions ambigües sobre les quals cal indicar com s'ha d'actuar. Presentem les dues situacions més freqüents:

- **Ambigüïtat causada per l'existència de membres amb idèntic nom en diverses classes base de les quals estem derivant.** Com que tots els membres de les classes base s'hereten, resulta que la classe derivada es pot trobar amb membres provinents de diferents classes base i que tinguin el mateix nom. Cal, doncs, que el llenguatge proporcionï algun mecanisme per distingir aquests membres.
- **Herència repetida en la jerarquia de classes.** Considerem la situació presentada en la imatge del marge. S'hi aprecia que la classe `DF` s'obté com a herència múltiple de les classes `D1` i `D2`. Per tant, hereta tots els membres existents en `D1` (membre `d1` propi i membre `b` heretat de la classe `B`) i a `D2` (membre `d2` propi i membre `b` heretat de la classe `B`). El problema és clar: `DF` hereta dues vegades el mateix membre `b` de la classe `B` i, això, la majoria de les vegades, no interessarà. Cal, doncs, que el llenguatge proporcionï algun mecanisme per poder avortar, o no, l'herència repetida.
- **Les variables de referència són polimòrfiques.** Les variables declarades per fer referència a objectes d'una classe `X` determinada es poden utilitzar per fer referència a objectes de qualsevol classe `Y` situada, en la jerarquia de classes, per sota de la classe `X`, és a dir, de qualsevol subclasse (directa o indirecta) de la classe `X`.

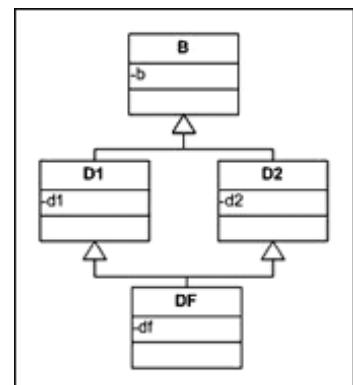
### Recordem que...

... la paraula reservada `final` també s'utilitza per declarar constants:

```
final <tpcl> <nom>;
```

en què `<tpcl>` és el nom d'un tipus primitiu o d'una classe i `<nom>` és el nom de la constant que es declara.

Hi ha diversos llenguatges OO que permeten l'herència múltiple, com C++, CLOS, Eiffel, Perl i Python, entre d'altres.



La classe `DF` hereta la dada `b` de `B` repetidament per mitjà de les classes `D1` i `D2`.

Així, doncs, com que la classe `Object` és la classe pare de totes les altres classes, podem utilitzar una variable `Object obj` per fer referència a qualsevol objecte de qualsevol classe.

Però aquesta característica ens causa un problema: com podem saber, dins un mètode que ha rebut un objecte apuntat per una variable de la classe `X`, la verdadera classe a què pertany l'objecte apuntat? (!!)

El llenguatge Java proporciona l'operador `instanceof`, que permet construir una expressió booleana que dóna resposta (`true` o `false`) a si l'objecte apuntat per una variable pertany a una classe determinada, seguint la sintaxi `<nomVariable> instanceof <NomClasse>`.

En utilitzar l'operador `instanceof` per detectar la classe a què pertany un objecte cal iniciar les preguntes per la classe situada més avall en la jerarquia de classes i continuar les preguntes cap amunt, ja que un objecte d'una classe derivada sempre és un objecte de qualsevol classe situada per damunt en la jerarquia de classes i l'operador `instanceof` ens respondrà afirmativament si li preguntem per la pertinença a qualsevol classe situada per damunt la seva classe en la jerarquia de classes.

#### **Com es pot preguntar per la classe a què pertany un objecte en el llenguatge C++?**

El llenguatge C++ proporciona el mecanisme `RTTI` (*run-time type information*) amb un objectiu similar al que proporciona l'operador `instanceof` del llenguatge Java, però la seva utilització és molt més complexa que la utilització de l'operador `instanceof` de Java.

### **2.4.1. Definició de classes derivades**

En el llenguatge Java, la definició de classes derivades 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 classes derivades, amb excepció del(s) constructor(s), el(s) qual(s), en cas de ser necessari(s), s'haurà(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 classe derivada ha d'incloure les dades, els iniciadors i els mètodes adequats a l'especialització de classe derivada 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 classe derivada cal aplicar els modificadors d'accés (`private`, `public`, `protected` o inexistent) que corresponguin.

Recordem que el codi incorporat en una classe derivada (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.

#### 2.4.2. Sobreescritura de membres. Polimorfisme?

En el procés de derivació, la classe derivada hereta tots els membres (dades i mètodes) de la classe base. A banda dels membres heretats, la nova classe pot heretar nous membres (dades i mètodes). Què succeeix si el nom d'algun dels nous membres coincideix amb el nom d'algun membre heretat? En aquesta situació es parla de sobreescritura i hem de distingir entre sobreescritura de dades i sobreescritura de mètodes.

La **sobreescritura de dades** no és gaire usual i si s'utilitza provoca confusió, ja que en una mateixa classe hi trobaríem dades diferents amb el mateix nom i això provoca embolics a l'hora d'accedir-hi. Per evitar malentesos és millor no provocar aquesta situació. Però si, tot i els consells, es decideix definir dades de noms coincidents amb les dades de la superclasse, com es pot distingir la dada heretada de la dada pròpia de la classe?

La paraula reservada `super` utilitzada en un mètode no constructor de una classe fa referència a la superclasse de la classe a què pertany el mètode i, per tant, dóna accés a les dades heretades de la classe base, amb la sintaxi `super.nomDadaHeretada`.

L'accessibilitat que proporciona la paraula `super` no se salta el control d'accés entre les classes proporcionat pels modificadors d'accés (`public`, `private`, `protected` o inexistent).

El terme anglès per a la sobreescritura, molt emprat en informàtica, és *overriding*.


A diferència de la sobreescritura de dades, la sobreescritura de mètodes sí que és usual.


La **sobreescritura de mètodes** és una característica de la programació orientada a objectes que permet a una classe derivada oferir una implementació específica d'un mètode heretat.

Si algun dels mètodes heretats de la classe base no proporciona la funcionalitat que s'espera dins la classe derivada, es pot tornar a dissenyar en la classe derivada.

Hi ha algunes regles a tenir en compte en la sobreescritura de mètodes:

- 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.

La versió heretada d'un mètode sobreescrit desapareix en la classe derivada, però no desapareixen els mètodes heretats que eren sobrecàrregues del mètode sobreescrit. 

Els llenguatges OO acostumen a proporcionar mecanismes per prohibir sobreescriure determinats mètodes. 

En el llenguatge Java això s'aconsegueix declarant com a `final` el mètode pel qual vol prohibir la sobreescritura, seguint la sintaxi següent:

```
[modificadorAccés][static][final] <valorRetorn> nomMètode (...)
```

Cal vigilar de no confondre els conceptes de sobrecàrrega i de sobreescritura.

La **sobrecàrrega** és la característica que permet tenir, en una mateixa classe, diferents mètodes amb el mateix nom, mentre que la **sobreescritura** és la característica que permet canviar la implementació d'un mètode de la classe base en la classe derivada.

I, respecte a l'accés als mètodes, hem de conèixer dues característiques:

- Si dins una classe cal accedir a la versió de la classe base 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).



El llenguatge Java proporciona un mecanisme de control d'excepcions que es veu en l'apartat "Excepcions" del nucli d'activitat "Classes fonamentals" d'aquesta unitat didàctica.

Recordem que la paraula reservada `final` també s'utilitza per declarar constants i per declarar classes que no puguin ser derivades.

#### Recordem que...

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

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

```
class X
{   met1 () {...codi X...} }

class Z extends X
{   met1 () {...codi Z...} // Sobreescritura de met1() d'X
    met2 () {...} // Mètode inexistent a la classe X
}
X ox = new X ();
X oz = new Z ();

ox.met1(); // (1)
oz.met1(); // (2)
ox.met2(); // (3)
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 `met1()` 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();
```

### Mètodes polimòrfics en el llenguatge C++

En el llenguatge C++ els mètodes no són, per defecte, polimòrfics. Hi ha, però, la possibilitat de fer-los polimòrfics utilitzant la paraula reservada `virtual` proporcionada pel llenguatge C++. En el moment en què un mètode es declara `virtual` en una classe, totes les seves sobreescritures en les classes derivades també són automàticament `virtual`.

Una vegada coneguts els conceptes de sobreescritura de mètodes i del comportament polimòrfic dels mètodes, podem definir el polimorfisme en el context actual de la programació orientada a objectes.

El **polimorfisme**, en el context actual de la programació orientada a objectes, defineix la propietat que tenen els mètodes heretats d'adequar el seu comportament a la classe a què pertany l'objecte, i això és possible gràcies a la sobreescritura de mètodes i al fet que els mètodes siguin polimòrfics.

### Exemple de sobreescritura 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.



Trobareu el fitxer `Sobreescritura.java` en la secció "Recursos de contingut" del web.

---

```
/* Fitxer: Sobreescritura.java
   Descripció: Exemplificació de sobreescritura de dades i mètodes
   Autor: Isidre Guixà
*/

class A
{
    int d=10;

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

    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 );
    }
}

class B extends A
{
    int d=20;

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

class C extends B
{
    int d=30;

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

    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();
        A ob = new B();
        A oc = new C();
        ((C)oc).visibilitat(); // (1)
        System.out.println("Crides al mètode xxx() existent a les tres classes:");
        oa.xxx(); // (2)
        ob.xxx(); // (3)
        oc.xxx(); // (4)
        System.out.println("Crides al mètode xxx(int x) existent a les tres classes:");
        oa.xxx(0); // (5)
        ob.xxx(0); // (6)
        oc.xxx(0); // (7)
    }
}
```

---

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:

---

```
G:\>javac Sobreescriptura.java

G:\>java -Dfile.encoding=cp850 C
Des del mètode "visibilitat" en C:
d en C = 30
d en B = 20
Crides al mètode xxx() existent a les tres classes:
d en A = 10
d en B = 20
d en A = 10
d en C = 30
d en B = 20
d en A = 10
Crides al mètode xxx(int x) existent a les tres classes:
Sóc xxx d'A aplicat sobre un objecte de la classe A
Sóc xxx d'A aplicat sobre un objecte de la classe B
Sóc xxx d'A aplicat sobre un objecte de la classe C
```

---

Finalment, sobre la sobreescriptura de mètodes, és interessant conèixer quatre mètodes de la classe `Object` (heretats, per tant, en totes les classes) per als quals en pot ser necessària o convenient la sobreescriptura: `finalize()`, `equals()`, `hashCode()` i `toString()`.

### 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 destruir-ne els objectes.

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

---

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

---

## Sobreescriptura dels 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.



Trobareu el fitxer  
CompararStringsViaOperador  
Comparacio.java en la secció  
"Recursos de contingut" del web.

### 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:

```
/* Fitxer: CompararStringsViaOperadorComparacio.java
   Descripció: Comparació d'objectes String via l'operadors ==
   Autor: Isidre Guixà
*/

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:

```
G:\>java CompararStringsViaOperadorComparacio
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?

Tenint en compte que l'operador de comparació == compara el valor de les referències (direccions a objectes) i dona resultat cert únicament si les referències comparades apunten el mateix objecte (direccions iguals), és lògic el resultat fals 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 dona resultat cert? 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 apuntes al mateix objecte. Aquest com-

portament, que en altres llenguatges com C++ seria molt perillós perquè tenim mecanismes per anar, mitjançant la referència (punter en C++) a l'interior de la cadena i modificar-ne el contingut, 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 `==`. Ens cal algun mecanisme per poder comparar el contingut dels objectes apuntats per referències enlloc de les direccions.

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 sobreescritura. És el mètode següent:

---

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

---

La implementació d'aquest mètode en la classe `Object` (que és la que s'hereta en cas de no sobre escriure'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 es sobre escriu, 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.

#### **Exemple de comparació de cadenes mitjançant el mètode `equals()`**

El següent programa ens mostra la utilització del mètode `equals()` per comparar cadenes i la diferència de resultats respecte la utilització de l'operador `==`.

---

```
/* Fitxer: CompararStringsViaMetodeEquals.java
   Descripció: Comparació d'objectes String via mètode equals().
   Autor: Isidre Guixà
*/

class CompararStringsViaMetodeEquals
{
    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','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.print("dni1 == dni2 : " + (dni1 == dni2));
        System.out.println("\tdni1.equals(dni2) : " + dni1.equals(dni2));
        System.out.print("dni1 == dni3 : " + (dni1 == dni3));
        System.out.println("\tdni1.equals(dni3) : " + dni1.equals(dni3));
        System.out.print("dni1 == dni4 : " + (dni1 == dni4));
        System.out.println("\tdni1.equals(dni4) : " + dni1.equals(dni4));
        System.out.print("dni3 == dni4 : " + (dni3 == dni4));
        System.out.println("\tdni3.equals(dni4) : " + dni3.equals(dni4));
    }
}
```

---



Trobareu el fitxer  
`CompararStringsViaMetodeEquals.java` en la secció  
 "Recursos de contingut" del web.

Si executem el programa, obtenim els resultats esperats:

```
G:\>java CompararStringsViaMetodeEquals
dni1 : 00000000
dni2 : 00000000
dni3 : 00000000
dni4 : 00000000
dni1 == dni2 : true dni1.equals(dni2) : true
dni1 == dni3 : false dni1.equals(dni3) : true
dni1 == dni4 : false dni1.equals(dni4) : true
dni3 == dni4 : false dni3.equals(dni4) : true
```

Com a primera aplicació de la sobreescritura del mètode `equals()`, podem pensar en la seva sobreescritura en la classe `Persona`, tenint en compte que considerarem que dos objectes `Persona` són iguals si tenen el mateix `dni`:

```
public final 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 sobreescritura 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 mateixa 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`.

El mètode `getClass()` de la classe `Object` és un mètode final i, per tant, no es pot sobreesciure.

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` derivada 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 final 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 sobreescritura del mètode `equals()` de la classe `Object` i, per tant, amb aquesta implementació, la nostra classe disposaria de dos mètodes `equals()`:

```
public final boolean equals (Object obj); // Heretat d'Object
public final boolean equals (Persona obj); // Nou a la classe
```

S'aconsella sobre escriure el mètode `equals()` de la classe `Object` enlloc de crear nous mètodes `equals()`.

### Sobreescritura del mètode `hashCode()`

En la definició d'una classe, si sobreescrivim el mètode `equals()` heretat de la classe `Object` també hem de sobre escriure el mètode `hashCode()` també heretat de la classe `Object`. **!!**

El mètode `hashCode()` de la classe `Object` és un mètode que, cridat sobre un objecte, retorna la direcció en què es troba. Per tant, és evident que dos objectes iguals segons el mètode `equals()` de la classe `Object` també tenen el mateix `hashCode()`. I aquesta similitud entre els mètodes `equals()` i `hashCode()` s'ha de mantenir sobre qualsevol classe en la qual, algun dels dos, es sobre escriui. L'expressió següent per a dos objectes `o1` i `o2` d'una classe sempre ha de ser certa:

```
!o1.equals(o2)) || (o1.hashCode() == o2.hashCode())
```

#### La màquina virtual Java

La màquina virtual Java, en la creació de tot objecte, assigna un valor `Hash` enter que caracteritza l'objecte i que és únic, exclusiu i immutable per a cada objecte creat.

La classe `Object` proporciona un mètode `hashCode()` que retorna aquest valor característic de l'objecte.

El mètode `hashCode()` s'ha de sobre escriure amb molt de compte!

Fixem-nos que, si dos objectes no són iguals segons el mètode `equals()`, poden tenir el mateix valor `hashCode()` o no. **!!**

Tot programador novell en el llenguatge Java comprèn ràpidament que la sobreescritura errònia del mètode `equals()` o la inexistència d'aquesta sobreescritura en una classe pot resultar fatal i, per tant, gairebé sempre se sobreescrigui el mètode `equals()`. Però la no sobreescritura del mètode `hashCode()` o la sobreescritura errònia també pot resultar fatal si els objectes de la classe s'utilitzen com a claus en classes que utilitzen tècniques *Hash* (com en les classes `Hashtable`, `HashMap`, `TreeMap`...). Per tant, si no sobreescrivim o sobreescrivim erròniament el mètode `hashCode()` és probable que no ho notem, però en patirà les conseqüències el programador que en un futur vulgui utilitzar els objectes de la nostra classe com a claus per a un *hashing*.

La implementació de les tècniques *Hash* se surten de l'objectiu d'aquest material i hi ha molts llibres que ho expliquen amb detall.

S'imposa, doncs, implementar el mètode `hashCode()` de manera que satisfaci la condició següent:

---

```
!o1.equals(o2)) || (o1.hashCode() == o2.hashCode())
```

---

Així, una implementació del mètode `hashCode()` que sempre verificaria la condició, podria ser:

---

```
int hashCode()  
{  
    return 0;  
}
```

---

L'objectiu d'una tècnica *Hash* és, a partir d'una clau, obtenir de manera ràpida l'element corresponent (accés directe per valor) gràcies al valor que retorna `clau.hashCode()` i, si totes les claus tenen el mateix `hashCode()`, aconseguirem poca eficiència (es convertiria internament en una cerca seqüencial amb temps de resposta lineal). Per tant, la versió proporcionada seria vàlida però del tot ineficient.

Normalment, la sobreescritura del mètode `hashCode()` es basa en la crida dels mètodes `hashCode()` d'alguna de les dades membres de la classe i, per tant, és molt important que cada classe tingui implementada correctament la sobreescritura del mètode `hashCode()`. **!!**

La majoria de les vegades, la sobreescritura del mètode `equals()` en una classe que té  $n$  dades membre es basa en comparacions d'un subconjunt d' $m$  dades de les  $n$  que formen part de la classe. En aquesta situació disposem d'un mètode `static genericHash()` que podem incorporar en un paquet d'utilitats i que ens serveix per dissenyar un mètode `hashCode()` per a la nostra classe que és coherent amb el mètode `equals()` i és eficient. La definició del mètode `genericHash()` és:

---

```
public static int genericHash(int... campsHash)
{
    int resultat = 17;
    for (int hash : campsHash)
    {
        resultat = 37 * resultat + hash;
    }
    return resultat;
}
```

---

La implementació del mètode `genericHash()` no té cap error en la declaració dels arguments, i si us ho sembla és perquè utilitza una tècnica poc coneguda: el llenguatge Java permet que el darrer argument d'un mètode sigui *n-variable*, fet que possibilita que el mètode es pugui cridar amb un nombre indefinit de paràmetres del mateix tipus.

La sintaxi per indicar un argument *n-variable* és:

---

```
<tipusArgument>... <nomArgument>
```

---

en què `nomArgument` correspon a una taula de dades `tipusArgument`.

Així, doncs, el mètode `genericHash()` es pot cridar amb un nombre indeterminat de valors enters que es recolliran en la taula `campsHash` sobre la qual s'efectua un recorregut per calcular el resultat. Coneixíeu la possibilitat de recorregut que mostra la instrucció `for` del mètode?

Perquè el resultat que dóna el mètode `genericHash()` sigui vàlid i coherent amb el mètode `equals()` dissenyat en la classe, cal cridar el mètode `genericHash()` passant com a arguments la llista de valors `hashCode()` de les dades membre de la classe que es comparen en el mètode `equals()`.

Exemplifiquem-ho. Suposem que la classe `C` té *n* dades de les quals el subconjunt d'*m* dades anomenades `camp1`, `camp2`... `campM` són les que s'utilitzen en el disseny del mètode `equals()` de manera similar al següent:

---

```
boolean equals(Object obj)
{
    if (obj == this) return true;
    if (obj == null) return false;
    if (obj instanceof C)
    {
        C aux = (C)obj;
        return camp1.equals(aux.camp1) && camp2.equals(C.camp2)
            && ... && campM.equals(C.campM);
    }
    return false;
}
```

---

En aquesta situació, programariem el mètode `hashCode()` de la classe com:

---

```
int hashCode()
{
    return genericHash(camp1.hashCode(), camp2.hashCode() ... campM.hashCode());
}
```

---

No és un objectiu d'aquest material fonamentar l'eficiència del mètode `genericHash()` per obtenir el `hashCode()` d'una classe.

Suposant que hem incorporat el mètode `genericHash()` en una classe anomenada `AlgorismesHash`, podem implementar el mètode `hashCode()` a la classe `Persona`:

---

```
public final int hashCode ()
{
    return AlgorismesHash.genericHash(dni.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!

#### Visualització d'una persona per mitjà de `System.out.println()`

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:

---

```
G:\>javac Persona04.java
```

```
G:\>java Persona
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()`.

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 valor següent:

---

```
getClass().getName() + '@' + Integer.toHexString(hashCode())
```

---

Si analitzem la funcionalitat dels diferents mètodes indicats (utilitzant la documentació de Java), arribarem a la conclusió que:

- **`getClass().getName()`** dóna el nom de la classe a què pertany l'objecte sobre el qual s'ha aplicat el mètode `toString()`.
- **`Integer.toHexString(hashCode())`** dóna, en format hexadecimal, el valor identificador (`hashCode()`) de l'objecte.

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.

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:

---

```
public String toString()
{
    return dni + " - " + nom;
}
```

---



Trobareu el fitxer `Persona05.java` (evolució del disseny de la classe `Persona` fins al punt actual) i el fitxer `AlgorismesHash.java` en la secció "Recursos de contingut" del web.

### 2.4.3. Construcció i inicialització d'objectes derivats

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 lògic, 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 `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 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 accessorres (`getter` i `setter`).



Trobareu els fitxers `Persona06.java` (evolució del disseny de la classe `Persona` fins al punt actual) i `Alumne06.java` en la secció "Recursos de contingut" del web.

```
/* Fitxer: Alumne06.java
   Descripció: Classe derivada de la classe Persona (fitxer Persona06.java) amb un gran nombre de
               constructors per comprovar la utilització de les crides super(...) i this(...)
               Inclou un petit programa que comprova el funcionament dels mètodes desenvolupats.
   Autor: Isidre Guixà
*/

class Alumne extends Persona
{
    private char nivell = '?';
    /** Valors vàlids:
        B = Batxillerat
        M = Cicle Fromatiu Mitjà
        S = Cicle Formatiu Superior
        ? = Desconegut
    */

    public Alumne () {}

    public Alumne (String sDni, String sNom, int nEdat, char cNivell)
    {
        super (sDni, sNom, nEdat);
        nivell = validarNivell (cNivell);
    }
    public Alumne (Persona p, char cNivell)
    {
        super (p);
        nivell = validarNivell (cNivell);
    }

    public Alumne (Persona p)
    {
        this (p, '?');
    }

    public Alumne (String sDni, String sNom, int nEdat)
    {
        this (sDni, sNom, nEdat, '?');
    }

    public Alumne (Alumne a)
    {
        this (a.dni, a.nom, a.edat, a.nivell);
    }

    public void setNivell (char nouNivell)
    {
        nivell = validarNivell (nouNivell);
    }

    private char validarNivell (char nivell)
    /** Valida el "nivell" passat per paràmetre, de manera que retorna el valor
        vàlid i en majúscula. Si no era vàlid, retorna '?'. */
    {
        nivell = Character.toUpperCase(nivell);
        if (nivell!='B' && nivell!='S' && nivell!='M') nivell='?';
        return nivell;
    }

    public char getNivell () { return nivell; }

    public void visualitzar ()
    {
        super.visualitzar ();
        System.out.println("Nivell.....:" + nivell);
    }

    public String toString()
    {
        String s = "Dni: " + dni + " - Nom: " + nom + " - Edat: " + edat + " - Nivell: ";
        switch (nivell)

```

```

        {
            case 'B': s = s + "Batxillerat"; break;
            case 'M': s = s + "Cicle F. Mitjà"; break;
            case 'S': s = s + "Cicle F. Superior"; break;
            default : s = s + "???"; break;
        }
        return s;
    }

    public static void main(String args[])
    {
        Alumne t[] = new Alumne[6];
        t[0] = new Alumne();
        t[1] = new Alumne("00000000", "Pepe Gotera", 33, 'b');
        t[2] = new Alumne("00000000", "Pepe Gotera", 33);
        t[3] = new Alumne(new Persona("00000000", "Pepe Gotera", 33), 'b');
        t[4] = new Alumne(new Persona("00000000", "Pepe Gotera", 33));
        t[5] = new Alumne(t[3]);

        for (int i=0; i<t.length; i++) System.out.println(t[i]);
    }
}

```

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.

L'execució del programa inclòs en la classe `Alumne` del fitxer `Alumne06.java` visualitza:

```

G:\>java Alumne
Dni: null - Nom: null - Edat: 0 - Nivell: ???
Dni: 00000000 - Nom: Pepe Gotera - Edat: 33 - Nivell: Batxillerat
Dni: 00000000 - Nom: Pepe Gotera - Edat: 33 - Nivell: ???
Dni: 00000000 - Nom: Pepe Gotera - Edat: 33 - Nivell: Batxillerat
Dni: 00000000 - Nom: Pepe Gotera - Edat: 33 - Nivell: ???
Dni: 00000000 - Nom: Pepe Gotera - Edat: 33 - Nivell: Batxillerat

```

## 2.4.4. Classes abstractes

En el desenvolupament de classes, de vegades pot interessar dissenyar una classe fonamental a partir de la qual es derivin altres classes i per a la qual es vulgui prohibir la creació d'objectes.

Així, per exemple, quan l'analista d'una possible aplicació per a la gestió informàtica d'un centre educatiu decideix dissenyar les classes `Alumne` i `Professor` i s'adona que ambdós conceptes tenen en comú el concepte de `Persona`, decideix que li convé dissenyar la classe `Persona` com a classe base de les classes `Alumne` i `Professor`. Pot passar que el dissenyador vulgui prohibir instanciar objectes de la classe `Persona` per garantir que en l'aplicació mai no es puguin gestionar persones sinó que sempre s'hagin de gestionar alumnes o professors. Per aconseguir això haurà de declarar la classe `Persona` com a abstracta.

Una classe **abstracta** és una classe base pensada per crear-ne especialitzacions i de la qual no es poden instanciar objectes.

En el llenguatge Java s'indica que una classe és abstracta amb la paraula reservada `abstract` en la declaració de la classe, seguint la sintaxi següent:

---

```
[abstract] [public] NomClasse ...
```

---

El compilador detecta qualsevol intent d'instanciar un objecte d'una classe abstracta.

Si es volen obligar totes les subclasses d'una classe abstracta a proporcionar un mètode determinat amb una implementació específica a cada subclasse, cal declarar el mètode a la classe base i declarar-lo abstracte, amb la sintaxi següent:

---

```
[modificadorAccés] [static] [abstract] <valorRetorn> nomMètode (<paràmetres>);
```

---

Les regles a tenir en compte en el disseny de mètodes abstractes són:

- Els mètodes abstractes en Java no poden incorporar codi.
- Una classe abstracta pot tenir mètodes abstractes i mètodes no abstractes.
- Un mètode abstracte ha de pertànyer obligatòriament a una classe abstracta.
- Si un mètode abstracte heretat no se sobreescriu en una classe derivada, la classe derivada també s'ha de declarar abstracta.

#### **Evolució de la classe `Persona` cap a la classe abstracta i, de retruc, evolució de la classe `Alumne`, derivada de la classe `Persona`**

A continuació farem evolucionar la classe `Persona` cap a una classe abstracta pensant que la classe `Alumne` en sigui una especificació. Quins problemes trobem?

El mètode `clonar()` que havíem incorporat i que ens generava una nova `Persona` a partir d'una persona existent no és vàlid, ja que en tractar-se d'una classe abstracta no ens permet crear persones i, d'altra banda, si no hi ha persones tampoc no té sentit l'existència d'un mètode que les cloni, no?

D'altra banda, si volem que les classes derivades de `Persona` proporcionin obligatòriament el mètode `clonar`, declararem aquest mètode com a abstracte. Així, doncs, el mètode `clonar()` en la classe `Persona` queda com:

---

```
public abstract Persona clonar ();
```

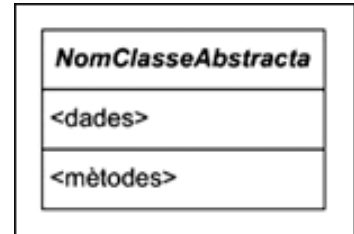
---

Estem obligats a sobreescrivre el mètode `clonar()` en la classe `Alumne`:

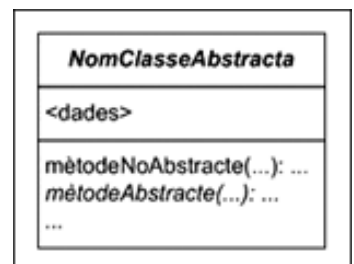
---

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

---



Notació UML per al disseny d'una classe abstracta, consistent a indicar-ne el nom en cursiva.



Notació UML per al disseny de mètodes abstractes, consistent a indicar-ne el nom en cursiva.

Amb aquestes modificacions, tenim la classe abstracta `Persona` i una classe derivada no abstracta `Alumne`. La versió evolucionada de la classe `Alumne` conté alguns constructors que poden semblar no vàlids perquè reben per paràmetre referències a `Persona`:

```
public Alumne (Persona p, char cNivell)
{
    super (p);
    nivell = validarNivell (cNivell);
}

public Alumne (Persona p)
{
    this (p, '?');
}
}
```



Trobareu els fitxers `Persona07.java` i `Alumne07.java` (evolució del disseny de les classes `Persona` i `Alumne` fins al punt actual) en la secció “Recursos de contingut” del web.

El compilador no es queixa! Com és possible? Té sentit la seva existència?

Respecte la primera pregunta, és natural que el compilador no es queixi. Recordem que les variables de referència són polimòrfiques i, per tant, una variable de referència a `Persona` (com són les variables `p` de la llista de paràmetres dels dos constructors) podria estar apuntant a objectes `Persona` (impossible perquè és classe abstracta) o a objectes de qualsevol classe derivada (`Alumne`, `Professor`...).

La segona pregunta té resposta afirmativa donat que pot interessar crear objectes `Alumne` a partir d'objectes d'altres classes derivades de `Persona`. Així, si `p` apunta un `Professor` i interessa crear un `Alumne` de manera que les dades referents a `Persona` es copiïn de `p`, els dos constructors ens poden servir.

## 2.5. Interfícies

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 penguin les classes corresponents a les diferents tipologies de treballadors (metal·lúrgics, hostaleria, informàtics, professors...). A més, disposem del concepte abstract 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 **interfícies**.

Una **interfície** és una maqueta contenidora d'una llista de mètodes abstractes i dades membre (de tipus primitius o de classes).

Les **dades membre**, si existeixen, són implícitament considerades `static` i `final`.

Els **mètodes**, si existeixen, són implícitament considerats `public`.

Per entendre en què ens poden ajudar les interfícies, ens cal saber:

- Una interfície pot ser implementada per múltiples classes, de manera similar a com una classe pot ser superclasse de múltiples classes.
- Les classes que implementen una interfície estan obligades a sobreescrivre tots els mètodes definits en la interfície. Si la definició d'algun dels mètodes a sobreescrivre coincideix amb la definició d'algun mètode heretat, aquest desapareix de la classe.
- Una classe pot implementar múltiples interfícies, a diferència de la derivació, que només es permet d'una única classe base.
- Una interfície introdueix un nou tipus de dada, per la qual mai no hi haurà cap instància, però sí objectes usuaris de la interfície –objectes de les classes que implementen la interfície. Totes les classes que implementen una interfície són compatibles amb el tipus introduït per la interfície.
- Una interfície no proporciona cap funcionalitat a un objecte (ja que la classe que implementa la interfície 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 interfícies 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 interfície no es pot instanciar, però sí s'hi pot fer referència.

Així, si `I` és una interfície i `C` és una classe que implementa la interfície, es poden declarar referències al tipus `I` que apuntin objectes de `C`:

---

```
I obj = new C (<paràmetres>);
```

---

- Les interfícies poden derivar d'altres interfícies i, a diferència de la derivació de classes, poden derivar de més d'una interfície.

Així, si dissenyem la interfície `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 interfície.

Algú no experimentat en la gestió d'interfícies pot pensar: per què tan enrenou amb les interfícies si haguéssim pogut dissenyar directament un mètode anomenat `calculImportJubilacio()` a les classes afectades sense necessitat de definir cap interfície?

La resposta rau en el fet que la declaració de la interfície porta implícita la declaració del tipus `Treballador` i, per tant, podrem utilitzar els objectes de totes les classes que implementin la interfície 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`:

---

```
public void enviarEsborranyIRPF(Treballador t) {...}
```

---

Pel fet d'existir la interfície `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)`.

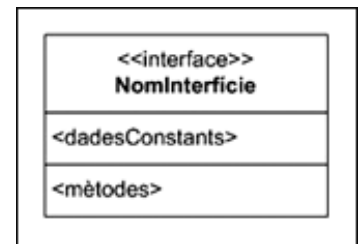
La sintaxis per declarar una interfície és:

---

```
[public] interface <NomInterfície> [extends <NomInterfície1>, <NomInterfície2>...]
{
    <cosInterfície>
}
```

---

Les interfícies també es poden assignar a un paquet. La inexistència del modificador d'accés `public` fa que la interfície sigui accessible a nivell del paquet.



Notació UML per al disseny d'una interfície, similar al disseny d'una classe però amb la paraula `<<interface>>` acompanyant el nom de la interfície.

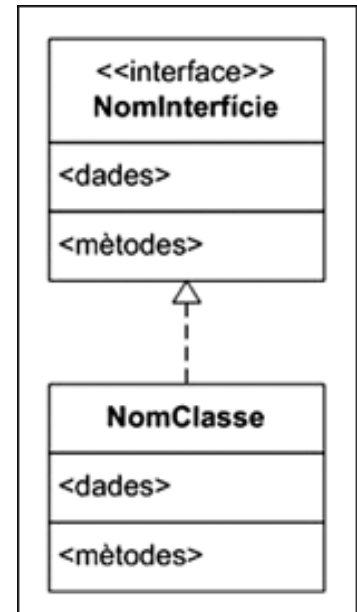


Per als noms de les interfícies, s'aconsella seguir el mateix criteri que per als noms de les classes. En la documentació de Java, les interfícies s'identifiquen ràpidament entre les classes perquè estan en cursiva.

El cos de la interfície és la llista de mètodes i/o constants que conté la interfície. 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 interfícies és:

```
[final] [public] class <NomClasse> [extends <NomClasseBase>]
    implements <NomInterfície1>, <NomInterfície2>...
{
    <CosDeLaClasse>
}
```



Notació UML per indicar que una classe implementa una interfície, consistent en una fletxa amb punta tancada i línia puntejada amb origen en la classe i final en la interfície.

Els mètodes de les interfícies 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 interfície són `static` i `final`, i atès que les interfícies 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 interfície, de manera tan simple com:

```
System.out.println (DiesSetmana.NOMS_DIES[DILLUNS]);
```

### Exemple de disseny d'interfície i implementació en una classe

Es presenten un parell d'interfícies 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 interfície `B`, però com que aquesta interfície deriva de la interfície `A` resulta que la classe està implementant les dues interfícies.



En la secció "Recursos de contingut" del web trobareu el fitxer `InterficieConstants.java`, que incorpora un exemple d'utilització de la interfície `DiesSetmana`.

```
/* Fitxer: ExempleInterficie.java
   Descripció: Exemple de declaració d'interfícies i implementació en una classe
   Autor: Isidre Guixà
*/

import java.util.Date;
interface A
{
    Date DARRERA_CREACIO = new Date(0,0,1);    // 1-1-1900

    void metodeA ();
}

interface B extends A
{
    int VALOR_B = 20;

    void metodeB ();
}

class ExempleInterficie implements B
{
    private long b;
    private Date dataCreacio = new Date();

    public ExempleInterficie (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);
        ExempleInterficie obj = new ExempleInterficie(5);
        obj.metodeA();
        obj.metodeB();
        A pa = obj;
        B pb = obj;
    }
}
```



**Trobareu el fitxer**  
**ExempleInterficie.java en**  
**la secció "Recursos de contingut"**  
**del web.**

L'exemple serveix per il·lustrar uns quants punts:

- Comprovem que les dades membre de les interfícies 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 interfícies i utilitzar-les per fer referència a objectes de la classe `ExempleInterficie`.

## 2.6. Classes internes

Les classes **internes** són classes definides dins d'una altra classe, anomenada **classe contenidora**.

La utilitat de les classes internes rau en el fet que possibiliten:

- Agrupar classes que lògicament haurien d'estar juntes.
- Controlar la visibilitat dins altres classes.
- Agrupar la definició i la utilització de les classes.
- Definir classes simples d'ajuda o adaptació.
- Aconseguir un codi més clar que evita l'excés de classes petites que els usuaris dels paquets no necessiten conèixer.

Hi ha quatre tipus de classes internes:

- **Classes internes `static`**, també anomenades *classes internes de nivell superior* o *classes imbricades*. Aquest mecanisme també es pot aplicar a la creació d'interfícies.
- **Classes internes no `static`** (no aplicables a interfícies):
  - Classes internes membre, declarades com un membre de classe
  - Classes internes locals, declarades dins un bloc de codi d'un mètode de la classe
  - Classes internes anònimes, declarades en el moment de cridar el constructor.

El concepte anglès per referir-se a classes imbricades és *nested classes*.

En qualsevol cas:

- La màquina virtual Java no coneix l'existència de les classes internes, ja que el compilador les converteix en classes globals, de manera que genera un arxiu `.class` per a la classe contenidora i tants arxius `.class` com classes internes contingui, seguint una nomenclatura similar a `Contenidora$Interna.class`.
- La visibilitat entre les classes contenidora i interna és total, és a dir, els modificadors d'accés no tenen efecte.

- Les classes internes poden derivar d'una classe i implementar de diverses interfícies.

### 2.6.1. Classes i interfícies internes `static`

Les classes i interfícies internes `static` es declaren com si fossin un membre de la classe contenidora i amb el modificador `static`:

---

```
class Contenidora
{
    ...
    static class/interface Interna
    { <cosClasse/InterfaceInterna> }
    ...
}
```

---

Es verifica que:

- Només es poden declarar dins una classe normal o dins una altra classe interna `static`.
- Des de l'exterior en la classe contenidora es poden crear objectes de la classe interna sense necessitar cap objecte de la classe contenidora, amb la sintaxi següent:

---

```
Contenidora.Interna ci = new Contenidora.Interna (<paràmetres>);
```

---

Atès que s'accedeix a la classe interna de manera quasi independent (només es necessita el nom de la classe contenidora i no es necessita cap instància), també s'anomenen **classes internes de nivell superior**.

- Des de dins la classe interna només es pot accedir directament als membres `static` de la classe contenidora, ja que els objectes de la classe interna no estan associats a cap objecte en concret de la classe contenidora. L'accés a membres no `static` només és possible sobre objectes de la classe contenidora.
- Des de dins la classe contenidora es poden crear objectes de la classe interna amb la sintaxi següent:

---

```
Interna ci = new Interna (<paràmetres>);
```

---

**Exemple de classe que inclou una classe interna static**


---

```

/* Fitxer: ClasseInternaStatic.java
   Descripció: Classe contenidora de classe interna static
   Autor: Isidre Guixà
*/

class Contenidora
{
    private static int aaa = 5;

    public void metContenidora()
    {
        System.out.println ("Mètode de la classe contenidora: aaa = " + aaa);
    }

    public static class Interna
    {
        private int xxx = 10;

        public void metInterna()
        {
            System.out.println ("Mètode de la classe interna - xxx = " + xxx);
        }

        public static void main (String args[])
        {
            System.out.println ("Mètode main() de la classe interna.");
            System.out.println ("Accés a membre static de la classe contenidora: aaa = " + aaa);
//          metContenidora(); // No és té accés directe a membres no static de la classe contenidora
            Contenidora ce = new Contenidora();
            ce.metContenidora(); // Accés als membres static de la classe contenidora via objecte
        }
    }

    public static void main (String args[])
    {
        System.out.println ("Mètode main() de la classe contenidora.");
        Interna ci = new Interna();
        ci.xxx = 20;
        ci.metInterna();
    }
}

class AccesInterna
{
    public static void main(String args[])
    {
        Contenidora.Interna ci = new Contenidora.Interna();
        ci.metInterna();
    }
}

```

---



**Trobareu el fitxer**  
 ClasseInternaStatic.java  
 en la secció "Recursos de  
 contingut" del web.

**Podem comprovar la compilació i execució dels diversos mètodes main():**

---

```
G:\>javac ClasseInternaStatic.java
```

```
G:\>java -Dfile.encoding=cp850 Contenidora
Mètode main() de la classe contenidora.
Mètode de la classe interna - xxx = 20
```

```
G:\>java -Dfile.encoding=cp850 Contenidora$Interna
Mètode main() de la classe interna.
Accés a membre static de la classe contenidora: aaa = 5
Mètode de la classe contenidora: aaa = 5
```

```
G:\>java -Dfile.encoding=cp850 AccesInterna
Mètode de la classe interna - xxx = 10
```

---

### 2.6.2. Classes internes membre

**Les classes internes membre** són classes internes no `static` que es declaren com si fossin un membre de la classe contenidora:

---

```
class Contenidora
{
    ...
    class Interna
    { <cosClasseInterna> }
    ...
}
```

---

Es verifica que:

- Cada instància de la classe interna està associada a una instància de la classe contenidora i es té accés directe a tots els membres de l'objecte al qual s'està associat.

A causa del lligam entre objecte de classe interna i objecte associat de classe contenidora, els operadors `this` i `new` adquireixen una sintaxi especial:

- Per fer referència a l'objecte associat de la classe contenidora:

---

```
Contenidora.this
```

---

- Per crear un objecte de la classe interna:

---

```
Interna ci;
ci=variableContenidora.new Interna(<paràmetres>);
```

---

El nou objecte queda associat a l'objecte de la classe contenidora apuntat per `variableContenidora`. No és necessari indicar `variableContenidora` si la creació s'efectua dins un mètode no `static` de la classe contenidora, ja que llavors el nou objecte queda associat a l'objecte de la classe contenidora sobre el qual s'aplica el mètode.

- La classe interna no pot tenir cap membre `static`, en particular no pot tenir cap mètode `main()`, ni pot tenir noms comuns amb la classe contenidora.
- Es pot accedir a un objecte de la classe interna encara que no es tingui cap referència a l'objecte corresponent de la classe contenidora.
- Un objecte de la classe contenidora pot està enllaçat amb diferents objectes de la classe interna. Per tant, des de la classe contenidora, per accedir als membres de la classe interna cal utilitzar una referència a l'objecte de la classe interna.
- Una classe interna pot accedir a tots els membres de qualsevol altra classe interna dins la mateixa classe contenidora.

- Des de l'exterior a la classe contenidora, per crear objectes de la classe interna es necessita tenir un objecte de la classe contenidora, utilitzant la sintaxi següent:

---

```
Contenidora ce = new Contenidora(<paràmetres>);
```

```
Contenidora.Interna ci = ce.new Interna (<paràmetres>);
```

---

### Exemple de classe que inclou una classe interna membre

```
/* Fitxer: ClasseInternaMembre.java
   Descripció: Classe contenidora de classe interna membre
   Autor: Isidre Guixà
*/

class Contenidora
{
    private static int comptadorObjectesInterna;

    public void metContenidora()
    {
        System.out.println ("comptadorObjectesInterna = " + comptadorObjectesInterna);
        Interna cil = new Interna(100);
        Interna ci2 = new Interna(200);
        // cil i ci2 estan associats a l'objecte de Contenidora sobre el que s'executa el mètode
        cil.metInterna();
        ci2.metInterna();
        System.out.println ("comptadorObjectesInterna = " + comptadorObjectesInterna);
    }

    public class Interna
    {
        private int xxx;

        public Interna (int x)
        {
            xxx = x;
            comptadorObjectesInterna++;
        }

        public void metInterna()
        {
            System.out.println ("Sóc objecte de la classe interna - xxx = " + xxx);
        }
    }

    public static void main (String args[])
    {
        System.out.println ("Mètode main() de la classe contenidora.");
        Contenidora ce = new Contenidora();
        ce.metContenidora();
        Interna ci = ce.new Interna(50);
        // Tot objecte d'Interna està associat a un objecte de Contenidora
        ci.metInterna();
    }
}

class AccésInterna
{
    public static void main(String args[])
    {
        Contenidora ce = new Contenidora();
        Contenidora.Interna ci = ce.new Interna(5);
        ci.metInterna();
    }
}
```



Trobareu el fitxer  
ClasseInternaMembre.java  
en la secció "Recursos de  
contingut" del web.

Podem comprovar la compilació i execució de tots dos mètodes `main()`:

---

```
G:\>javac ClasseInternaMembre.java

G:\>java -Dfile.encoding=cp850 Contenidora
Mètode main() de la classe contenidora.
comptadorObjectesInterna = 0
Sóc objecte de la classe interna - xxx = 100
Sóc objecte de la classe interna - xxx = 200
comptadorObjectesInterna = 2
Sóc objecte de la classe interna - xxx = 50

G:\>java -Dfile.encoding=cp850 AccésInterna
Sóc objecte de la classe interna - xxx = 5
```

---

### 2.6.3. Classes internes locals

Les **classes internes locals** són classes internes `no static` que es declaren dins un bloc de codi (d'un mètode), en lloc de declarar-se com a membres de la classe contenidora:

---

```
class Contenidora
{
    ...
    [...] <nomMètode> (<llistaArguments>) // Declaració mètode
    {
        ...
        class Interna
        { <cosClasseInterna> }
        ...
    } // Fi del mètode de la classe contenidora
    ...
}
```

---

Es verifica el següent:

- Només són visibles i utilitzables dins el bloc de codi en el qual s'han definit.
- Tenen accés a tots els membres de la classe contenidora.
- Tenen accés a les variables locals i paràmetres del mètode contenidor declarats com a `final`.
- No poden dur cap modificador d'accés (`public`, `protected`, `private`) ni ser declarades `static`.



### Exemple de classe que inclou una classe interna local

```
/* Fitxer: ClasseInternaLocal.java
   Descripció: Classe contenidora de classe interna local
   Autor: Isidre Guixà
*/

class Contenedora
{
    private int x=10;

    public void metode(final char c)
    {
        final int y=20;

        {
            class Local
            {
                public int z=30;

                public void metLocal ()
                {
                    System.out.println("Dins metLocal - z = " + z);
                    System.out.println("Dins metLocal - y = " + y);
                    System.out.println("Dins metLocal - x = " + x);
                    System.out.println("Dins metLocal - c = " + c);
                    // Es pot accedir a la variable local y del mètode contenidor i
                    // al paràmetre c del mètode contenidor per estar declarades final.
                }
            }
            Local obj = new Local();
            obj.metLocal();
        }
        Local obj = new Local();    // Local no és visible en aquest punt
        obj.metLocal();
    }

    public static void main (String args[])
    {
        Contenedora ce = new Contenedora();
        ce.metode('S');
    }
}
```



Trobareu el fitxer  
ClasseInternaLocal.java  
en la secció "Recursos de  
contingut" del web.

Podem comprovar la compilació i execució del programa:

```
G:\>javac ClasseInternaLocal.java
```

```
G:\>java Contenedora
Dins metLocal - z = 30
Dins metLocal - y = 20
Dins metLocal - x = 10
Dins metLocal - c = S
```

### 2.6.4. Classes internes anònimes

Les **classes internes anònimes** són classes similars a les classes internes locals però sense nom.

Han de ser subclasse d'una altra classe (sempre ho poden ser de la classe `Object`) o implementar una interfície.

El fet de no tenir nom provoca el següent:

- Només se'n pot crear un objecte.

- No hi pot haver cap constructor, ja que els constructors s'han d'anomenar com la classe. Pot tenir, però, iniciadors.
- La definició de la classe s'ha d'efectuar de manera especial:
  - Indicant la classe base de la qual hereta (sense la paraula `extends`) i seguida de la definició de la classe anònima entre claus:

---

```
new <classeBase> (<paràmetres>) { <cosClasseInterna> }
```

---

Els paràmetres s'utilitzaran en la crida del constructor de la classe base que correspongui.

- Indicant la interfície que implementa (sense la paraula `implements`) i seguida de la definició de la classe anònima entre claus:

---

```
new <interfície> () { <cosClasseInterna> }
```

---

En aquest cas, la classe anònima deriva de la classe `Object`. El nom de la interfície va seguida de parèntesis, ja que el constructor de la classe `Object` no té arguments.

- Per les classes anònimes compilades, el compilador produeix fitxers amb un nom com `Contenidora$1.class`, assignant un nombre correlatiu a cada classe anònima declarada dins la classe contenidora.

És aconsellable que el codi de les classes anònimes sigui simple, per evitar la imbricació excessiva de codi i fer-les fàcils de llegir i interpretar. !!



Trobareu el fitxer `ClasseInternaAnonima.java` en la secció "Recursos de contingut" del web.

#### Exemple de classe que inclou una classe interna anònima

---

```
/* Fitxer: ClasseInternaAnonima.java
   Descripció: Classe contenidora de classe interna anònima
   Autor: Isidre Guixà
*/

class Contenidora
{
    private int x=10;

    public void metode(Object obj)
    {
        System.out.println(obj);
    }

    public static void main (String args[])
    {
        Contenidora ce = new Contenidora();
        ce.metode( new Object()
        {
            public int z=30;

            { System.out.println ("S'està creant un objecte de la classe anònima.");}

            public String toString()
            {
                return "El valor de z és " + z;
            }
        }
        );
    }
}
```

---

Podem comprovar la compilació i execució del programa:

```
G:\>javac ClasseInternaAnonima.java

G:\>dir /B Contenedora*
Contenedora$1.class
Contenedora.class

G:\>java -Dfile.encoding=cp850 Contenedora
S'està creant un objecte de la classe anònima.
El valor de z és 30
```


## 2.7. Classes genèriques

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 dona en el disseny de classes pensades per a la gestió de conjunts de dades (l·listes, 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 l·listes es diferencien en el tipus de dada que emmagatzemen (Integer, Persona, Date) però els mètodes a implementar en les tres l·listes són idèntics (afegirPerInici, afegirPelFinal, recorregut, extreurePrimerElement, extreureDarrerElement...) i, si no disposéssim de les classes genèriques, hauríem d'implementar tres l·listes diferents amb la repetició consegüent de codi idèntic.

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...>
{
    ...
}
```

El llenguatge C++ també incorpora les classes genèriques conegudes com a *template*.

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 <nomTipusConcret1,nomTipusConcret2...> obj;  
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 (...);
```

---

Aquest tipus de definicions, en què la classe és genèrica, són molt problemàtiques, com es veu al final de l'exemple següent.

#### **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.



Trobareu el fitxer `ExempleClasseGenerica.java` a en la secció "Recursos de contingut" del web.

---

```
/* Fitxer: ExempleClasseGenerica.java
   Descripció: Exemple de disseny i utilització de classe genèrica amb 2 tipus genèrics.
   Autor: Isidre Guixà
*/

import java.util.Date;

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();
    }

    public T1 getX1() { return x1; }

    public T2 getX2() { return x2; }

    public static void main (String args[])
    {
        ExempleClasseGenerica <Integer, Float> obj1 =
            new ExempleClasseGenerica <Integer,Float> (new Integer(20), new Float(42.45));
        ExempleClasseGenerica <Double, Double> obj2 =
            new ExempleClasseGenerica <Double,Double> (new Double(4.32), new Double(7.45));
        ExempleClasseGenerica obj3 = new ExempleClasseGenerica (new Integer(22), new Date());
        ExempleClasseGenerica obj4 = new ExempleClasseGenerica (50, "Hola");
        System.out.println(obj1.toString());
        System.out.println(obj2.toString());
        System.out.println(obj3.toString());
        System.out.println(obj4.toString());
    }
}
```

---

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 `T2`. 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 dona un avís:

---

```
G:\>javac ExempleClasseGenerica.java
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:

---

```
G:\>javac -Xlint:unchecked ExempleClasseGenerica.java
ExempleClasseGenerica.java:34: warning: [unchecked] unchecked call to ExempleClasseGenerica(T1,T2) as a member of the raw type ExempleClasseGenerica
    ExempleClasseGenerica obj3 = new ExempleClasseGenerica (new Integer(22), new Date());
                                ^
ExempleClasseGenerica.java:35: warning: [unchecked] unchecked call to ExempleClasseGenerica(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:

---

```
G:\>java ExempleClasseGenerica
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:

---

```
G:\>javac ExempleClasseGenerica.java
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.

```
/* Fitxer: MetodeAmbClasseGenerica.java
   Descripció: Exemple de mètode que admet objectes d'una classe genèrica
   Autor: Isidre Guixà
*/

import java.util.Date;

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);
    }
}
```



Trobareu el fitxer `MetodeAmbClasseGenerica.java` en la secció "Recursos de contingut" del web.

L'execució del mètode `main()` dóna el resultat:

```
G:\>java MetodeAmbClasseGenerica
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 o que implementi una interfície X. En qualsevol cas, la sintaxi és:

```
[public] [final|abstract] NomClasse <T1 extends X, T2 extends Y...>
{
    ...
}
```

Veiem que s'utilitza la paraula reservada `extends` independentment que X sigui una classe o una interfície.

### 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.



Trobareu el fitxer `TaulaValorsNumerics.java` en la secció "Recursos de contingut" del web.

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.

---

```

/* Fitxer: TaulaValorsNumerics.java
   Descripció: Exemple complet de classe genèrica
   Autor: Isidre Guixà
*/

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 genèric 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:

---

```
G:\>java -Dfile.encoding=cp850 TaulaValorsNumerics
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
```

---

### 3. Classes fonamentals

El llenguatge Java proporciona una gran quantitat de paquets de classes 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).

#### 3.1. L'API de Java. *Frameworks*

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 ells 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, i la seva consulta resulta imprescindible en qualsevol desenvolupament.

L'API de Java és immensa. La versió actual (1.6) conté, segons la seva documentació, 203 paquets que inclouen 3.973 elements entre classes, interfícies, excepcions... Per tant, el coneixement en profunditat de l'API no és una tasca trivial, però 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, ajudi a obtenir la solució definitiva.

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).

#### L'API d'un llenguatge

L'API d'un llenguatge de programació és el conjunt més o menys ampli de biblioteques o biblioteques contenidores de tipus de dades, accions i funcions que els llenguatges de programació proporcionen per ser utilitzades per la comunitat de programadors.

El mot API és l'acrònim del terme anglès *application programming interface*, traduït al català per *interfície de programació d'aplicacions*.

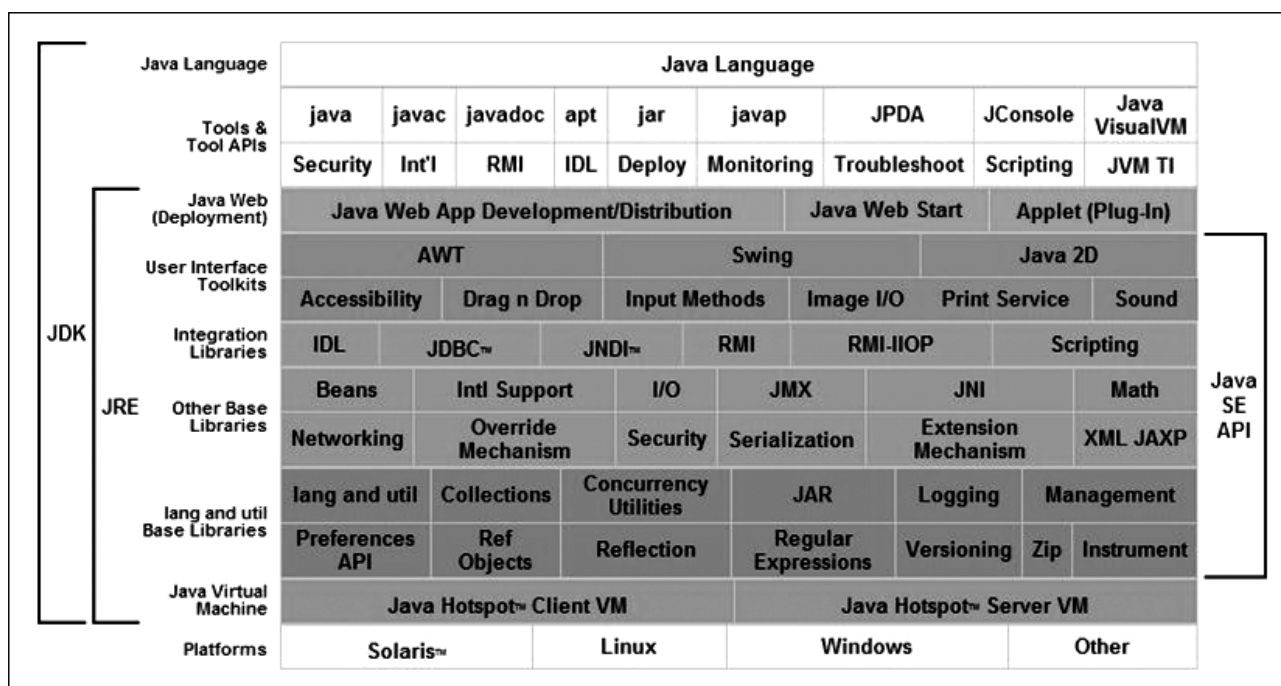


En la secció "Adreces d'interès" del web podeu trobar la documentació de Java referent *Java Platform API Specification*.

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 usals 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 5 que mostra tots els components de la plataforma Java.

Figura 5. Components de la plataforma Java Se 6



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)...

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.

Dins l'API de Java podem considerar diversos conjunts de classes coneguts com a *frameworks*, ja que estan pensats per donar resposta a una problemàtica determinada. Però també tenim *frameworks* en Java desenvolupats per tercers. La taula 10 ens presenta alguns *frameworks* de Java, i ens indica si és inclòs en l'API de Java o si pertany a tercers.

Taula 10. Alguns *frameworks* de Java

Framework	Utilitat	Propietari/responsable
AWT ( <i>abstract window toolkit</i> )	Interfícies gràfiques d'usuari en Java	∈ API Java
Swing		
JCF ( <i>Java collections framework</i> )	Arquitectura unificada per representar i manipular col·leccions	
Input methods framework	Introducció de text des de diversos dispositius (teclat, escriptura a mà, reconeixement de veu) i en qualsevol llenguatge humà	
Apache struts	Aplicacions web sota el patró de disseny MVC en la plataforma Java EE	Apache Software Foundation
JMF ( <i>Java media framework</i> )	Tractament de continguts multimèdia	Sun Microsystems
JAI ( <i>Java advanced imaging</i> )	Processament d'imatges en Java	Sun Microsystems
JUnit	Verificació de classes Java de manera controlada, per poder avaluar el comportament de cada mètode d'una classe	Open Source
Spring	Desenvolupament d'aplicacions Java	
Hibernate	Mapatge d'atributs entre un SGBDR i el model d'objectes d'una aplicació mitjançant fitxers declaratius XML	
ADF ( <i>application development framework</i> )	Desenvolupament d'aplicacions empresarials	Oracle
Canigó	Arquitectura comuna per desenvolupar aplicacions Java EE	Generalitat de Catalunya

## 3.2. Representació i manipulació de col·leccions

Una col·lecció és una agrupació d'elements (objectes) en la qual cal 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, i sempre s'ha distingit entre l'accés per posició i l'accés per clau.

El llenguatge Java, conscient de la importància de l'organització de la informació, 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 realitzen càlculs (cerques, ordenacions...) en els objectes de les implementacions.

### 3.2.1. 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*,

#### La gestió de fitxers

Recordem els quatre tipus d'accés en la gestió de fitxers:

- Accés seqüencial per posició
- Accés directe per posició
- Accés seqüencial per clau
- Accés directe per clau

Diverses combinacions d'aquests tipus provocaven l'existència de diferents tipologies de sistemes gestors de fitxers (seqüencials, relatius, indexats...).

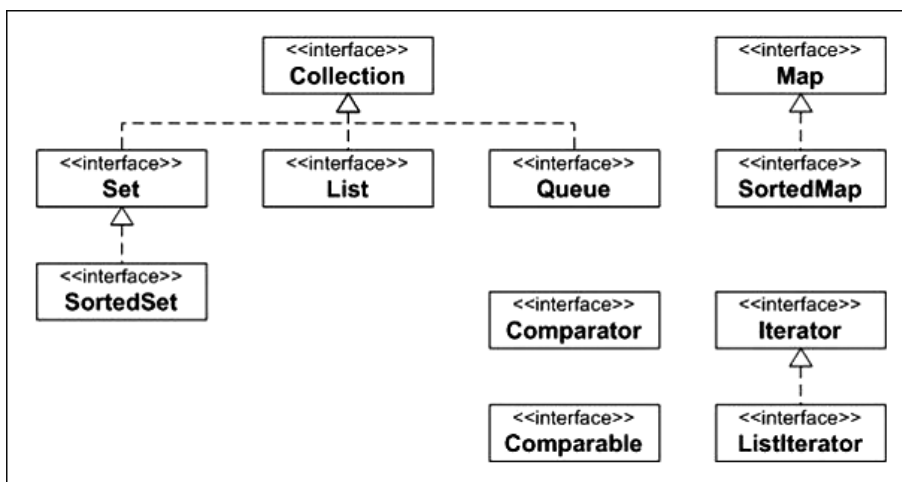
`SortedSet`, `Map` i `SortedMap`, coneegudes normalment com *interfícies del JCF*.

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 6 mostra les interfícies del *framework* de col·leccions de Java amb la jerarquia que hi ha entre elles.

Figura 6. 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);
}
```

#### Mètodes optional en interfícies

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*.

L'explicació detallada sobre la gestió d'excepcions la trobareu en l'apartat "Excepcions" d'aquest nucli d'activitat.

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();
    Object 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:

```
public interface ListIterator<E> extends Iterator<E>{
    boolean hasNext();
    Object next();
    boolean hasPrevious();
    Object previous();
    int nextIndex();
    int previousIndex();
    void remove();
    void set(java.lang.Object);
    void add(java.lang.Object);
}
```

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  $x$ , 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:

```
public interface Comparable<T> {
    int compareTo(T o);
}
```

El mètode `compareTo()` compara l'objecte sobre el qual s'aplica el mètode amb l'objecte `o` 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 úni-



Trobareu l'explicació detallada sobre la gestió d'excepcions en l'apartat "Excepcions" d'aquest nucli d'activitat.



cament 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` dissenyada fins ara. 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()`:

```
abstract class Persona implements Comparable
{
    ...
    public final int compareTo (Object obj)
    {
        if (obj instanceof Persona) return dni.compareTo(((Persona)obj).dni);
        throw new ClassCastException(); // Instrucció que genera una excepció
    }
    ...
}
```

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 demostra:

```
/* Fitxer: ProvaPersona08.java
   Descripció: Classe per comprovar l'ordenació de taules de Persona (versió Persona08)
   Autor: Isidre Guixà
*/

import java.util.Arrays;

public class ProvaPersona08
{
    public static void main(String args[])
    {
        Persona t[] = new Persona[6];
        t[0] = new Alumne("999999999", "Anna", 20);
        t[1] = new Alumne("000000000", "Pep", 33, 'm');
        t[2] = new Alumne("222222222", "Maria", 40, 's');
        t[3] = new Alumne("666666666", "Àngel", 22);
        t[4] = new Alumne("111111111", "Joanna", 25, 'M');
        t[5] = new Alumne("555555555", "Teresa", 30, 'S');

        System.out.println("Contingut inicial de la taula:");
        for (int i=0; i<t.length; i++) System.out.println("    "+t[i]);
        Arrays.sort(t);
        System.out.println("Contingut de la taula després d'haver estat ordenada:");
        for (int i=0; i<t.length; i++) System.out.println("    "+t[i]);
    }
}
```



Trobareu el fitxer `Persona08.java` (evolució del disseny de la classe `Persona` fins al punt actual) i el fitxer `ProvaPersona08.java` en la secció "Recursos de contingut" del web.

La seva execució dona el resultat:

---

```
G:>java -Dfile.encoding=cp850 ProvaPersona08
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. Mitjà
    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. Mitjà
    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.



Trobareu el fitxer `ProvaComparator.java` en la secció "Recursos de contingut" del web.

---

```

/* Fitxer: ProvaComparator.java
   Descripció: Definició d'una classe que implementa la interfície Comparator
               per establir ordre entre objectes de classes tant dispars com:
               Persona, Date, Double i Short.
   Autor: Isidre Guixà
*/

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);
        t[4] = new Short((short)22);
        t[5] = new Date(109, 0, 1);
        System.out.println("Contingut inicial de la taula:");
        for (int i=0; i<t.length; i++)
            System.out.println("    "+t[i].getClass().getName()+" - "+t[i]);
        Arrays.sort(t, new xComparator());
        System.out.println("Contingut de la taula després d'haver estat ordenada:");
        for (int i=0; i<t.length; i++)
            System.out.println("    "+t[i].getClass().getName()+" - "+t[i]);
    }
}

/* La següent classe defineix l'ordre entre objectes de les classes Persona, Date, Double i Short
   de manera que: Date < Double < Persona < Short
   Entre objectes de la mateixa classe, l'ordre el defineix l'ordre natural definit per la
   implementació del mètode compareTo() de la interfície Comparable
*/
class xComparator implements Comparator <Object>
{
    public int compare (Object o1, Object o2)
    {
        if (o1 instanceof Date)
            if (o2 instanceof Date) return ((Date)o1).compareTo((Date)o2);
            else return -1;
        else if (o1 instanceof Double)
            if (o2 instanceof Date) return 1;
            else if (o2 instanceof Double) return ((Double)o1).compareTo((Double)o2);
            else return -1;
        else if (o1 instanceof Persona)
            if (o2 instanceof Date || o2 instanceof Double) return 1;
            else if (o2 instanceof Persona) return ((Persona)o1).compareTo((Persona)o2);
            else return -1;
        else if (o1 instanceof Short)
            if (o2 instanceof Short) return ((Short)o1).compareTo((Short)o2);
            else return 1;
        else throw new ClassCastException(); // Instrucció que genera una excepció
    }
}

```

---

**L'execució del programa dona el resultat:**

---

```

G:\>java -Dfile.encoding=cp850 ProvaComparator
Contingut inicial de la taula:
Alumne - Dni: 99999999 - Nom: Anna - Edat: 20 - Nivell: ???
java.util.Date - Sun Apr 05 18:26:00 CEST 2009
Alumne - Dni: 22222222 - Nom: Maria - Edat: 40 - Nivell: Cicle F. Superior
java.lang.Short - 22
java.lang.Double - 33.33
java.util.Date - Thu Jan 01 00:00:00 CET 2009
Contingut de la taula després d'haver estat ordenada:
java.util.Date - Thu Jan 01 00:00:00 CET 2009
java.util.Date - Sun Apr 05 18:26:00 CEST 2009
java.lang.Double - 33.33
Alumne - Dni: 22222222 - Nom: Maria - Edat: 40 - Nivell: Cicle F. Superior
Alumne - Dni: 99999999 - Nom: Anna - Edat: 20 - Nivell: ???
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 algebraïques 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:

---

```
public interface SortedSet<E> extends Set<E> {  
    // Range-view  
    SortedSet<E> subSet(E fromElement, E toElement);  
    SortedSet<E> headSet(E toElement);  
    SortedSet<E> tailSet(E fromElement);  
  
    // Endpoints  
    E first();  
    E last();  
  
    // Comparator access  
    Comparator<? super E> comparator();  
}
```

---

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:

---

```
public interface List<E> extends Collection<E> {
    // Positional access
    E get(int index);
    E set(int index, E element);    //optional
    void add(int index, E element); //optional
    E remove(int index);           //optional
    boolean addAll(int index, Collection<? extends E> c); //optional

    // Search
    int indexOf(Object o);
    int lastIndexOf(Object o);

    // Iteration
    ListIterator<E> listIterator();
    ListIterator<E> listIterator(int index);

    // Range-view
    List<E> subList(int from, int to);
}
```

---

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`:

---

```
public interface Queue<E> extends Collection<E> {
    E element();
    boolean offer(E e);
    E peek();
    E poll();
    E remove();
}
```

---

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 11 ens mostra la classificació dels mètodes segons aquests criteris.

Taula 11. Classificació dels mètodes de la interfície `Queue` segons la reacció a un error

Operació	Mètode que provoca excepció	Mètode que retorna un valor especial
Encuar	<code>boolean add (E e)</code>	<code>boolean offer(E e)</code>
Desencuar	<code>E remove()</code>	<code>E poll()</code>
Inici	<code>E element()</code>	<code>E peek()</code>

## Interfícies `Map` i `SortedMaps`

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:

```
public interface Map<K,V> {

    // Basic operations
    V put(K key, V value);    //optional
    V get(Object key);
    V remove(Object key);    //optional
    boolean containsKey(Object key);
    boolean containsValue(Object value);
    int size();
    boolean isEmpty();

    // Bulk operations
    void putAll(Map<? extends K, ? extends V> m); // optional
    void clear();                               // optional

    // Collection Views
    public Set<K> keySet();
    public Collection<V> values();
    public Set<Map.Entry<K,V>> entrySet();

    // Interface for entrySet elements
    public interface Entry {
        K getKey();
        V getValue();
        V setValue(V value);
    }
}
```

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`:

---

```
public interface SortedMap<K, V> extends Map<K, V> {
    // Range-view
    SortedMap<K, V> subMap(K fromKey, K toKey);
    SortedMap<K, V> headMap(K toKey);
    SortedMap<K, V> tailMap(K fromKey);

    // Endpoints
    K firstKey();
    K lastKey();

    // Comparator access
    Comparator<? super K> comparator();
}
```

---

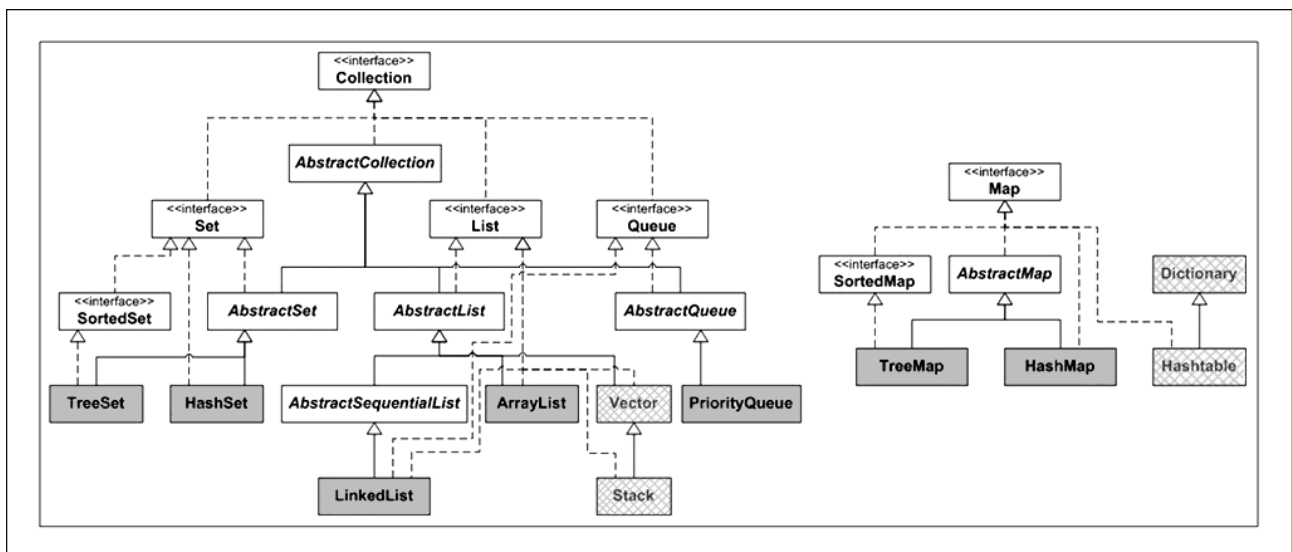
### 3.2.2. Implementacions

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 7 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 7 presenta les classes no abstractes amb un fons no blanc per identificar-les de manera ràpida.

Figura 7. Jerarquia d'interfícies i implementacions més usals a partir de les interfícies `Collection` i `Map`



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 *framework*, de manera que mantenen els mètodes originals i, a més, implementen les interfícies `List` (classes `Vector` i `Stack`) i `Map` (classe `Hashtable`). La figura 7 mostra les classes històriques amb un fons trametat 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 propietat `length` 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



Trobareu el fitxer `VectorValorsNumerics.java` en la secció "Recursos de contingut" del web.

mitjana dels valors continguts, comparació de mitjana, capacitat i nombre d'elements vers altres vectors i implementació del mètode toString().

---

```
/* Fitxer: VectorValorsNumerics.java
   Descripció: Exemple d'utilització de la classe Vector
   Autor: Isidre Guixà
*/

import java.util.Vector;

class VectorValorsNumerics extends Vector<Number>
{
    public VectorValorsNumerics(int capacitatInicial, int increment)
    {
        super(capacitatInicial, increment);
    }

    public double mitja ()
    {
        double suma=0;
        int valorsNoNuls=0;
        for (int i=0; i<size(); i++)
        {
            suma = suma + get(i).doubleValue();
            valorsNoNuls++;
        }
        if (valorsNoNuls==0) 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:

---

```

G:\>java VectorValorsNumerics
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.



Trobareu el fitxer `HashtablePersones.java` en la secció "Recursos de contingut" del web.

#### 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
   Descripció: Exemple d'utilització de la classe Hashtable
   Autor: Isidre Guixà
*/

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;

        a = new Alumne("999999999","Anna",20);
        htp.put(a.getDni(), a);
        a = new Alumne("000000000","Pep",33,'m');
        htp.put(a.getDni(), a);
        a = new Alumne("555555555","Teresa",30,'S');
        htp.put(a.getDni(), a);
        a = new Alumne("22222222","Maria",40,'s');
        htp.put(a.getDni(), a);
        a = new Alumne("66666666","Àngel",22);
        htp.put(a.getDni(), a);
        a = new Alumne("11111111","Joanna",25,'M');
        htp.put(a.getDni(), a);
        a = new Alumne("55555555","Maria Teresa",30,'S');
        htp.put(a.getDni(), a);

        System.out.println("Contingut de la Hashtable via enumeracions:");
        ep = htp.elements();
        es = htp.keys();
        while (es.hasMoreElements())
            System.out.println (es.nextElement() + " >>> " + ep.nextElement());

        System.out.println();
        System.out.println("Contingut de la Hashtable cridant el mètode toString:");
        System.out.println(htp);

        System.out.println();
        System.out.println("Efectuem alguna recerca de persones per la clau \"dni\":");
        dni = "66666666";
        System.out.println("Dni : " + dni + " >>> " + htp.get(dni));
        dni = "6666";
        System.out.println("Dni : " + dni + " >>> " + htp.get(dni));
    }
}

```

L'execució del mètode `main()` dona el resultat:

---

```
G:\>java -Dfile.encoding=cp850 HashtablePersones
Contingut de la Hashtable via enumeracions:
66666666 >>> Dni: 66666666 - Nom: Àngel - Edat: 22 - Nivell: ???
00000000 >>> Dni: 00000000 - Nom: Pep - Edat: 33 - Nivell: Cicle F. Mitjà
11111111 >>> Dni: 11111111 - Nom: Joanna - Edat: 25 - Nivell: Cicle F. Mitjà
22222222 >>> Dni: 22222222 - Nom: Maria - Edat: 40 - Nivell: Cicle F. Superior
99999999 >>> Dni: 99999999 - Nom: Anna - Edat: 20 - Nivell: ???
55555555 >>> Dni: 55555555 - Nom: Maria Teresa - Edat: 30 - Nivell: Cicle F. Superior

Contingut de la Hashtable cridant el mètode toString:
{66666666=Dni: 66666666 - Nom: Àngel - Edat: 22 - Nivell: ???, 00000000=Dni: 00000000 - Nom: Pep
- Edat: 33 - Nivell: Cicle F. Mitjà, 11111111=Dni: 11111111 - Nom: Joanna - Edat: 25 - Nivell: Ci
cle F. Mitjà, 22222222=Dni: 22222222 - Nom: Maria - Edat: 40 - Nivell: Cicle F. Superior, 9999999
9=Dni: 99999999 - Nom: Anna - Edat: 20 - Nivell: ???, 55555555=Dni: 55555555 - Nom: Maria Teresa
- Edat: 30 - Nivell: Cicle F. Superior}

Efectuem alguna recerca de persones per la clau "dni":
Dni : 66666666 >>> Dni: 66666666 - Nom: Àngel - Edat: 22 - Nivell: ???
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.

### 3.2.3. 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 enu-

meració en aquest material. És aconsellable fer una ullada a la documentació corresponent de Java.

La taula 12 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 12. Recull de mètodes de les classes `Arrays` i `Collections`

Mètode	Objectiu
<code>Arrays.sort(T[])</code>	Ordenar taules clàssiques de Java
<code>Arrays.toString(T[])</code>	Obtenir una representació <code>String</code> dels continguts d'una taula
<code>Arrays.fill(T[], ...)</code>	Emplenar totes les cel·les d'una taula amb valors determinats
<code>Arrays.equals((T[], T[], ...)</code>	Comparar dues taules
<code>Arrays.binarySearch(T[], T)</code>	Efectuar una cerca dicotòmica en una taula ordenada
<code>Collections.sort(List&lt;T&gt;)</code>	Ordenar objectes <code>List</code>
<code>Collections.binarySearch(List&lt;T&gt;, Object)</code>	Efectuar una cerca dicotòmica en un objecte <code>List</code> ordenat
<code>Collections.reverse(List&lt;T&gt;)</code>	Invertir l'ordre dels elements en un objecte <code>List</code>
<code>Collections.max(Collection&lt;E&gt;)</code>	Trobar el màxim valor d'un objecte <code>Collection</code>
<code>Collections.min(Collection&lt;E&gt;)</code>	Trobar el mínim valor d'un objecte <code>Collection</code>

### 3.3. 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 les funcions (si estem en programació estructurada i modular) o els mètodes (si estem en programació orientada a objectes) 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.

Com a exemple, el control que cal efectuar en el llenguatge C en intentar obrir un fitxer:

---

```
FILE *f;
...
f = fopen ("c:\arxiu.txt", "r+");
if (f == NULL)
{
    printf ("No s'ha pogut obrir l'arxiu");
    /* Tractament que correspongui */
}
else /* Tractament que correspongui */
```

---

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 qual s'ha produït l'error.

El model de gestió d'excepcions que proporciona el llenguatge Java (idèntic al del llenguatge C++) 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. (❗)

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 mala-

ment en la càrrega d'un servei d'un proveïdor, quan deixa de respondre un canal d'entrada/sortida...

- Les excepcions corresponen a situacions excepcionals que els programes es poden trobar en temps d'execució, incloent-hi, 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'`Error` 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.

I, en referència als mecanismes de gestió d'excepcions que proporciona Java, ens cal saber com es gestionen les excepcions, com es generen (*llançen*, 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.

### 3.3.1. Captura

El llenguatge Java proporciona el mecanisme `try - catch` per capturar una excepció i definir l'actuació que correspongui.

El mecanisme `try - catch` 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 `catch` 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();
}
```

F. Yellin (1995). *Low Level Security in Java*. Fourth International WWW Conference.

#### 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
   Descripció: Exemple d'excepció de la classe RuntimeException
               provocada per error de programació i no capturada
   Autor: Isidre Guixà
*/

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:

```
G:\>java -Dfile.encoding=cp850 Excepcio01
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.



Trobareu el fitxer  
Excepcio01.java en la secció  
"Recursos de contingut" del web.

### 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.



Trobareu el fitxer  
Excepcio02.java en la secció  
"Recursos de contingut" del web.

```
/* Fitxer: Excepcio02.java
   Descripció: Exemple d'excepció de la classe RuntimeException
               provocada per error de programació i capturada
   Autor: Isidre Guixà
*/

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 sortir de límits!!!");
        }
        System.out.println("Final del programa");
    }
}
```

L'execució del programa mostra:

```
G:\>java -Dfile.encoding=cp850 Excepcio02
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.



Trobareu el fitxer  
`Excepcio03.java` en la secció  
"Recursos de contingut" del web.

```
/* Fitxer: Excepcio03.java
   Descripció: Exemple d'excepció provocada per error de programació
               amb intent erroni de captura i amb exemple de bloc finally
   Autor: Isidre Guixà
*/

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:

```
G:\>java -Dfile.encoding=cp850 Excepcio03
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()`.



Trobareu el fitxer  
`Excepcio04.java` en la secció  
"Recursos de contingut" del web.

```
/* Fitxer: Excepcio04.java
   Descripció: Exemple de propagació de l'excepció cap el mètode superior
               en cas de no ser tractada en el propi mètode i, avortament
               del programa si la propagació arriba fins el mètode main()
               sense cap gestió.
   Autor: Isidre Guixà
*/

public class Excepcio04
{
    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.");
    }

    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");
    }

    public static void main(String args[])
    {
        System.out.println("Iniciem el programa i anem a executar met01");
        met01();
        System.out.println("Tornem a estar en el main després de finalitzar met01");
    }
}
```

L'execució d'aquest programa dóna el resultat:

```
G:\>java -Dfile.encoding=cp850 Excepcio04
Iniciem el programa i anem a executar met01
Entrem en el mètode met01 i anem a executar met02
Posició 0 : Hola
Posició 1 : Adéu
Posició 2 : Fins demà
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException:
3
    at Excepcio04.met02(Excepcio04.java:15)
    at Excepcio04.met01(Excepcio04.java:22)
    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.



Trobareu el fitxer  
`Excepcio05.java` en la secció  
"Recursos de contingut" del web.

---

```
/* Fitxer: Excepcio05.java
   Descripció: Exemple de propagació de l'excepció cap el mètode superior
               trobant un mètode abans d'arribar al mètode main() que captura
               l'excepció.
   Autor: Isidre Guixà
*/

public class Excepcio05
{
    static void met03()
    {
        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 met03 s'ha acabat.");
    }

    static void met02()
    {
        System.out.println("Entrem en el mètode met02 i anem a executar met03");
        met03();
        System.out.println("Tornem a estar en met02 després de finalitzar met03");
    }

    static void met01()
    {
        try
        {
            System.out.println("Entrem en el mètode met01 i anem a executar met02");
            met02();
            System.out.println("Tornem a estar en met01 després de finalitzar met02");
        }
        catch (ArrayIndexOutOfBoundsException e)
        {
            System.out.println("El programador estava a la lluna... S'ha sortir de límits!!!");
        }
    }

    public static void main(String args[])
    {
        System.out.println("Iniciem el programa i anem a executar met01");
        met01();
        System.out.println("Tornem a estar en el main després de finalitzar met01");
    }
}
```

---

L'execució d'aquest programa dona la sortida:

---

```
G:\>java -Dfile.encoding=cp850 Excepcio05
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
El programador estava a la lluna... S'ha sortir de límits!!!
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.

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 d'obtenció d'informació d'una excepció

En el darrer exemple (fitxer `Excepcio05.java`), 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.

---

```
static void met01()
{
    try
    {
        System.out.println("Entrem en el mètode met01 i anem a executar met02");
        met02();
        System.out.println("Tornem a estar en met01 després de finalitzar met02");
    }
    catch (ArrayIndexOutOfBoundsException e)
    {
        System.out.println("Estem dins el bloc catch que ha capturat l'excepció.");
        System.out.println("Informació que dona el mètode getMessage():");
        System.out.println(e.getMessage());
        System.out.println("Informació que dona el mètode printStackTrace():");
        e.printStackTrace();
        System.out.println("Informació que dona el mètode toString():");
        System.out.println(e);
    }
}
```

---

L'execució del programa ens mostra la informació que proporciona cada mètode:

---

```
G:\>java -Dfile.encoding=cp850 Excepcio06
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():
3
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 dona 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.



**Trobareu el fitxer**  
`Excepcio06.java`, evolució del  
fitxer `Excepcio05.java`, amb la  
modificació introduïda, en la  
secció "Recursos de contingut" del  
web.

### 3.3.2. 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`.



Així, 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 8 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 8. Informació, en la documentació dels mètodes, relativa a la gestió d'excepcions

**FileOutputStream**

```
public FileOutputStream(String name)
    throws FileNotFoundException
```

Excepció amb obligatorietat de ser gestionada

Creates an output file stream to write to the file with the specified name. A new `FileDescriptor` object is created to represent this file connection.

First, if there is a security manager, its `checkWrite` method is called with `name` as its argument.

If the file exists but is a directory rather than a regular file, does not exist but cannot be created, or cannot be opened for any other reason then a `FileNotFoundException` is thrown.

**Parameters:**

`name` - the system-dependent filename

**Throws:**

`FileNotFoundException` - if the file exists but is a directory rather than a regular file, does not exist but cannot be created, or cannot be opened for any other reason

`SecurityException` - if a security manager exists and its `checkWrite` method denies write access to the file.

**See Also:**

`SecurityManager.checkWrite(java.lang.String)`

Excepcions llançades pel mètode

**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:

---

```
unreported exception nomExcepció; must be caught or declared to be thrown
```

---

#### **Exemple de programa que no gestiona les excepcions de gestió obligatòria**

Considerem el següent fitxer `.java`:

---

```
/* Fitxer: Excepcio07.java
   Descripció: Exemple de programa que no gestiona excepcions susceptibles
               de ser llançades per un mètode i que no són RuntimeException.
               El compilador no compila el fitxer.
   Autor: Isidre Guixà
*/
import java.io.*;

public class Excepcio07
{
    public static void main (String args[])
    {
        FileOutputStream f = new FileOutputStream ("C:\\arxiu.txt");
        f.close();
    }
}
```

---

La seva compilació detecta els errors següents:

---

```
G:\>javac Excepcio07.java
Excepcio07.java:13: unreported exception java.io.FileNotFoundException; must be caught or
declared to be thrown
    FileOutputStream f = new FileOutputStream ("C:\\arxiu.txt");
                        ^
Excepcio07.java:14: unreported exception java.io.IOException; must be caught or declared to be thrown
        f.close();
            ^
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.

Així, doncs, 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:

---

```
[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 delegació de gestió d'excepcions

El fitxer `.java` següent mostra un mètode que en el seu interior crida els mètodes `FileOutputStream` (`String`)`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.

```
/* Fitxer: Excepcio08.java
   Descripció: Exemple de mètode que opta per no capturar excepcions d'obligada gestió
               tot delegant la gestió als mètodes que l'invocuin.
   Autor: Isidre Guixà
*/

import java.io.*;

class Excepcio08
{
    public void metodeAmbClausulaThrows (String nomFitxer)
        throws FileNotFoundException, IOException
    {
        FileOutputStream f = new FileOutputStream (nomFitxer);
        f.close();
        System.out.println ("El metodeAmbClausulaThrows ha finalitzat.");
    }
}
```

Fixem-nos que com que l'excepció `FileNotFoundException` és subclasse de l'excepció `IOException`, hauríem pogut indicar, a la clàusula `throws`, únicament l'excepció `IOException` i el compilador hauria traduït el programa, però, d'aquesta manera, els mètodes que cridin `metodeAmbClausulaThrows()` 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.

```
/* Fitxer: Prova01Excepcio08.java
   Descripció: Exemple de mètode que invoca un mètode amb clàusula "throws" sense gestionar
               les excepcions indicades a la clàusula. El compilador no tradueix el fitxer.
   Autor: Isidre Guixà
*/

import java.io.*;

public class Prova01Excepcio08
{
    public static void main (String args[])
    {
        Excepcio08 exc = new Excepcio08();
        exc.metodeAmbClausulaThrows("c:\\arxiu.txt");
        System.out.println ("Hem tornat del metodeAmbClausulaThrows");
        System.out.println ("El programa ha finalitzat.");
    }
}
```

Veiem l'informe del compilador:

```
G:\>javac Prova01Excepcio08.java
Prova01Excepcio08.java:13: unreported exception java.io.FileNotFoundException; must be caught or
declared to be thrown
        exc.metodeAmbClausulaThrows("c:\\arxiu.txt");
            ^
1 error
```



#### Trobareu els fitxers

Excepcio08.java,  
Prova01Excepcio08.java,  
Prova02Excepcio08.java i  
Prova03Excepcio08.java en  
la secció "Recursos de contingut"  
del web d'aquest crèdit.

Fixem-nos que el compilador només ens informa de la primera excepció indicada a la clàusula `throws` del mètode `AmbClausulaThrows()` que no és gestionada.

Millorem el codi anterior gestionant l'excepció `FileNotFoundException`:

---

```
/* Fitxer: Prova02Excepcio08.java
   Descripció: Exemple de mètode que invoca un mètode amb clàusula "throws" sense gestionar
               totes les excepcions indicades a la clàusula. El compilador no tradueix el fitxer.
   Autor: Isidre Guixà
*/
import java.io.*;

public class Prova02Excepcio08
{
    public static void main (String args[])
    {
        Excepcio08 exc = new Excepcio08();
        try
        {
            exc.metodeAmbClausulaThrows("c:\\arxiu.txt");
        }
        catch (FileNotFoundException e)
        {
            System.out.println("S'ha capturat l'excepció FileNotFoundException");
        }
        System.out.println("El programa ha finalitzat.");
    }
}
```

---

El programador ja ha gestionat l'excepció `FileNotFoundException` però s'ha oblidat de l'excepció `IOException` i l'informe del compilador és clar:

---

```
G:\>javac Prova02Excepcio08.java
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
   Descripció: Exemple de mètode que invoca un mètode amb clàusula "throws" gestionant totes
               les excepcions indicades a la clàusula. El compilador sí tradueix el fitxer.
   Autor: Isidre Guixà
*/
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.

---

```
G:\>java -Dfile.encoding=cp850 Prova03Excepcio08
La crida del programa ha d'indicar un paràmetre:
    nomArxiu amb el corresponent camí de directoris.

G:\>java -Dfile.encoding=cp850 Prova03Excepcio08 c:\arxiu.txt
El metodeAmbClausulaThrows ha finalitzat.
El programa ha finalitzat.

G:\>attrib +r c:\arxiu.txt

G:\>java -Dfile.encoding=cp850 Prova03Excepcio08 c:\arxiu.txt
S'ha capturat l'excepció FileNotFoundException, amb informació:
java.io.FileNotFoundException: c:\arxiu.txt (Acceso denegado)
El programa ha finalitzat.

G:\>java -Dfile.encoding=cp850 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.
```

---

### 3.3.3. 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.

```
/* Fitxer: Excepcio09.java
   Descripció: Exemple de llançament d'excepció d'obligada gestió en un mètode
   Autor: Isidre Guixà
*/

public class Excepcio09
{
    static void verificaLengthTaula (int n, String t[]) throws Exception
    /* 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 */
    {
        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:

```
G:\>java -Dfile.encoding=cp850 Excepcio09
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 mis-



Trobareu el fitxer  
`Excepcio09.java` en la secció  
"Recursos de contingut" del web.

satge 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 `Exception` que es llança per una excepció `RuntimeException`.



Trobareu el fitxer `Excepcio10.java` en la secció "Recursos de contingut" del web.

```
/* Fitxer: Excepcio10.java
   Descripció: Exemple de llançament d'excepció RuntimeException en un mètode
   Autor: Isidre Guixà
*/

public class Excepcio10
{
    static void verificaLengthTaula (int n, String t[])
    /* Mètode que avalua si la taula t té n cel·les, provocant, en cas de ser avaluada com a fals,
       una excepció RuntimeException */
    {
        if (t.length!=n) throw new RuntimeException ("La taula no té la llargada indicada.");
        System.out.println ("Sortida de verificaLengthTaula.");
    }

    public static void main (String args[])
    {
        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.");
        System.out.println ("Programa finalitzat.");
    }
}
```

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:

```
G:\>java -Dfile.encoding=cp850 Excepcio10
Punt 1.
Sortida de verificaLengthTaula.
Punt 2.
Exception in thread "main" java.lang.RuntimeException: La taula no té la llargada indicada.
    at Excepcio10.verificaLengthTaula(Excepcio10.java:12)
    at Excepcio10.main(Excepcio10.java:21)
```

En aquest cas, l'execució del `main()` no finalitza correctament, ja que es produeix l'excepció no gestionada.

### 3.3.4. 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.



Trobareu el fitxer `Excepcio11.java` en la secció "Recursos de contingut" del web.

#### Exemple de classe excepció creada pel programador

```
/* Fitxer: Excepcio11.java
   Descripció: Exemple d'excepció creada pel programador
   Autor: Isidre Guixà
*/

public class Excepcio11
{
    public static void main (String args[])
    {
        try
        {
            provocoExcepcio(0);
            provocoExcepcio(10);
        }
        catch (LaMevaExcepcio e)
        {
            e.printStackTrace();
        }
        System.out.println ("El programa finalitza correctament");
    }
    public static void provocoExcepcio(int valor) throws LaMevaExcepcio
    {
        System.out.println ("Valor: " + valor);
        if (valor!=0) throw new LaMevaExcepcio (valor);
        System.out.println ("No s'ha provocat l'excepció.");
    }
}

class LaMevaExcepcio extends Exception
{
    private Integer valor;

    public LaMevaExcepcio (int xxx)
    {
        valor = new Integer(xxx);
    }

    public String toString ()
    {
        return "Exception LaMevaExcepcio: Error motivat per valor = " + valor.toString();
    }
}
```

Observem l'execució del programa:

```
G:\>java -Dfile.encoding=cp850 Excepcio11
Valor: 0
No s'ha provocat l'excepció.
Valor: 10
Exception LaMevaExcepcio: Error motivat per valor = 10
    at Excepcio11.provocoExcepcio(Excepcio11.java:25)
    at Excepcio11.main(Excepcio11.java:13)
El programa finalitza correctament
```

### 3.3.5. Efecte de l'herència

Si un mètode d'una classe és una sobreescritura 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 apunten a objectes de classes derivades mantenint la gestió d'excepcions.

### 3.4. Obtenció de dades per l'entrada estàndard

La manera més simple de visualitzar alguna informació per la sortida estàndard (consola si no es redirecciona) es basa en la utilització dels mètodes `print()` i `println()` aplicats sobre l'objecte `System.out`. Tothom que hagi fet alguna petita incursió en el llenguatge Java coneix aquests mètodes, però..., i per obtenir informació per l'entrada estàndard (teclat)? Això ja és més complicat perquè necessitem conèixer la gestió d'excepcions.

Per poder utilitzar l'entrada estàndard necessitem utilitzar l'objecte `System.in` que pertany a la classe `InputStream` i, per tant, ens caldrà conèixer-ne els mètodes.

El mètode més bàsic és el mètode `InputStream.read()`, que permet llegir el byte següent de l'entrada sobre la qual s'estigui aplicant. Així, doncs, per efectuar la lectura del byte següent hem d'executar una cosa similar a aquesta:

---

```
char c = (char) System.in.read();
```

---

Si fem una ullada a la documentació d'aquest mètode veurem que llança l'excepció `IOException` i, per tant, la instrucció anterior ha d'estar dins un bloc `try-catch` o en un mètode que també llanci l'excepció.

Però, com que la lectura byte a byte és molt feixuga, ens convé disposar d'algun mètode per llegir una línia sencera i Java ens proporciona el mètode `BufferedReader.readLine()`. Aquest mètode, que llança l'excepció `IOException`, permet llegir una línia de text, que no és més que una seqüència de caràcters finalitzada amb els caràcters `'\n'` (*line feed*), `'\r'` (*carriage return*) o la seqüència `"\r\n"`.

El mètode `readLine()` s'executa sobre un objecte `BufferedReader` i ens manca saber com obtenir un objecte d'aquesta classe que estigui vinculat a l'objecte `System.in`. La recepta és:

---

```
BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
```

---

Amb aquesta definició, totes les lectures efectuades sobre l'objecte `br` en realitat es realitzen sobre `System.in`, amb l'avantatge que llegim una línia sencera.

#### Exemple de programa que captura text pel teclat

El programa següent permet efectuar l'entrada d'un text per teclat. L'entrada s'efectua línia a línia, que es van emmagatzemant en un vector. En finalitzar la introducció de text, es mostra el contingut de les línies del vector.



Trobareu el fitxer `Entrada.java` en la secció "Recursos de contingut" del web.

```
/* Fitxer: Entrada.java
   Descripció: Exemple d'entrada de text per l'entrada estàndard
   Autor: Isidre Guixà
*/
import java.io.*;
import java.util.Vector;

public class Entrada
{
    public static void main(String args[])
    {
        BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
        Vector<String> textos = new Vector<String>();
        String text;
        System.out.println("Iniciï la introducció de text.");
        System.out.println("Per finalitzar, introdueixi una línia buida (en blanc):");
        try
        {
            do
            {
                text = br.readLine();
                if (!(text.isEmpty())) textos.add(text);
            } while (!(text.isEmpty()));
            System.out.println("S'ha introduït " + textos.size() + " línies:");
        }
        catch (IOException e)
        {
            System.out.println("S'ha produït una excepció en la captura de dades:");
            System.err.println(e);
            System.out.println("S'ha introduït " + textos.size() + " línies abans de l'excepció");
        }
        finally
        {
            for (int i=0; i<textos.size(); i++) System.out.println (textos.get(i));
        }
    }
}
```

L'execució d'aquest programa sense provocar cap interrupció dona la sortida:

```
G:\>java -Dfile.encoding=cp850 Entrada
Iniciï la introducció de text.
Per finalitzar, introdueixi una línia buida (en blanc):
Un flux (stream) és el terme abstracte usat per referir-se al mecanisme
que permet transmetre un conjunt de dades seqüencials des d'un origen de
dades (data source) fins un destí (data sink).

S'ha introduït 3 línies:
Un flux (stream) és el terme abstracte usat per referir-se al mecanisme
que permet transmetre un conjunt de dades seqüencials des d'un origen de
dades (data source) fins un destí (data sink).
```

### 3.5. API de reflexió

L'API de Java proporciona moltes possibilitats i una d'elles és la d'accedir des d'un programa, en temps d'execució, a tota la seva informació i la in-

formació de l'API del llenguatge i de les biblioteques a què tingui accés. I no solament permet l'accés a la informació, sinó també la manipulació. Totes aquestes possibilitats es concentren en l'**API de reflexió** formada per les classes del paquet `java.lang.reflect` que s'utilitzen conjuntament amb la classe `java.lang.Class<T>`. La taula 13 presenta un resum d'algunes de les classes involucrades.

Taula 13. Resum d'algunes de les classes involucrades en la reflexió

Classe	Objectiu
<code>java.lang.Class</code>	Proporciona mètodes que obtenen informació sobre tots els membres d'una classe.
<code>java.lang.reflect.Field</code>	Proporciona mètodes que modifiquen o obtenen informació sobre els camps d'una classe.
<code>java.lang.reflect.Method</code>	Proporciona mètodes per accedir i cridar els mètodes <u>no constructors</u> d'una classe i obtenir-ne les definicions.
<code>java.lang.reflect.Array</code>	Proporciona mètodes <code>static</code> per accedir a taules Java i crear-ne dinàmicament.
<code>java.lang.reflect.Constructor&lt;T&gt;</code>	Proporciona accés de reflexió als constructors.

Per fer-nos una idea, aquesta API permet cridar dinàmicament mètodes (incloent-hi constructors) d'una altra classe, de manera que podem executar accions que van des de saltar-nos els modificadors d'accés d'una definició de classe fins a fer que dos objectes intercanviïn informació de dades privades sense utilitzar els mètodes `getter` i/o `setter`.

Per tant, aquesta API proporciona moltes possibilitats, però cal tenir en compte que és complexa i que molts errors que normalment es detecten en temps de compilació no es detectaran fins a l'execució. **!!**

Cal fer una ullada a la documentació de Java per conèixer els mètodes que proporciona cada classe.

#### Utilització de reflexió per obtenir una llista dels mètodes no constructors d'una classe

A continuació es presenta un programa senzill per obtenir informació sobre tots els mètodes d'una classe:

```

/* Fitxer: DumpMethods.java
   Descripció: Facilita el llistat de tots els mètodes d'una classe que cal indicar en executar
               el programa
   Autor: Glen McCluskey - Article "Using Java Reflection"
           Documentació de Java referent a "Reflection"
*/
import java.lang.reflect.*;

public class DumpMethods
{
    public static void main(String args[])
    {
        try
        {
            Class c = Class.forName(args[0]);
            Method m[] = c.getDeclaredMethods();
            for (int i = 0; i < m.length; i++)
                System.out.println(m[i].toString());
        }
        catch (Throwable e)
        {
            System.err.println(e);
        }
    }
}

```



Trobareu el fitxer `DumpMethods.java` en la secció "Recursos de contingut" del web.

El programa està pensat per cridar-lo acompanyat del nom d'una classe existent. En cas que no s'indiqui una classe o s'indiqui una classe inexistente, captura l'error i informa de l'excepció. Fixem-nos en el resultat quan es crida sobre la classe `Stack` del llenguatge Java o sobre la mateixa classe `DumpMethods`:

---

```
G:\>java DumpMethods java.util.Stack
public synchronized java.lang.Object java.util.Stack.pop()
public java.lang.Object java.util.Stack.push(java.lang.Object)
public boolean java.util.Stack.empty()
public synchronized java.lang.Object java.util.Stack.peek()
public synchronized int java.util.Stack.search(java.lang.Object)

G:\>java DumpMethods DumpMethods
public static void DumpMethods.main(java.lang.String[])
```

---

Aquest programa, amb el mètode `Class.forName()`, carrega la classe especificada en l'argument i utilitza el mètode `Method.getDeclaredMethods()` per recollir la llista de mètodes de la classe indicada. La màquina virtual ha de tenir accés, durant l'execució del programa, a la classe que l'usuari indiqui. En el cas que ens ocupa, la màquina virtual té accés a les classes proporcionades pel mateix llenguatge (`Stack`) i a la classe que s'està executant (`DumpMethods`). També és possible indicar qualsevol classe sempre que l'arxiu corresponent `.class` sigui accessible per la variable d'entorn `CLASSPATH` o l'opció `-cp` en el moment d'executar el programa.

La informació que s'obté és part de la que proporciona l'eina `javap -private`, ja que aquesta eina proporciona informació de tots els membres, mentre que el programa desenvolupat només proporciona informació dels mètodes no constructors:

---

```
G:\>javap java.util.Stack
Compiled from "Stack.java"
public class java.util.Stack extends java.util.Vector{
    public java.util.Stack();
    public java.lang.Object push(java.lang.Object);
    public synchronized java.lang.Object pop();
    public synchronized java.lang.Object peek();
    public boolean empty();
    public synchronized int search(java.lang.Object);
}

G:\>javap DumpMethods
Compiled from "DumpMethods.java"
public class DumpMethods extends java.lang.Object{
    public DumpMethods();
    public static void main(java.lang.String[]);
}
```

---

### Utilització de reflexió per obtenir el detall dels mètodes no constructors d'una classe

A continuació es presenta un programa senzill per obtenir informació detallada de tots els mètodes no constructors d'una classe (paràmetres, excepcions que llancen i valor de retorn).



Trobareu el fitxer `Method1.java` en la secció "Recursos de contingut" del web.

---

```
/* Fitxer: Method1.java
   Descripció: Facilita informació detallada per tots els mètodes no constructors d'una classe
               (paràmetres, excepcions que llancen i valor de retorn)
   Autor: Glen McCluskey - Article "Using Java Reflection"
   Documentació de Java referent a "Reflection"
*/
import java.lang.reflect.*;

public class Method1
{
    private int f1(Object p, int x) throws NullPointerException
```

```

    {
        if (p == null)
            throw new NullPointerException();
        return x;
    }

    public static void main(String args[])
    {
        try
        {
            Class cls = Class.forName("Method1");
            Method methlist[] = cls.getDeclaredMethods();
            for (int i = 0; i < methlist.length; i++)
            {
                Method m = methlist[i];
                System.out.println("name = " + m.getName());
                System.out.println("decl class = " + m.getDeclaringClass());
                Class pvec[] = m.getParameterTypes();
                for (int j = 0; j < pvec.length; j++)
                    System.out.println(" param #" + j + " " + pvec[j]);
                Class evec[] = m.getExceptionTypes();
                for (int j = 0; j < evec.length; j++)
                    System.out.println("exc #" + j + " " + evec[j]);
                System.out.println("return type = " + m.getReturnType());
                System.out.println("-----");
            }
        }
        catch (Throwable e)
        {
            System.err.println(e);
        }
    }
}

```

---

En aquest cas, el programa visualitza la informació detallada dels mètodes no constructors de la pròpia classe, però amb petites modificacions es pot aconseguir un programa que visualitzi la informació detallada de tots els mètodes no constructors d'una classe indicada en cridar el programa.

Per poder comprovar que apareix la informació detallada de les excepcions, la classe incorpora un mètode amb clàusula `throws`. L'execució del programa dóna la sortida següent:

---

```

G:\>java Method1
name = f1
decl class = class Method1
  param #0 class java.lang.Object
  param #1 int
exc #0 class java.lang.NullPointerException
return type = int

-----

name = main
decl class = class Method1
  param #0 class [Ljava.lang.String;
return type = void
-----

```

---

Aprofitem aquest exemple per comentar que quan l'API de reflexió ha d'informar sobre el tipus d'una dada que és una taula d'un tipus X, utilitza la forma `class [X`. És a dir, la paraula `class` seguida del claudàtor d'obertura `[` i el tipus de dades de la taula. No mostra el claudàtor de tancament `]`.

### Utilització de reflexió per obtenir el detall dels constructors d'una classe

A continuació es presenta un programa senzill per obtenir informació detallada de tots els constructors d'una classe (paràmetres i excepcions que llancen).

```
/* Fitxer: Constructor1.java
   Descripció: Facilita informació detallada per tots els constructors d'una classe
               (paràmetres i excepcions que llancen)
   Autor: Glen McCluskey - Article "Using Java Reflection"
           Documentació de Java referent a "Reflection"
*/
import java.lang.reflect.*;

public class Constructor1
{
    public Constructor1() {}

    protected Constructor1(int i, double d) {}

    public static void main(String args[])
    {
        try
        {
            Class cls = Class.forName("Constructor1");
            Constructor ctorlist[] = cls.getDeclaredConstructors();
            for (int i = 0; i < ctorlist.length; i++)
            {
                Constructor ct = ctorlist[i];
                System.out.println("name = " + ct.getName());
                System.out.println("decl class = " + ct.getDeclaringClass());
                Class pvec[] = ct.getParameterTypes();
                for (int j = 0; j < pvec.length; j++)
                    System.out.println("param #" + j + " " + pvec[j]);
                Class evec[] = ct.getExceptionTypes();
                for (int j = 0; j < evec.length; j++)
                    System.out.println("exc #" + j + " " + evec[j]);
                System.out.println("-----");
            }
        }
        catch (Throwable e)
        {
            System.err.println(e);
        }
    }
}
```



Trobareu el fitxer  
Constructor1.java en la  
secció "Recursos de contingut" del  
web.

En aquest cas, el programa visualitza la informació detallada dels constructors de la pròpia classe, però amb petites modificacions es pot aconseguir un programa que visualitzi la informació detallada de tots els constructors d'una classe indicada en cridar el programa.

Per poder comprovar que apareix informació de constructors, la classe incorpora un parell de constructors. L'execució del programa dóna la sortida següent:

```
G:\>java Constructor1
name = Constructor1
decl class = class Constructor1
-----
name = Constructor1
decl class = class Constructor1
param #0 int
param #1 double
-----
```



Trobareu el fitxer Field1.java  
en la secció "Recursos de  
contingut" del web

### Utilització de reflexió per obtenir informació detallada de les dades d'una classe

A continuació es presenta un programa senzill per obtenir informació detallada de totes les dades classe.

---

```

/* Fitxer: Field1.java
   Descripció: Facilita informació detallada per tots els camps d'una classe
   Autor: Glen McCluskey - Article "Using Java Reflection"
   Documentació de Java referent a "Reflection"
*/

import java.lang.reflect.*;

public class Field1
{
    private double d;
    public static final int i = 37;
    String s = "testing";

    public static void main(String args[])
    {
        try
        {
            Class cls = Class.forName("Field1");
            Field fieldlist[] = cls.getDeclaredFields();
            for (int i = 0; i < fieldlist.length; i++)
            {
                Field fld = fieldlist[i];
                System.out.println("name = " + fld.getName());
                System.out.println("decl class = " + fld.getDeclaringClass());
                System.out.println("type = " + fld.getType());
                int mod = fld.getModifiers();
                System.out.println("modifiers = " + Modifier.toString(mod));
                System.out.println("-----");
            }
        }
        catch (Throwable e)
        {
            System.err.println(e);
        }
    }
}

```

---

En aquest cas, el programa visualitza la informació detallada de les dades de la pròpia classe, però amb petites modificacions es pot aconseguir un programa que visualitzi la informació detallada de totes les dades d'una classe indicada en cridar el programa. L'execució del programa dóna la sortida següent:

---

```

G:\>java Field1
name = d
decl class = class Field1
type = double
modifiers = private
-----
name = i
decl class = class Field1
type = int
modifiers = public static final
-----
name = s
decl class = class Field1
type = class java.lang.String
modifiers =
-----

```

---

És possible utilitzar la reflexió per cridar mètodes (constructors i no constructors), canviar els valors de les dades sense utilitzar els mètodes `setter` i crear i manipular taules de Java.

### Utilització de reflexió per cridar un mètode no constructor

El programa següent ens mostra com es pot cridar, en temps d'execució, un mètode no constructor d'una classe utilitzant la reflexió.

```
/* Fitxer: Method2.java
   Descripció: Exemple de la reflexió per invocar un mètode no constructor
   Autor: Glen McCluskey - Article "Using Java Reflection"
   Documentació de Java referent a "Reflection"
*/

import java.lang.reflect.*;

public class Method2
{
    public int add(int a, int b)
    {
        return a + b;
    }

    public static void main(String args[])
    {
        try
        {
            Class cls = Class.forName("Method2");
            Class partypes[] = new Class[2];
            partypes[0] = Integer.TYPE;
            partypes[1] = Integer.TYPE;
            Method meth = cls.getMethod("add", partypes);
            Method2 methobj = new Method2();
            Object arglist[] = new Object[2];
            arglist[0] = new Integer(37);
            arglist[1] = new Integer(47);
            Object retobj = meth.invoke(methobj, arglist);
            Integer retval = (Integer)retobj;
            System.out.println(retval.intValue());
        }
        catch (Throwable e)
        {
            System.err.println(e);
        }
    }
}
```



Trobareu el fitxer Method2.java en la secció "Recursos de contingut" del web.

En aquest exemple se suposa que coneixem el detall del mètode a cridar, però és clar que si no el coneixem podem obtenir-lo amb l'ajut de l'API de reflexió.

El mètode `Class.getMethod()` ens permet obtenir una referència `meth` al mètode que volem cridar indicant els tipus de tots els arguments per definir quina sobrecàrrega del mètode volem cridar. Com que el mètode que volem cridar no és `static`, ens cal tenir un objecte de la classe sobre el qual cridar el mètode. Una vegada tenim l'objecte (apuntat per `methobj`) i el mètode (apuntat per `meth`) ens cal preparar la llista d'arguments (taula `arglist`) per, posteriorment, cridar el mètode `Method.invoke()` i informar de l'objecte i els arguments: `meth.invoke(methobj, arglist)`. Aquest mètode retorna una referència a un objecte, la qual serà `null` si el mètode cridat no retornava res.

L'exemple que ens ocupa crida el mètode `add` per sumar els valors 37 i 47:

```
G:\>java Method2
84
```



Trobareu el fitxer Constructor2.java en la secció "Recursos de contingut" del web.

### Utilització de reflexió per cridar un mètode constructor

El programa següent ens mostra com es pot cridar, en temps d'execució, un mètode constructor d'una classe utilitzant la reflexió.



---

```
/* Fitxer: Constructor2.java
   Descripció: Exemple de la reflexió per invocar un mètode constructor
   Autor: Glen McCluskey - Article "Using Java Reflection"
   Documentació de Java referent a "Reflection"
*/
import java.lang.reflect.*;

public class Constructor2
{
    public Constructor2() {}

    public Constructor2(int a, int b)
    {
        System.out.println("a = " + a + " b = " + b);
    }

    public static void main(String args[])
    {
        try
        {
            Class cls = Class.forName("Constructor2");
            Class partypes[] = new Class[2];
            partypes[0] = Integer.TYPE;
            partypes[1] = Integer.TYPE;
            Constructor ct = cls.getConstructor(partypes);
            Object arglist[] = new Object[2];
            arglist[0] = new Integer(37);
            arglist[1] = new Integer(47);
            Object retobj = ct.newInstance(arglist);
        }
        catch (Throwable e)
        {
            System.err.println(e);
        }
    }
}
```

---

En aquest exemple se suposa que coneixem el detall del constructor a cridar, però és clar que si no el coneixem podem obtenir-lo amb l'ajut de l'API de reflexió.

El mètode `Class.getConstructor()` ens permet obtenir una referència `ct` al constructor que volem cridar indicant els tipus de tots els arguments per definir quina sobrecàrrega del constructor volem cridar. Una vegada tenim el constructor (apuntat per `ct`) ens cal preparar la llista d'arguments (taula `arglist`) per, posteriorment, cridar el mètode `Constructor.newInstance()` sobre el constructor: `ct.newInstance(arglist)` que retornarà la referència a l'objecte creat.

L'exemple que ens ocupa crida el constructor `Constructor2(int, int)` amb els paràmetres 37 i 47. Com que aquest constructor imprimeix un resultat podem comprovar la creació de l'objecte corresponent en l'execució de l'exemple:

---

```
G:\>java Constructor2
a = 37 b = 47
```

---

### Utilització de reflexió per canviar les dades d'un objecte

El programa següent ens mostra com es pot canviar, en temps d'execució, les dades d'un objecte utilitzant la reflexió.



Trobareu el fitxer `Field2.java` en la secció "Recursos de contingut" del web.

---

```

/* Fitxer: Field2.java
   Descripció: Exemple de la reflexió per modificar les dades d'objectes
   Autor: Glen McCluskey - Article "Using Java Reflection"
   Documentació de Java referent a "Reflection"
*/

import java.lang.reflect.*;

public class Field2
{
    public double d;

    public static void main(String args[])
    {
        try
        {
            Class cls = Class.forName("Field2");
            Field fld = cls.getField("d");
            Field2 f2obj = new Field2();
            System.out.println("d = " + f2obj.d);
            fld.setDouble(f2obj, 12.34);
            System.out.println("d = " + f2obj.d);
        }
        catch (Throwable e)
        {
            System.err.println(e);
        }
    }
}

```

---

En aquest exemple se suposa que sabem que la classe `Field2` disposa d'una dada membre `d` que és `double`, però és clar que si no la coneixem podem obtenir aquesta informació amb l'ajut de l'API de reflexió.

El mètode `Class.getField()` ens permet obtenir una referència `fld` a la definició de la dada a la qual volem accedir indicant el nom de la dada. Fixem-nos que tot això s'ha dut a terme sobre la classe `Field2` i no sobre cap instància de la classe. Per tal de comprovar que podem modificar el contingut de la dada per a un objecte, procedim a crear un objecte, apuntat per `f2obj`. L'objectiu de l'exemple és comprovar que per modificar `f2obj.d` podem cridar el mètode `Field.setDouble()` sobre la referència `fld` que apunta a la definició de la dada membre tot indicant per paràmetre la instància (`f2obj`) sobre la qual cal aplicar la modificació i el nou valor: `fld.setDouble(f2obj, 12.34)`.

En l'exemple que ens ocupa, com que visualitzem el valor de `f2obj.d` abans i després de la modificació, podem comprovar que aquesta es duu a terme:

---

```

G:\>java Field2
d = 0.0
d = 12.34

```

---

Però, l'exemple proporcionat modifica una dada d'accés `public` i, en un exemple anterior (`Field1.java`) que contenia dades `private`, el mètode `Class.getDeclaredFields()` mostrava totes les dades: `public`, `private` i `protected`. Podem utilitzar la reflexió per modificar dades privades de classes que ni tant sols forin part del mateix paquet?

Per comprovar-ho podem fer una nova versió del fitxer `Field2.java` per modificar la dada `private` publicid `d` declarada a la classe `Field1`. Per poder comprovar el valor de `d` previ i posterior a la modificació, fem evolucionar `Field1.java` vers una nova versió `Field1Bis.java` que proporioni el mètode `toString()` per mostrar el contingut de les dades.



Trobareu els fitxers `Field2Bis.java` i `Field1Bis.java` en la secció "Recursos de contingut" del web.

---

```

/* Fitxer: Field2Bis.java
   Descripció: Exemple de la reflexió per modificar les dades d'objectes de la classe Field1Bis
   Autor: Isidre Guixà
*/
import java.lang.reflect.*;

public class Field2Bis
{

```

```

public static void main(String args[])
{
    try
    {
        Class cls = Class.forName("Field1Bis");
        Field fld = cls.getField("d");
        Field1Bis flobj = new Field1Bis();
        System.out.println(flobj);
        fld.setDouble(flobj, 12.34);
        System.out.println(flobj);
    }
    catch (Throwable e)
    {
        System.err.println(e);
    }
}

```

El programa es compila sense cap problema, però l'execució...

---

```
G:\>javac Field2Bis.java
```

```
G:\>java Field2Bis
java.lang.NoSuchFieldException: d
```

---

Aquesta situació ens serveix per exemplificar el que comentàvem més amunt referent al fet que molts errors es troben en fase d'execució perquè en fase de compilació no es poden comprovar.

El problema ve del fet que la dada `d` en la classe `Field1Bis` és privada i, en temps d'execució, el mètode `Class.getField(String)` no la detecta. Si fem una ullada a la documentació d'aquest mètode veurem que indica clarament que només detecta les dades `public`, heretades o no heretades.

En el mètode `main()` de `Field1.java` es podia accedir a totes les dades, ja que utilitzàvem el mètode `Class.getDeclaredFields()` que mostra totes les dades (`public`, `private` i `protected`) declarades a la classe (no mostra les heretades). Similar a aquest mètode, també disposem del mètode `Class.getDeclaredField(String)` que permet accedir directament a una dada declarada a la classe (`public`, `protected` o `private`). Així, doncs, passem a la versió `Field2Ter.java` per utilitzar `getDeclaredField(String)`.

---

```

/* Fitxer: Field2Ter.java
   Descripció: Exemple de la reflexió per modificar les dades d'objectes de la classe Field1Bis
   Autor: Isidre Guixà
*/

import java.lang.reflect.*;
public class Field2Ter
{
    public static void main(String args[])
    {
        try
        {
            Class cls = Class.forName("Field1Bis");
            Field fld = cls.getDeclaredField("d");
            Field1Bis flobj = new Field1Bis();
            System.out.println(flobj);
            fld.setDouble(flobj, 12.34);
            System.out.println(flobj);
        }
        catch (Throwable e)
        {
            e.printStackTrace();
        }
    }
}

```



**Trobareu el fitxer**  
**Field2Ter.java** en la secció  
**"Recursos de contingut"** del web.

El programa es compila sense cap problema, però l'execució encara dona problemes...

```
G:\>javac Field2Ter.java
```

```
G:\>java Field2Ter
d = 0.0 i = 37 s = testing
java.lang.IllegalAccessException: Class Field2Ter can not access a member of class Field1Bis with
modifiers "private"
    at sun.reflect.Reflection.ensureMemberAccess(Reflection.java:65)
    at java.lang.reflect.Field.doSecurityCheck(Field.java:960)
    at java.lang.reflect.Field.getFieldAccessor(Field.java:896)
    at java.lang.reflect.Field.setDouble(Field.java:889)
    at Field2Ter.main(Field2Ter.java:18)
```

Fixem-nos que l'error es produeix en la línia 18 del `main()`, en cridar el mètode `setDouble()`, que és quan intentem modificar la dada membre `private`. És a dir, continuem topant amb el mur de l'accés, cosa que no ens ha d'estranyar atès que la seguretat d'accés a les dades és un dels pilars fonamentals en Java.

Però..., saltem-nos el caràcter de privacitat del membre `Field1Bis.d` gràcies a la utilització del mètode `AccessibleObject.setAccessible()` aplicant-lo sobre l'objecte `Field` que apunta al membre `Field1Bis.d`. Això és possible perquè la classe `Field` (a l'igual de les classes `Constructor` i `Method`) deriva de la classe `AccessibleObject`. Així, doncs, arribem a la versió definitiva de la classe `Field2`:

```
/* Fitxer: Field2Quater.java
   Descripció: Exemple de la reflexió per modificar les dades d'objectes de la classe Field1Bis
   Autor: Isidre Guixà
*/
import java.lang.reflect.*;
public class Field2Quater
{
    public static void main(String args[])
    {
        try
        {
            Class cls = Class.forName("Field1Bis");
            Field fld = cls.getDeclaredField("d");
            fld.setAccessible(true);
            Field1Bis flobj = new Field1Bis();
            System.out.println(flobj);
            fld.setDouble(flobj, 12.34);
            System.out.println(flobj);
        }
        catch (Throwable e)
        {
            System.err.println(e);
        }
    }
}
```

Ara sí que el programa es compila i executa sense cap problema...

```
G:\>javac Field2Quater.java
```

```
G:\>java Field2Quater
d = 0.0 i = 37 s = testing
d = 12.34 i = 37 s = testing
```

En aquest darrer exemple ha quedat clar que ens podem saltar el sistema de seguretat, però això dependrà de l'administrador de seguretat (`SecurityManager`) que tinguem activat. Així, per exemple, el que hem fet en el darrer exemple seria impossible des d'un Applet, però la gestió de la seguretat en Java és un capítol a part.



**Trobareu el fitxer**  
`Field2Quater.java` en la  
secció "Recursos de contingut" del  
web.

#### Applet

Petita aplicació Java resident en els servidors web que baixa al navegador d'un client per ser executada.

### Utilització de reflexió per crear i manipular taules unidimensionals

L'exemple següent crea una taula `String[10]` i omple la posició 5 amb una cadena. Posteriorment, el valor es recupera i es visualitza.

```
/* Fitxer: Array1.java
   Descripció: Exemple de creació i manipulació d'una taula via reflexió.
   Autor: Glen McCluskey - Article "Using Java Reflection"
   Documentació de Java referent a "Reflection"
*/
import java.lang.reflect.*;

public class Array1
{
    public static void main(String args[])
    {
        try
        {
            Class cls = Class.forName("java.lang.String");
            Object arr = Array.newInstance(cls, 10);
            Array.set(arr, 5, "this is a test");
            String s = (String)Array.get(arr, 5);
            System.out.println(s);
        }
        catch (Throwable e)
        {
            System.err.println(e);
        }
    }
}
```



Trobareu el fitxer `Array1.java` en la secció "Recursos de contingut" del web.

El mètode `Array.newInstance()` crea una taula d'objectes de la classe decidida en temps d'execució. Els mètodes `Array.set()` i `Array.get()` permeten accedir a les cel·les de la taula i actuar segons convingui tenint en compte que és possible que calgui utilitzar conversions `cast`.

### Utilització de reflexió per crear i manipular taules multidimensionals

L'exemple següent crea una taula tridimensional per reflexió i mostra exemples de com es pot gestionar. El mateix codi incorpora les explicacions corresponents a cada instrucció.

```
/* Fitxer: Array2.java
   Descripció: Exemple de gestió de taules multidimensionals via reflexió
   Autor: Glen McCluskey - Article "Using Java Reflection"
   Documentació de Java referent a "Reflection"
*/
import java.lang.reflect.*;

public class Array2
{
    public static void main(String args[])
    {
        int dims[] = new int[]{5, 10, 15};
        Object arr = Array.newInstance(Integer.TYPE, dims);
        // arr és una taula d'Integer de 3 dimensions: 5, 10, 15
        // Per donar visibilitat:
        // arr és una referència a una taula de 5 cel·les
        // on cada cel·la és una referència a una taula de 10 cel·les
        // cadascuna de les quals, és una referència a una taula de 15 cel·les
        Object arrobj = Array.get(arr, 3);
        // arrobj apunta a la taula de 10 cel·les que era apuntada per arr[3]
        Class cls = arrobj.getClass().getComponentType();
        System.out.println(cls);
        // Aquest missatge ens ha de retornar que es tracta d'una taula.
        // Utilitzarà la nomenclatura class [I
        // Tot i que és una taula de 2 dimensions, no ens ho informa
        arrobj = Array.get(arrobj, 7);
        // arrobj apunta a la taula de 15 cel·les que era apuntada per arr[3][7]
        Array.setInt(arrobj, 12, 37);
        // Hem ubicat l'Integer 37 a la posició 12 de la taula apuntada per arr[3][7],
```



Trobareu el fitxer `Array2.java` en la secció "Recursos de contingut" del web.

```
// és a dir, hem ubicat l'Integer 37 a la posició arr[3][7][12]
int arrcast[][][] = (int[][][])arr;
// Hem creat una referència "arrcast" a una taula tridimensional i li hem assignat
// la taula tridimensional apuntada per "arr"
System.out.println(arrcast[3][7][12]);
// Accedim a la posició de la taula "arr" on havíem intentat ubicar l'Integer 37
// utilitzant la referència "arrcast" que ens permet utilitzar els claudàtors per
// accedir a qualsevol posició, fet que no podem utilitzar amb la referència "arr"
// donat que no està definida com a taula. Haguéssim, però, pogut fer cast:
System.out.println(((int[][][])arr)[3][7][12]);
    }
}
```

---

L'execució del programa dóna la sortida següent:

---

```
G:\>java Array2
class [I
37
37
```

---