Biblioteques. Proves. Recursivitat

Joan Arnedo Moreno

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

Índex

Introducció

Ara mateix, es pot considerar que el desenvolupament de programes ben estructurats es fonamenta en un seguit de recursos bàsics:

- L'organització dels valors a tractar mitjançant tipus de dades apropiades.
- L'ús de les estructures seqüencial, condicional i repetitiva.
- La descomposició de problemes mitjançant disseny descendent.

Un dels recursos més importants és el darrer, la descomposició del problema, ja que si es realitza correctament permet reutilitzar un bloc de codi, en forma de mètode, en diferents parts del programa sense haver d'escriure'l repetides vegades. En aquest sentit, una eina força interessant seria poder aplicar aquesta reutilització, no només dins un mateix programa, sinó en qualsevol programa. Un cop s'ha resolt un subproblema per a un cas concret, poder usar els mètodes resultants a altres programes diferents sense haver de fer "copiar i enganxar" del codi. A mesura que aneu resolent programes, us resulta més fàcil fer-ne el codi de nous.

Un altre aspecte a tenir en compte vinculat a la descomposició del problema és el fet que, en els programes complexos, el nombre de mètodes resultants pot ser força gran, cosa que pot fer que el fitxer del vostre programa creixi força fins al punt de ser difícil de llegir o seguir. Potser també seria interessant disposar de mecanismes per ordenar els mètodes segons algun criteri (temàtica, tipus de paràmetres, etc.), de la mateixa manera que és més polit poder ordenar un seguit de fotos en carpetes etiquetades, en lloc de tenir-les totes en una de sola.

En l'apartat "Programació modular" es presenta el mecanisme que proporcionen els llenguatges estructurats per poder disposar d'aquesta possibilitat, mitjançant la realització d'un disseny modular de les aplicacions. Així, doncs, aprendreu els conceptes de **mòdul** aplicables a la majoria de llenguatges, i els dureu a la pràctica en el llenguatge Java. Les grans aplicacions es trossegen en mòduls, cadascun dels quals té una funcionalitat determinada. Així, en una aplicació per a una gestió integral d'un centre docent, podríem tenir el mòdul que gestioni el personal que treballa en el centre, el mòdul que gestiona l'alumnat, el mòdul que gestiona els horaris, etc. La llista podria anar-se ampliant. Aquests mòduls tenen una certa autonomia, però necessiten mantenir relacions amb la resta de mòduls que formen l'aplicació.

Un cop disposeu ja de totes les eines per generar programes complexos, en l'apartat "Biblioteques. Proves. Recursivitat" es mostren tot un seguit de tècniques complementàries per poder garantir que aquests són robustos i ben documentats. També s'introduirà breument el concepte de **recursivitat**, un mecanisme avançat per poder simplificar la codificació de certs tipus de problemes complexos.

Per aconseguir els nostres objectius, és especialment important reproduir en el vostre entorn de treball tots els exemples incorporats en el text, les activitats i els exercicis d'autoavaluació.

Resultats d'aprenentatge

En finalitzar aquesta unitat, l'alumne/a:

1. Prova, depura, comenta i documenta els programes.
2. Defineix el concepte de llibreries i la seva utilitat.
3. Utilitza llibreries en l'elaboració de programes
4. Coneix les nocions bàsiques de la recursivitat i llurs aplicacions clàssiques.

1. Descomposició en classes i biblioteques

A l'hora de crear una unitat didàctica o un llibre, estructurar-la en apartats o capítols permet que sigui molt més fàcil plantejar els seus continguts, des del punt de vista de l'autor, i seguir o comprendre, des del punt de vista del lector (o el mateix autor, si en el futur vol fer esmenes). Ara bé, per molt ordenada que sigui aquesta estructura, a mesura que creix la mida del document, tard o d'hora arribarà el moment que caldrà plantejar-se si més aviat el que resulta necessari és començar una nova unitat o un nou volum. Un document massa llarg es fa feixuc i difícil de gestionar. Quan això succeeix, normalment la divisió es plantejarà de manera que cada unitat o volum correspongui a una temàtica i resultats d'aprenentatge concrets.

En aquest sentit, i com ja heu vist, un programa no és gaire diferent. La seva subdivisió en diferents mètodes a partir de l'aplicació del disseny descendent permet tenir codi endreçat i polit, molt més fàcil tant de crear com d'entendre. Addicionalment, també permet reusar codi útil en diferents parts del programa, en lloc d'haver de reescriure'l diverses vegades. Ara bé, també de la mateixa manera, si el programa és complex i acaba requerint de molts mètodes, la quantitat de mètodes de tota mena i la mida del seu fitxer també pot arribar a ser considerable.

Una solució que permeten molts llenguatges de programació és seguir exactament el camí de l'analogia plantejada: dividir el codi del programa en diferents parts, d'acord a una temàtica concreta.

Un programa monolític és com un bloc de pedra, fet d'una sola peça.
Font: Richard-sr

> Un **programa modular** és aquell que s'ha desenvolupat dividint-lo en components fàcils de modificar i intercanviar, anomenats **mòduls**. Cada mòdul agrupa un conjunt de funcions o mètodes que realitzen tasques relacionades.

En contraposició a un programa modular, hi ha els **programes monolítics**, en els quals tot el codi es concentra en un únic mòdul. Normalment, en el cas de programes complexos, se sol considerar un mal hàbit fer programes monolítics.

1.1 Programes amb múltiples classes

La manera més directa de fer un programa modular és establir una correspondència un a un entre mòduls i fitxers on està escrit el codi font d'un programa. Per tant, cada mòdul es representa amb un fitxer de codi font diferent. Aquesta aproximació es pot usar en molts llenguatges de programació. En el cas de Java, un programa modular està compost de l'agregació de múltiples classes, en lloc de només una com heu treballat fins ara.

1.1.1 Què és realment una classe?

Abans de veure com és possible fer un programa modular, val la pena recapitular sobre el concepte de "classe" i la seva aplicació dins dels programes que es desenvolupen en Java. Si bé una definició estricta i formal d'aquest concepte està molt vinculada a l'orientació a objectes, i per tant més enllà de l'àmbit d'aquest temari, si feu recull de tots els cops que s'ha usat des d'una perspectiva pràctica, us adonareu que ha estat per referir-se a tres coses.

- **Un programa en Java**. D'una banda, els fitxers dels vostres programes, pròpiament, són classes (al cp i a la fi, s'inicien amb la declaració `public class...`). En aplicar disseny descendent, el seu codi queda distribuït en un mètode principal (`main`), que indica la seva primera instrucció i el punt d'inici del seu flux de control, junt amb diferents mètodes addicionals que poden ser invocats directament.

- **Un repositori de mètodes**. D'altra banda, també s'ha usat el terme classe per referir-se a biblioteques de mètodes, que actuen com extensions en les instruccions disponibles per defecte en el llenguatge. Abans de poder fer-ho, però, cal inicialitzar-les correctament. L'exemple més clar és la classe `Scanner`, que ofereix un repertori de mètodes per controlar la lectura de dades des del teclat (`nextLine()`, `nextInt()`, `hasNextFloat()`, etc.).

- **Un tipus compost**. Finalment, aquest mateix terme s'ha usat com a sinònim de tipus compost. En aquest cas, l'exemple per antonomàsia és la classe `String`, utilitzada per referir-se a cadenes de text dins de Java. Els tipus compostos de Java permeten manipular dades complexes mitjançant la invocació de mètodes (`charAt(...)`, `indexof(...)`, etc.).

El codi font de `Scanner` i `String` està disponible a la pàgina de descàrregues d'Oracle (*Java SE 6 JDK Source Code*). Ara bé, no és pas senzill d'entendre!

Si us hi fixeu, en tots tres casos hi ha un factor comú. Tots disposen d'un conjunt de mètodes que és possible invocar. I és que, en realitat, tot i les seves diferències en el context sota el qual s'usen i la manera d'invocar els seus mètodes, els tres casos són exactament el mateix en darrera instància: codi font dins un fitxer anomenat `NomClasse.java`, amb la declaració `public class NomClasse...`, i estructurat com un seguit de mètodes declarats dins seu.

Per tant, tot i que no sabeu exactament quin és el seu codi font al complet, està clar que la classe `Scanner` ha estat desenvolupada per algú altre, dins d'un fitxer anomenat `Scanner.java`, i una part del seu codi és:

```
1  public class Scanner {
2    //Altres declaracions (constants, vars. globals...) ...
3    public String nextLine() {
4      //Codi del mètode ...
5    }
6    public int nextInt() {
7      //Codi del mètode ...
8    }
9    //Altres mètodes ...
10 }
```

El mateix passa amb la classe `String` (amb les definicions dels seus propis mètodes):

```
public class String {
  //Altres declaracions (constants, vars. globals...) ...
  public int indexOf(String textACercar)() {
    //Codi del mètode ...
  }
  public int length() {
    //Codi del mètode ...
  }
  //Altres mètodes ...
}
```

1.1.2 Estructura d'un programa modular en Java

Un cop s'ha vist que totes les classes, independentment del seu rol, en realitat es codifiquen de la mateixa manera (com fitxers que contenen un seguit de mètodes que poden ser invocats), és el moment de veure com es relacionen a escala general per generar un programa modular.

Primer de tot, es recapitularà també sobre què és un programa, preveient tots els nous elements que heu anat aprenent. Recordeu que un programa no és més que una seqüència ordenada d'instruccions que es van executant d'inici a fi. Aquesta seqüència té certes particularitats, ja que mitjançant les estructures de control (selecció i repetició), és possible crear bifurcacions o bucles a la seqüència. A més a més, per a programes complexos, també es poden trobar invocacions a mètodes, fent possible distribuir les instruccions en blocs diferents, que poden ser executats repetides vegades en diferents moments del procés. Tots aquests mètodes estan escrits dins un mateix fitxer, que conté el codi font de tot el programa. D'entre tots els mètodes, hi ha el principal (`main`), el qual indica quina és la primera instrucció.

Doncs bé, l'única diferència d'un programa modular en Java és que els mètodes, en lloc d'estar escrits tots en el mateix fitxer, estan distribuïts dins de diferents fitxers (diferents classes), tal com es contrasta a la figura 1.1. Fixeu-vos que només canvia on estan escrits físicament els mètodes, però el flux de control, amb l'ordre d'execució de les instruccions i invocació dels mètodes, és exactament el mateix. En aquest exemple, el programa modular es composaria de tres fitxers diferents, anomenats `Principal.java`, `Modul1.java` i `Modul2.java`.

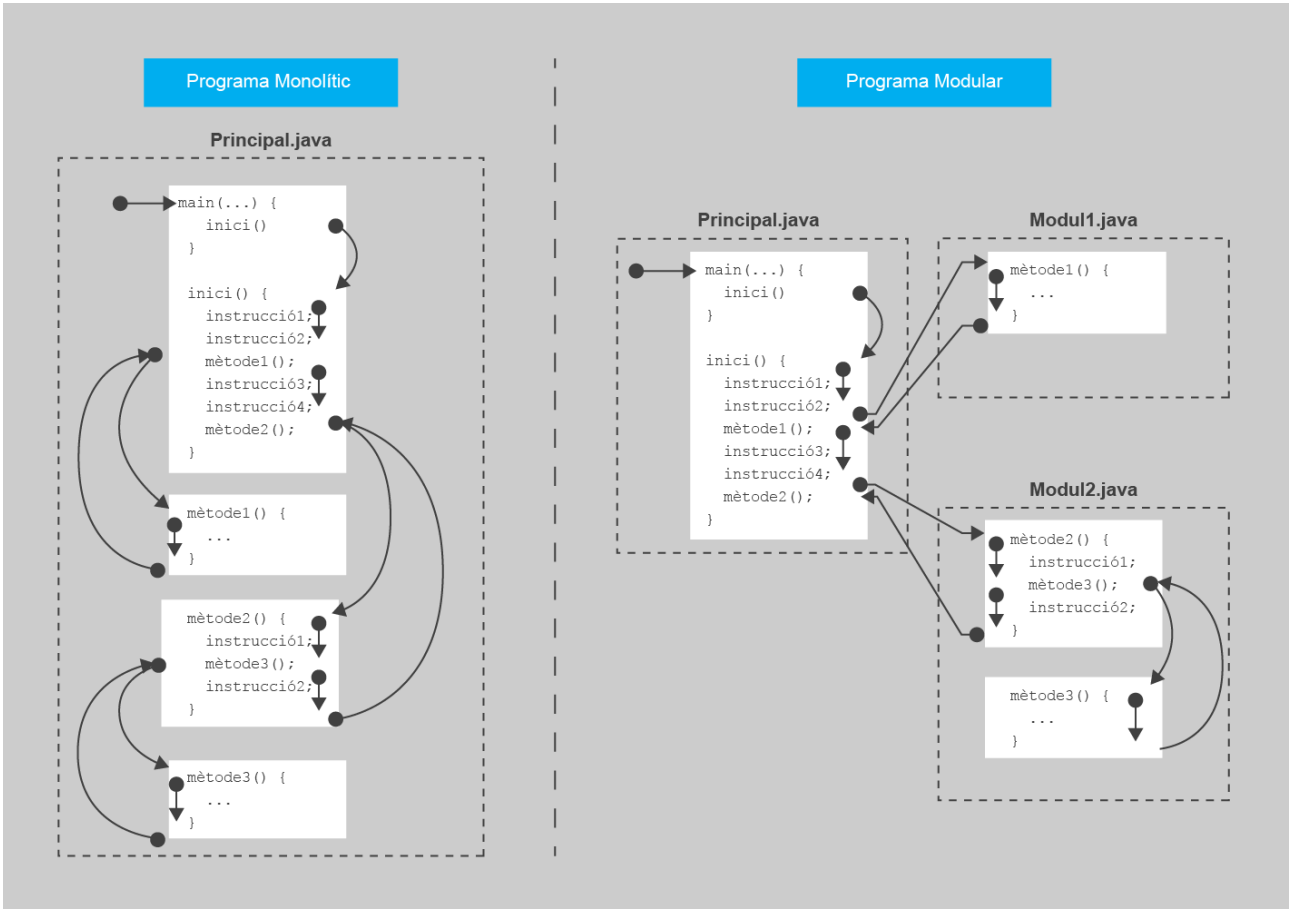
Normalment, l'escriptura de mètodes en fitxers separats no és arbitrària, sinó que se sol fer d'acord a algun criteri d'ordre, com agrupar mètodes per funcionalitats semblants (una classe amb els mètodes relacionats amb operar amb *arrays*, una altra amb els que llegeixen dades des del teclat, etc.). Per tant, el quid de la qüestió per fer un programa modular és ser capaç de decidir amb criteri com distribuir els diferents mètodes en diferents classes (i en quantes) i saber com invocar mètodes declarats a altres classes.

D'entre totes les classes que composen un programa modular en Java, hi ha d'haver una d'especial, a la qual se sol referir com a **classe principal** (*Main class*).

Els IDE solen distingir gràficament el fitxer de la classe principal amb alguna icona especial.

Aquesta és l'única, d'entre totes, que disposa d'un mètode principal declarat en el seu codi (la resta no el tenen). A la figura, **Principal.java** seria la classe principal. Per executar un programa modular en Java, només cal executar aquesta classe principal. A partir d'aquí, l'execució de les diferents instruccions segueix el flux de control habitual, partint del mètode principal d'aquesta classe, tal com s'ha mostrat a la figura.

FIGURA 1.1. Flux de control d'un programa monolític i un de modular en Java



Els mètodes main i inici

Donat el tipus de plantilla que s'usa en aquests materials per poder fer classes amb mètodes, a la classe principal hi ha d'haver tant el mètode `main` com el mètode **inici**. Cap dels dos és necessari a la resta de classes.

Les classes addicionals dins un programa modular, que no són la principal, normalment tenen dos orígens. Poden ser classes creades per vosaltres mateixos, o bé creades per altres desenvolupadors. En qualsevol dels dos casos, totes estan declarades i codificades en fitxers `.java` per separat.

1.1.3 Definició i ús de classes addicionals

Com acabeu de veure, és possible dividir el codi dels vostres programes en diferents classes, en lloc de tenir-lo en una de sola. Atès que és el cas més senzill, aquesta secció se centrarà en la definició de classes addicionals que es comporten simplement com repositoris de mètodes, i no és el cas dels tipus compostos.

Primer de tot, cal establir un criteri general sota el qual decidir quan val la pena dividir els mètodes d'un programa entre diferents classes. Normalment, un primer criteri molt bàsic que us ha de fer pensar a aplicar modularitat és trobar-se davant d'un programa amb un fitxer de codi font llarg, on hi ha molts mètodes. Ara bé, el més important de tots és identificar mètodes que realitzen tasques del que es podria considerar una mateixa temàtica i que creieu que el seu codi us pot ser d'utilitat en el futur per a altres programes que genereu.

Com a fil argumental per veure tot plegat, vegeu un exemple concret no gaire complex. Suposeu un programa per dur a terme algunes operacions típiques (màxim, mínim i mitjana) sobre el conjunt de notes d'una classe. Per centrar l'exemple en l'aspecte important ara, generar un programa dividit en diferents classes, s'obviaran detalls com la interfície (entrada des del teclat, menús, etc.).

El programa tal com es podria plantejar directament partint d'una descomposició per disseny descendent, podria ser el que es mostra tot seguit:

```
1  public class RegistreNotes {
2    public static void main(String[] args) {
3      RegistreNotes programa = new RegistreNotes();
4      programa.inici();
5    }
6    public void inici() {
7      double[] notes = {2.0, 5.5, 7.25, 3.0, 9.5, 8.25, 7.0, 7.5};
8      double max = calcularMaxim(notes);
9      double min = calcularMinim(notes);
10     double mitjana = calcularMitjana(notes);
11     System.out.println("La nota màxima és " + max + ".");
12     System.out.println("La nota mínima és " + min + ".");
13     System.out.println("La mitjana de les notes és " + mitjana + ".");
14   }
15   public double calcularMaxim(double[] array) {
16     double max = array[0];
17     for (int i = 1; i < array.length; i++) {
18       if (max < array[i]) {
19         max = array[i];
20       }
21     }
22     return max;
23   }
24   public double calcularMinim(double[] array) {
25     double min = array[0];
26     for (int i = 1; i < array.length; i++) {
27       if (min > array[i]) {
28         min = array[i];
29       }
30     }
31     return min;
32   }
33   public double calcularMitjana(double[] array) {
34     double suma = 0;
35     for (int i = 0; i < array.length; i++) {
36       suma = suma + array[i];
37     }
38     return suma/array.length;
39   }
40 }
```

Tot i no ser un programa molt llarg, s'aplicarà modularitat. D'entrada, una característica que sí que es pot identificar és el fet que hi ha tres mètodes de temàtica similar: fer operacions típiques basades en recorreguts sobre *arrays* de reals (`calcularMaxim`, `calcularMinim`, `calcularMitjana`). Aquesta mena

d'operacions són molt generals i de ben segur que se solen usar en diferents programes. Per tant, pot tenir sentit declarar-los en una classe a part que serveixi com un repertori general de mètodes per treballar amb *arrays* de reals.

Definició de classes addicionals

Per dividir un programa en classes diferents, només cal crear tants fitxers com classes es vol, cadascun amb el seu nom i definició pròpia, tal com heu fet fins ara. D'entre totes elles alguna haurà de ser la classe principal (en aquest cas, que contingui com a mínim els mètodes `main` i `inici`). La resta contindrà els mètodes que es vulgui distribuir.

Si a l'exemple volem que `RegistreNotes` sigui la classe principal i declarar els tres mètodes per treballar amb *arrays* de reals en una nova classe anomenada `CalculsArrayReals`, caldria crear un nou fitxer amb el contingut següent:

```
1  public class CalculsArrayReals {
2    public double calcularMaxim(double[] array) {
3      double max = array[0];
4      for (int i = 1; i < array.length; i++) {
5        if (max < array[i]) {
6          max = array[i];
7        }
8      }
9      return max;
10   }
11   public double calcularMinim(double[] array) {
12     double min = array[0];
13     for (int i = 1; i < array.length; i++) {
14       if (min > array[i]) {
15         min = array[i];
16       }
17     }
18     return min;
19   }
20   public double calcularMitjana(double[] array) {
21     double suma = 0;
22     for (int i = 0; i < array.length; i++) {
23       suma = suma + array[i];
24     }
25     return suma/array.length;
26   }
27 }
```

Fora de la manca d'un mètode principal a les classes addicionals, totes les classes són exactament iguals a nivell de sintaxi. Tot el que heu après es pot aplicar per a qualsevol d'elles (definició i ús de constants, variables globals, àmbit de variables, etc.).

Ús de classes addicionals

Un cop declarats en una classe a part, els mètodes poden ser eliminats de la classe original (`RegistreNotes`), ja que en cas contrari tindríeu codi repetit. Precisament, en un programa modular, el que succeeix és que des d'una classe s'invoquen mètodes declarats a una classe diferent. Però cada mètode només està escrit un únic cop entre totes les classes del programa.

La invocació de mètodes que no estan escrits en el mateix fitxer és diferent del cas que heu vist fins ara, on totes estan escrites dins el fitxer d'una mateixa classe. Ja no es pot dur a terme simplement escrivint el nom del mètode (junt amb els seus paràmetres, si escau). Primer cal un pas previ d'inicialització, a partir del qual es permet la invocació dels mètodes externs. En dur a terme aquest procés, s'assigna un identificador a partir del qual és possible invocar els mètodes d'aquella classe, usant-lo com a prefix a la invocació. Sense aquest pas previ, és impossible invocar mètodes escrits a altres classes.

La sintaxi per fer-ho és la següent:

```
NomClasse identificador = new NomClasse();
identificador.nomMètode(paràmetres);
```

El comportament de la invocació del mètode és idèntica a quan es fa sobre un mètode escrit a la mateixa classe (ús de paràmetres, avaluació d'acord al seu valor de retorn). Només canvia la sintaxi, ja que es requereix aquest prefix producte de la inicialització. Cal dir que, donades diverses invocacions a mètodes d'una mateixa classe externa, només és necessari fer la inicialització una única vegada. L'àmbit i la validesa de l'identificador és el mateix que una variable.

En nomenclatura Java, el procés d'inicialització s'anomena formalment instanciació de la classe.

En el cas de l'exemple, el codi de la classe `RegistreNotes` canviaria ara pel següent:

```
public class RegistreNotes {
  public static void main(String[] args) {
    RegistreNotes programa = new RegistreNotes();
    programa.inici();
  }
  public void inici() {
    double[] notes = {2.0, 5.5, 7.25, 3.0, 9.5, 8.25, 7.0, 7.5};
    //Per cridar els mètodes cal inicialitzar la classe que els conté
    CalculsArrayReals calculador = new CalculsArrayReals();
    //Un cop fet, cal cridar-los usant com a prefix l'identificador
    double max = calculador.calcularMaxim(notes);
    double min = calculador.calcularMinim(notes);
    double mitjana = calculador.calcularMitjana(notes);
    System.out.println("La nota màxima és " + max + ".");
    System.out.println("La nota mínima és " + min + ".");
    System.out.println("La mitjana de les notes és " + mitjana + ".");
  }
}
```

Reutilització de mòduls

Com ja s'ha dit, el motiu principal per dividir un programa en mòduls és poder reusar-los en altres programes on cal fer tasques iguals. Per exemple, suposeu que se us demana fer un programa per dur a terme un registre de temperatures. En aquest, es vol mostrar la diferència màxima entre les temperatures enregistrades (quant hi ha entre el valor màxim i el mínim). Analitzant aquest problema, ja podeu veure a primera vista que cal treballar amb un *array* de valors reals amb els quals fer operacions típiques sobre llistes de valors: calcular el màxim i el mínim. Precisament algunes de les tasques que tot just acabeu de resoldre per gestionar notes.

Fins ara, en un cas com aquest, simplement podríeu agafar el programa en el qual ja s'ha resolt aquest problema i fer "copiar i enganxar" del codi al nou programa. Mitjançant la modularitat, no cal fer això. N'hi ha prou a incorporar la classe on hi ha els mètodes que us interessa usar en el nou programa, i invocar-los directament. Per tant, per a aquest nou programa podríeu incorporar una còpia de la classe `CalculsArrayReals` i fer la classe principal següent:

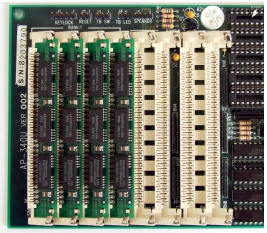
```
1  public class RegistreTemperatures {
2    public static void main(String[] args) {
3      RegistreTemperatures programa = new RegistreTemperatures();
4      programa.inici();
5    }
6    public void inici() {
7      double[] temps = {20.0, 21.5, 19.0, 18.5, 17.5, 19.0, 22.25, 21.75};
8      //Per cridar els mètodes cal inicialitzar la classe que els conté
9      CalculsArrayReals calculador = new CalculsArrayReals();
10     //Un cop fet, cal cridar-los usant com a prefix l'identificador
11     double max = calculador.calcularMaxim(temps);
12     double min = calculador.calcularMinim(temps);
13     double dif = max - min;
14     System.out.println("La diferència de temperatura màxima " + dif + ".");
15   }
16 }
```

Repte 1. Feu un programa que, donats dos *arrays* de valors reals, us digui quin dels dos té el valor mitjà més alt. Aprofiteu el principi de modularitat per fer-lo, de manera que la seva classe principal només tingui definits els mètodes `main` i `inici`.

1.1.4 Avantatges de la modularitat usant classes addicionals

Un cop vist que hi ha la possibilitat de crear programes modulars distribuint el seu codi en diferents classes, val la pena plantejar-se quins són els avantatges que aporta. Primer de tot, però, cal dir que aquesta aproximació només sol tenir sentit en programes de certa complexitat, o on hi ha un conjunt extens de mètodes amb funcionalitats genèriques, que clarament poden ser de gran utilitat en la creació de programes futurs. En cas contrari, quan hi ha molts pocs mètodes, no és necessari aplicar-la estrictament.

El principal avantatge de la modularitat és poder reutilitzar directament codi ja generat anteriorment en nous programes, sense haver de tornar-lo a escriure o copiar-lo per classes diferents cada cop que es vol utilitzar en diferents programes. Aquest codi sempre estarà a la mateixa classe dins de qualsevol programa, cosa que el fa més fàcil de localitzar i gestionar. D'altra banda, si mai en el futur es fan esmenes a algun dels mètodes d'aquesta classe (per exemple, arreglar un error que s'ha detectat al cap d'un temps), només cal reemplaçar la versió antiga de la classe per la nova. Això es fa simplement mirant si hi ha un fitxer i, si és així, sobreescriure'l. Compareu això amb la feinada que seria cercar els bocins de codi esmenats i reemplaçar-los usant un editor (obrir amb un editor el codi font dels vostres programes, cercar si hi ha els mètodes modificats i sobreescriure'ls amb la nova versió si és el cas).



Les classes són com els mòduls de memòria d'un ordinador. Se'n pot treure una per posar-ne una altra de millorada. Font: Appaloosa

Un altre avantatge interessant de generar un programa de manera modular és facilitar el treball en equip, una opció indispensable dins del món del desenvolupament de programari actual, on les aplicacions són complexes i calen diferents especialistes que s'han de coordinar. Distribuint el codi en diferents classes, n'hi ha prou a establir l'esquelet de totes elles (la declaració dels mètodes), i llavors es pot repartir la feina de manera que cada desenvolupador faci el codi d'unes poques classes d'entre totes. Tots poden treballar en paral·lel sense problemes, cosa que és molt més complicada si tot està dins el mateix fitxer.

Evidentment, la modularitat també té algun inconvenient menor del qual val la pena ser-ne conscient. Donada una classe creada amb anterioritat i que es vol reutilitzar, potser dels diversos mètodes continguts en realitat només en voleu fer servir uns pocs. Per tant, el programa sovint conté classes amb bocins de codi que mai s'usen. En el cas de l'exemple del registre de temperatures, el mètode per calcular la mitjana mai s'usa, però estarà definit igualment en una part del codi font. Ara bé, es considera que els avantatges de la modularitat són tan superiors a aquest inconvenient que és un mal menor sobre el qual no cal pensar-hi gaire si voleu dissenyar els vostres programes.

Els **programes modulars** permeten facilitar la tasca de desenvolupar programari complex, de manera que sigui més fàcil generar i mantenir el codi, especialment quan es treballa en equip.

1.2 Biblioteques de classes: packages

Normalment, per aplicar el principi de modularitat dins d'un programa n'hi ha prou a organitzar els mètodes dins de classes diferenciades, d'acord a algun criteri d'ordenació o de reutilització en el futur. Ara bé, per a casos on es vol crear un programa amb un cert grau de complexitat, pot succeir que el nombre de classes resultants també sigui força gran. En un cas com aquest, també podria ser convenient disposar d'un mecanisme que permeti organitzar conjunts de classes dins d'un programa, de la mateixa manera que una classe organitza un conjunt de mètodes. Això permet aplicar modularitat a conjunts de classes en bloc, en lloc de fer-ho a nivell individual, i també fer-les més fàcils de localitzar i gestionar.

Una biblioteca de classes, o **package** en Java, és un conjunt de classes vinculades entre elles d'acord a algun criteri temàtic o d'organització del seu codi.

Sovint, els criteris per decidir com dividir les classes d'un programa en conjunts de *packages* diferents són força subjectius. En aquest cas, el que heu de tenir en compte és que es tracta d'una eina de la qual disposeu lliurement per organitzar els diferents fitxers d'un programa al vostre gust, de la mateixa manera que es pot

organitzar un conjunt de fotografies referents a diversos viatges de moltes maneres usant una estructura de carpetes.

En aquest sentit, cal dir que, si bé la majoria dels exemples amb els quals heu treballat fins ara (i amb els quals es continuarà treballant) són programes amb un nombre de mètodes massa limitat com per justificar de manera clara l'ús de *packages*, es tracta d'un element prou important de Java com per fer necessari conèixer el seu funcionament.

1.2.1 Definició de packages

Per assignar un conjunt de classes a un *package*, primer cal triar un identificador, que servirà com el nom d'aquest. Com sempre, aquest identificador hauria de ser autoexplicatiu respecte a la funció de les classes que contindrà. Un cop triat, per fer que una classe hi pertanyi, n'hi ha prou que a la primera línia del seu fitxer amb el codi font s'escrigui la sentència mostrada tot seguit. La seva aparició ha de ser estrictament sempre a la primera línia de text, fins i tot abans de la declaració de classe (`public class...`).

```
1 package identificadorPackage;
```

URL

Inicials de "Localitzador Uniforme de Recursos" (*Uniform Resource Locator*, en anglès). És la cadena de caràcters que informa al navegador on us voleu connectar.

Cal fer notar, doncs, que el concepte *package* en Java no té una entitat pròpia diferenciada. Al contrari que les classes, no és cap fitxer concret que cal editar. Serà donat implícitament per totes les classes que es declaren com a part d'ell en el seu codi font.

Com a convenció de codi, l'identificador d'un *package* hauria de seguir un format especial, de la mateixa manera que hi ha convencions per als noms de les classes, mètodes i variables. El programa no deixa de funcionar si no es fa així, però val la pena seguir aquesta convenció, ja que és el format usat a la immensa majoria de programes en Java. En fer-ho, el vostre codi aporta la imatge d'estar fet per programadors seriosos i acurats. En aquest cas, es tracta que els noms dels *packages* s'escriuen sempre tots en minúscula i separant paraules diferents per un punt, de manera que tenen un aspecte semblant a l'URL escrit a la barra d'adreça d'un navegador web. Per exemple:

Els creadors del llenguatge Java desaconsellen usar el *package* per defecte.

```
1 package unitat5.apartat1.exemple1;
```

Estrictament, totes les classes del Java pertanyen a algun **package**. En cas de no incloure cap sentència **import**, es considera que aquella classe pertany a un *package* especial anomenat per defecte (*default package*). No és possible crear una classe que "no pertanyi a cap *package*". Per tant, tots els programes que heu creat fins al moment s'han compost de classes que formaven part del *package* per defecte, tot i no usar la sentència **import**.

La manera com té el Java d'ordenar les classes dins un programa en *packages* (inclòs el *package* per defecte) comporta dues restriccions que heu de tenir sempre en compte:

- Donada una classe, aquesta únicament pot pertànyer a un *package*. No és possible usar més d'una vegada la sentència **package** en el codi font d'una classe.

- Donat un *package*, a dintre seu mai hi poden haver dues classes amb el mateix nom. Per norma general, podeu considerar que aquesta regla s'aplica ignorant majúscules i minúscules.

Packages i IDE

Per tal de facilitar la feina del desenvolupador, la majoria d'IDE actuals, en realitat, sí que tracten els *packages* com una entitat especial dins dels seus projectes. Així, doncs, abans de poder crear cap classe continguda dins d'un *package*, és necessari crear-lo explícitament de manera individual. Un cop fet, és possible afegir-hi classes, de manera semblant a com es gestionarien fitxers dins d'una carpeta. Quan es crea una nova classe dins de cada *package*, l'IDE ja s'encarrega d'incloure automàticament la sentència **import** a l'inici del codi font. Addicionalment, mentre no es crei cap *package* amb un nom concret, també solen mostrar explícitament el *package* per defecte, on hi van incloent les noves classes que creeu.

Tingueu en compte, però, que aquesta característica només és un mecanisme de la interfície d'usuari dels IDE per fer la seva gestió més senzilla.

Repte 2. Genereu un projecte en el vostre IDE amb un *package* anomenat `unitat5.apartat1.repte2`. Feu que dins d'aquest *package* s'incloguin les classes de l'exemple del registre de les notes i executeu-lo, comprovant que funciona correctament.

1.2.2 Criteris per crear biblioteques de classes

Normalment, els criteris usats per dividir conjunts de classes en biblioteques diferents solen ser:

- Un programa complet, una jerarquia de *packages*.
- Molts programes petits, una jerarquia de *packages*.
- Un conjunt de classes sense un programa principal, un *package*.

Com que l'ús de biblioteques no és més que un mecanisme per ordenar les classes d'un programa, inicialment es pot fer una analogia amb l'ordenació d'un conjunt de fotografies de diferents viatges dins de carpetes. Cada viatge equival a un programa diferent en aquesta analogia.

1. Un programa complet, una jerarquia de packages.

   Inicialment, podríeu tenir una carpeta per a cada viatge, en cadascuna de les quals hi ha totes les fotos associades directament. Això seria equivalent a usar cada biblioteca per encapsular un programa complet, compost per la

seva classe principal i un seguit de classes addicionals. Així, doncs, s'assignaria el mateix identificador a totes les classes que conformen el programa. Sota aquest criteri, les classes dels exemples, en ser programes diferents, podrien anar dins dels diferents *packages* `unitat5.apartat1.exemple1`, `unitat5.apartat1.repte1`, etc.

Partint d'aquest criteri, però, podríeu considerar posar algunes de les les fotos d'un viatge, a la seva vegada, en carpetes segons el dia, o el monument visitat, de manera que hi ha una carpeta amb fotos "generals" i algunes subcarpetes que agrupen visites concretes. De fet, aquest és el cas més típic quan s'usen biblioteques en Java per encapsular aplicacions. Hi ha una biblioteca on es desa la classe principal i després d'altres on s'agrupen les classes addicionals per tipus de funció. En aquest cas, el que es fa és estendre l'identificador base amb noms que indiquin el tipus de classes addicionals.

Tingueu en compte que tots aquests *packages* són diferents a nivell d'estructura del programa. Però el nom se sol triar estratègicament perquè a partir d'ell es dedueixi ja a simple vista que hi ha una relació jeràrquica entre les classes. Les classes del nivell més alt de la jerarquia fan ús de les de nivells més baixos.\\Més endavant, si l'aplicació fa altres accions d'altres temàtiques, es podrien afegir un nou *package*. Per exemple, per tractar la lectura de dades pel teclat es podria generar el *package* **unitat5.apartat1.exemple1.entrada**, en el qual anirien les classes vinculades a l'entrada de dades.

Per exemple, en el cas de l'exemple mostrat, podríem tenir dos *packages*:

- `unitat5.apartat1.exemple1`, que engloba la classe principal.
- `unitat5.apartat1.exemple1.arrays`, que engloba totes les classes per tractar les dades. Ara mateix només contindria `CalculsArrayReals`, però més endavant podria incloure altres classes per extreure informació addicional de les dades del programa (predir temperatures en el futur, fer histogrames o gràfics, etc.)

2. Molts programes petits, una jerarquia de packages

   Una altra opció possible és englobar més d'un programa en un únic *package*. Això se sol fer quan es vol gestionar fàcilment un conjunt de programes molt simples, normalment d'una sola classe, que sovint usen les mateixes classes addicionals. Seguint l'analogia, si heu fet diversos viatges molt breus, o de passada, a una mateixa ciutat, de manera que de cada viatge només teniu 3 o 4 fotos, no val la pena diferenciar-los, surt més a compte posar-les totes barrejades a la mateixa carpeta, anomenada d'acord a la ciutat.

   L'objectiu principal és evitar l'explosió en el nombre de *packages*, que al final acaben tenint una o dues classes. Ara bé, això té una particularitat, i és que dins un mateix *package* hi ha més d'una classe principal barrejada. De fet, un *package* no té cap límit en el nombre de classes principals que pot contenir. N'hi ha prou que, dins el vostre entorn de desenvolupament, trieu de manera adient quina és la que voleu que s'executi d'entre totes les existents. Java s'encarrega de la resta.

Si us pareu a pensar un moment, aquest és el cas precisament de la majora d'exemples o reptes proposats, en ser programes amb molts pocs fitxers i que sovint compartiran algunes classes addicionals, per la qual cosa és el que s'usarà a partir d'ara. Per exemple, organitzant les classes segons els *packages* anomenats `unitat5.apartat1.exemples` i `unitat5.apartat1.reptes`.

3. Un conjunt de classes sense programa principal, un package

Finalment, poden haver-hi classes que es consideren de propòsit general, útils en molts programes diferents. Això pot succeir si voleu deixar la porta oberta a reutilitzar classes que esteu implementant en un altre programa nou, més endavant. O sigui, fotografies que apliquen a més d'un viatge. Normalment, el que faríeu es crear una carpeta a part i posar-les-hi, en lloc de tenir-les en la d'un viatge concret. En un cas com aquest, la introducció del concepte de biblioteca de classes planteja una nova possibilitat a l'hora de crear codi. Concretament, el cas invers de l'anterior, el desenvolupament de repertoris de classes que no conformen pròpiament un programa, ja que cap d'elles és la principal.

En l'exemple vist, aquest és el cas de la classe `CalculsArrayReals`. En un cas com aquest, les classes de propòsit general se solen crear en una biblioteca a part. De ben segur que un seguit de classes per fer operacions amb *arrays* s'usaran més enllà de la unitat 5. Per tant, no té sentit definir-les en un *package* anomenat `unitat5.apartat1.exemples.arrays`, sinó usant un nom que no estigui associat a aquesta jerarquia. En aquest cas, per exemple, podria ser el *package* `utilitats.arrays`.

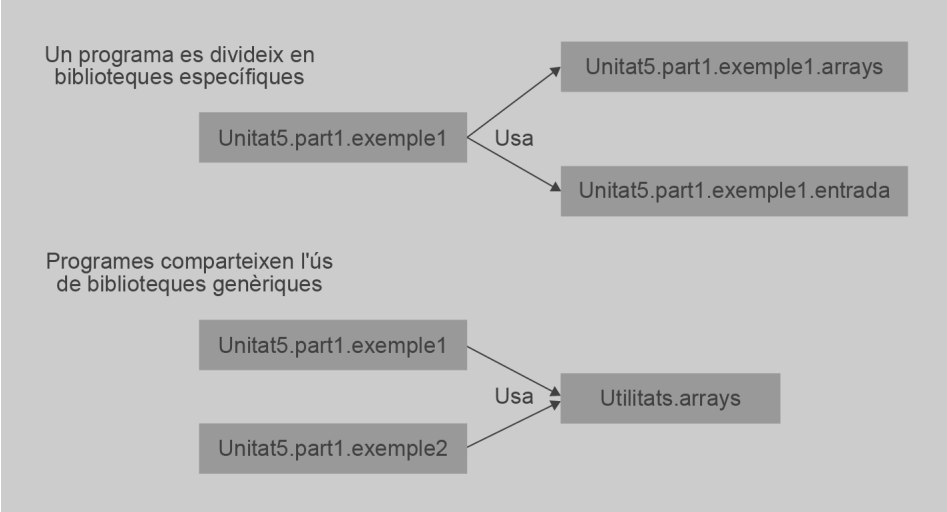
Després d'una certa experiència amb Java al final és evident quina mena d'operacions típiques s'acaben repetint en tots els programes.

Encara més, es pot plantejar el desenvolupament d'un conjunt de classes en una biblioteca que no siguin producte parcial de la generació d'un programa, que posteriorment han resultat ser reutilitzades, sinó que ja des del punt de partida s'han creat com eines de suport per a programes futurs. Per tant, no és inconvenient tenir *packages* sense cap classe principal (de fet, és el cas més habitual). Un cop s'identifica que un conjunt de classes s'usa en molts programes diferents, el millor és fer que formin part de la seva pròpia biblioteca independent, fora de la jerarquia del nom de cap aplicació concreta.

La figura 1.2 mostra un esquema que contrasta aquest criteri per tal de gestionar classes addicionals amb el de crear una relació jeràrquica entre el *package* amb la classe principal.

En darrera instància, però, heu de tenir en compte que no hi ha una resposta totalment correcta a l'hora d'englobar classes dins de biblioteques. Només s'han enumerat uns criteris que se solen usar dins els programes professionals realitzats en Java. Però, en el fons, els *packages* només són una eina per organitzar les classes de manera coherent perquè us resulti més còmoda a vosaltres.

FIGURA 1.2. Criteris d'estructuració de programes en biblioteques



1.2.3 Ús de classes d'altres packages

Per referir-nos a una classe simplement s'havia de dir el seu nom (o el nom del seu fitxer). Atès que tota classe sempre pertany a una biblioteca, i dins de biblioteques diferents hi poden haver classes amb igual nom, pot ser interessant disposar d'una eina per dir exactament de quina classe s'està parlant en cada cas.

> El **nom qualificat d'una classe** és la combinació de l'identificador del seu *package* junt amb el seu nom, separats per un punt.

Exemples de noms qualificats, partint dels exemples anteriors, serien `unitat5.apartat1.exemples.RegistreNotes` o `utilitats.arrays.CalculsArrays`. Amb un nom qualificat queda totalment clar a quin *package* pertany la classe a la qual us esteu referint en un moment donat, independentment del context.

El nom qualificat de les classes és útil ja que des del codi d'una classe que pertany a un *package* concret, només es poden usar directament classes amb què comparteixi el mateix nom de *package*. Si s'usa una classe d'un de diferent, el compilador donarà un error, dient que no la reconeix. Això passaria si, en organitzar les classes `RegistreNotes` i `CalculsArrayReals` en biblioteques diferents, des de la primera s'intenta usar directament la segona. Ja no funciona. Cal establir algun mecanisme per indicar exactament quina classe s'està usant. Java ofereix tres maneres, més o menys relacionades.

1. Inicialització usant el nom qualificat

D'una banda, per al cas de classes d'altres *packages*, es pot usar el seu nom qualificat dins del codi font per referir-se a la seva inicialització.

```java
package unitat5.apartat1.exemples;
public class RegistreNotes {
  ...
  //Ús del nom qualificat per accedir a una classe d'un altre package
  utilitats.arrays.CalculsArrayReals calculador = new utilitats.arrays.
      CalculsArrayReals();
  ...
}
```

```java
package utilitats.arrays;
public class CalculsArrayReals {
  //Codi
  ...
}
```

2. Importació explícita

Una altra manera, molt més còmoda si una classe s'usa diverses vegades dins el codi, és **importar-la** prèviament a la capçalera del fitxer de codi font, entre la declaració del *package* i la declaració `public class...`. Per fer aquesta importació, cal especificar el nom qualificat de la classe que es vol usar. La sintaxi general és:

```java
import nomQualificat;
```

En cas de voler usar més d'una classe, ja sigui de la mateixa biblioteca o de diferents, caldrà posar tantes sentències **import** com correspongui. Per exemple, si useu cinc classes que no pertanyen al mateix *package* que la classe que esteu creant, caldrà incloure cinc sentències **import** per separat. En aquest procés, les relacions de jerarquia entre els identificadors del *package* no importen, cal posar cada classe amb el seu *package* explícitament, una per una.

Seguint l'exemple anterior de divisió de les classes `CalculsArrayReals` i `RegistreNotes` en *packages* diferents, per poder usar la primera classe a la segona d'aquesta manera, caldria fer-ho així:

```java
package unitat5.apartat1.exemples;
//Cal importar la classe "CalculsArrayReals", ja que és d'un altre package
import utilitats.arrays.CalculsArrayReals;
public class RegistreNotes {
  //Codi
  ...
}
```

```java
package utilitats.arrays;
//No s'usa cap classe fora d'aquest package, no s'importa res
public class CalculsArrayReals {
  //Codi
  ...
}
```

3. Importació general

Finalment, hi ha un tercer mecanisme que permet importar automàticament totes les classes dins una biblioteca. Aquest es fa servir sovint quan es volen usar moltes classes del mateix *package* i importar-les una a una es fa pesat. Es tracta d'usar

un asterisc, "*", en lloc del nom de la classe en el nom qualificat. Aquest fa de comodí i equival a dir "absolutament totes les classes del *package*".

```
1 package unitat5.apartat1.exemples;
2 //S'importarien totes les classes del package utilitats.arrays
3 import utilitats.arrays.*;
4 public class RegistreNotes {
5   //Codi
6   ...
7 }
```

Per norma general, és aconsellable usar la importació explícita, ja que així a l'inici del vostre codi font sempre disposareu d'una llista detallada de quines classes d'altres biblioteques esteu usant.

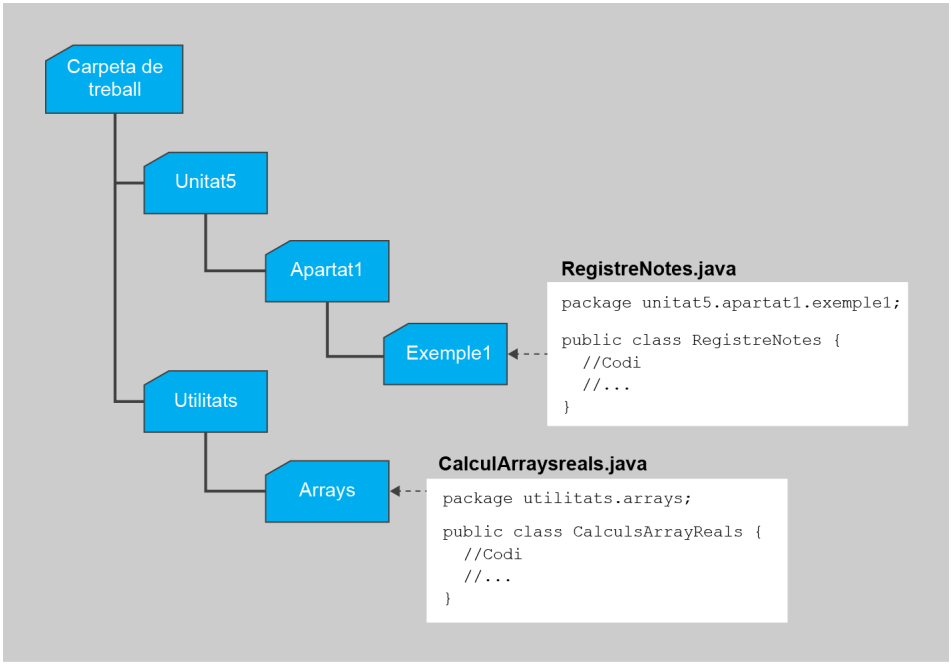
Repte 3. Modifiqueu l'exemple del registre de temperatures de manera que la classe `RegistreTemperatures` pertanyi al *package* `unitat5.apartat1.repte3` i la classe `CalculsArrayReals` al *package* `utilitats.arrays`. Assegureu-vos que el programa s'executa correctament.

1.2.4 Estructura dels fitxers dins dels packages

Cada IDE desa les classes en una carpeta considerada "de treball" diferent, però sempre estarà dins del projecte creat. Sovint s'anomena "src".

L'identificador triat com a nom d'un *package* a Java indica de quina manera s'organitzen les classes contingudes dins el sistema de fitxers del vostre ordinador. Partint de la carpeta considerada de treball, totes les classes que formen part del *package* han d'estar ubicades dins d'una jerarquia de carpetes on cada carpeta té el nom de cadascun dels textos separats per punts que conformen el nom complet del *package*. A la darrera carpeta és on estaran ubicats realment els fitxers `.java`.

Figura 1.3. Desplegament de classes en carpetes d'acord al nom del package.



Organitzar les classes en aquesta estructura de carpetes segons el seu *package* serveix per garantir dues coses importants. D'una banda, que les classes de diferents *packages* estan endreçades en carpetes diferents, fàcilment identificables, de manera que no es barregen totes. D'altra banda, també permet que dins de dos *packages* diferents hi pugui haver classes amb el mateix nom sense que impliqui cap col·lisió en el sistema de fitxers (que els fitxers `.java` se sobreescriguin).

Si se segueix amb l'exemple de les classes vinculades al registre de notes, partint de criteri de fer que `CalculsArrayReals` pertanyi a una biblioteca comuna, les classes estarien distribuïdes de la manera que mostra la figura 1.3.

Si en un moment donat es volen usar les classes del *package* `utilitats.arrays` en un altre programa, no n'hi ha prou a copiar simplement els fitxers .java al directori de la nova aplicació. Cal que aquestes es copiïn mantenint la seva estructura de carpetes. Per tant, caldria copiar l'estructura existent partint de la carpeta **utilitats**.

Packages i IDE (II)

Els IDE actuals també s'encarreguen de gestionar l'organització de l'estructura de directoris d'acord al nom dels *packages* de manera automàtica, un cop se n'ha creat un explícitament i s'hi van afegint classes. Per tant, si n'useu un, no us heu de preocupar d'aquest fet. De totes maneres, aneu molt en compte si copieu classes entre projectes, ja que cal seguir les regles tot just explicades.

1.3 L'API del llenguatge Java

Per realitzar programes modulars no sempre és necessari crear tots i cadascun dels mòduls que formen part del programa. Per sort, sovint els llenguatges de programació incorporen biblioteques de mòduls auxiliars que ja han estat completament desenvolupats pels creadors del llenguatge o altres col·laboradors. Aquests mòduls auxiliars estenen el llenguatge de programació, oferint funcionalitats que van més enllà del que permet la seva sintaxi estrictament o incorporant l'opció d'executar blocs de codi que resolen tasques que es consideren de propòsit general i que poden ser d'utilitat en una àmplia gamma de programes.

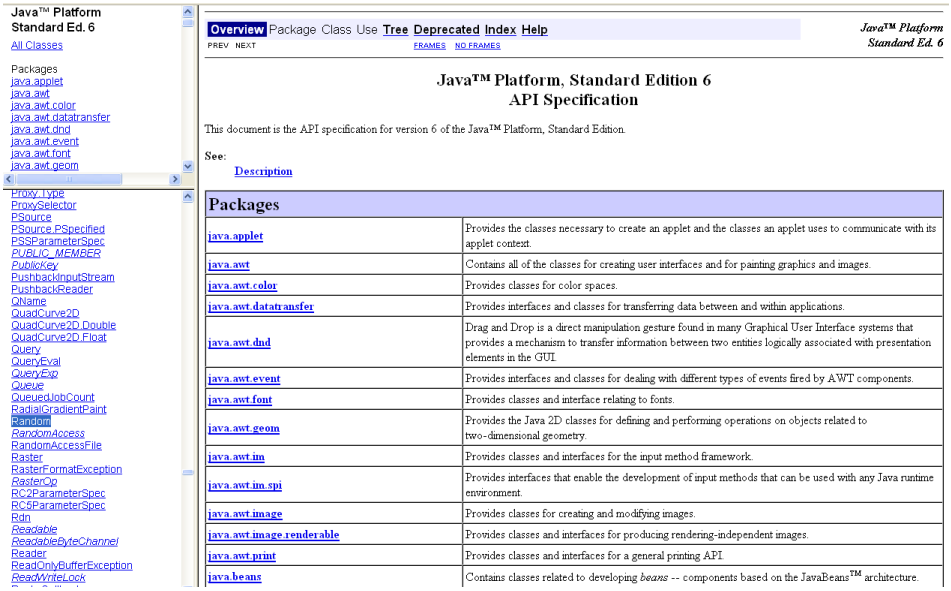
En el cas del llenguatge Java, el seu *kit* de desenvolupament (JDK) incorpora un quantiós repositori de classes, organitzades per diferents *packages* d'acord a la seva temàtica, que poden ser accedides lliurement en realitzar qualsevol programa. Aquest repositori és el que s'anomena l'**API** de Java. Ara bé, l'objectiu d'aquesta secció no és conèixer totes les classes disponibles, sinó simplement que tingueu un marc general sobre com cal usar-ne una que sigui senzilla, un cop ja us és coneguda la seva existència i que pot ser útil per fer un programa.

Per a la versió 1.6, el llistat de totes les classes del Java, junt amb la seva documentació, la podeu trobar a l'adreça d'Oracle de les Adreces d'interès del web.

Si observeu la documentació de l'API de Java, de la qual la figura 1.4 mostra una idea de la seva estructura, veureu que ofereix un enorme repertori de classes que contenen mètodes amb funcionalitats molt diverses, cosa que, de vegades, fa ben complicat trobar d'entre totes la classe que conté els mètodes ideals per a cada cas, si és que existeix. De fet, podeu considerar que intentar esbrinar

si hi ha una classe que conté mètodes que us puguin resultar útils als vostres programes a partir d'aquesta documentació és inviable. La documentació es va crear partint del supòsit que ja sabeu el nom de la classe que voleu utilitzar. Per tant, l'aproximació correcta per usar-la és cercar quina classe pot ser la que realitza les tasques que voleu dur a terme, a partir d'una font bibliogràfica externa o d'una cerca a internet, i llavors referir-se a l'API només per estudiar amb més detall quins mètodes ofereix.

FIGURA 1.4. L'API de Java es compon realment d'una gran quantitat de classes



Com a exemple, entre els *packages* que contenen les classes més populars, podeu trobar:

- `java.lang`: conté totes les classes vinculades a operacions essencials dels tipus de dades del llenguatge.
- `java.util`: una mena de calaix de sastre amb classes de propòsit general.
- `java.io`: conté totes les classes vinculades a entrada / sortida (tractament de fitxers).
- `javax.swing`: conté les classes bàsiques vinculades a la creació d'interfícies gràfiques.

L'avantatge d'aquesta aproximació és que, com que aquestes classes ja estan creades i incorporades com a part del llenguatge, si en trobeu una que ja faci la feina requerida, us estalviareu haver d'escriure parts de codi font del programa.

1.3.1 Ús de classes dins les biblioteques de Java

L'única excepció són les classes del *package* anomenat `java.lang`, que es poden usar sense haver d'importar-les.

Per poder usar qualsevol classe d'entre les definides a l'API de Java cal seguir exactament les mateixes passes que per a una classe realitzada per vosaltres, però

dins un *package* diferent: importar-la a l'inici del codi font i inicialitzar-la. Un cop fet, la sintaxi per invocar els seus mètodes és exactament la mateixa. Aquest procés només té una particularitat molt especial, i és que no cal disposar del codi font de la classe en qüestió. En estar incorporada dins el JDK, Java ja sap localitzar el seu codi automàticament sense haver-lo d'incorporar vosaltres explícitament.

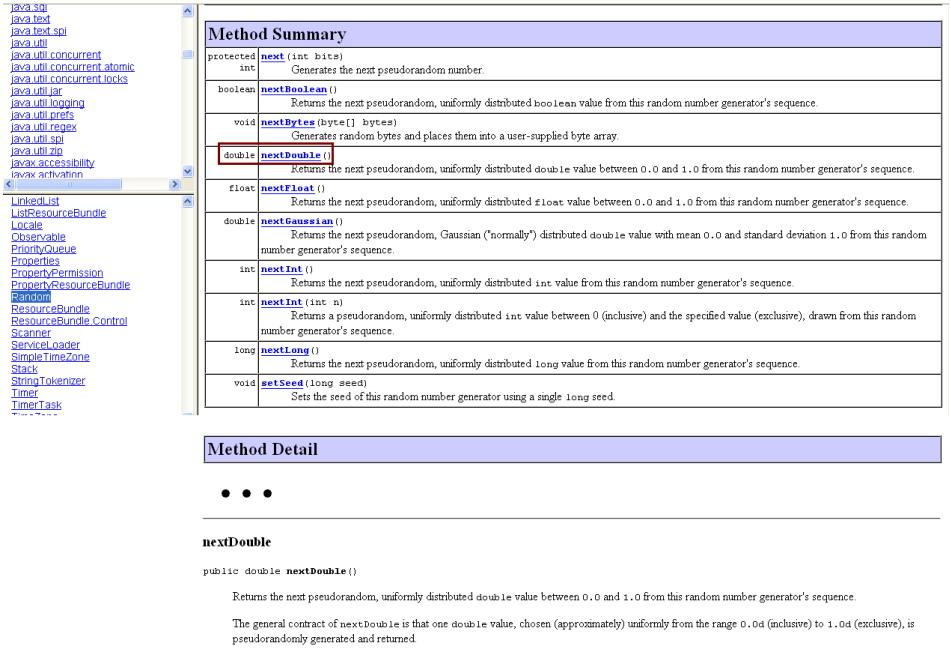
Per repassar aquest procés, s'usarà com a exemple senzill partint des de zero. Suposeu que heu de fer un programa que ha de generar dos valors reals a l'atzar, entre 0 i 100, i que els mostri per pantalla. D'entrada, resoldre aquest problema suposaria haver de crear dins del codi font algun mètode que generés aquests valors reals aleatoris. Per sort, investigant una mica fonts de documentació, no és complicat trobar que existeix una classe anomenada **Random**, ja incorporada a l'API de Java, la qual permet fer aquest tipus de tasques. Per tant, estudiant com usar els seus mètodes us podeu estalviar feina de codificació. Cercant-la a la documentació de l'API de Java, es veu que forma part del *package* `java.util`.

Sense encara saber els detalls d'aquesta classe, es pot fer l'esquelet de la classe principal del programa, amb la corresponent importació. Aneu amb compte amb el fet que aquest exemple inclou la classe dins un *package*.

```
1  package unitat5.apartat1.exemples;
2  //Importació de la classe, en estar en un altre package
3  import java.util.Random;
4  public class RealsAleatoris {
5    public static void main(String[] args) {
6      RealsAleatoris programa = new RealsAleatoris();
7      programa.inici();
8    }
9    public void inici() {
10     //Cal generar un valor a l'atzar
11     //Cal generar-ne l'altre
12     //Mostrar-los
13   }
14 }
```

El vostre programa només cal que es compongui d'aquesta classe. En formar part de l'API de Java, l'ús de la classe **Random** no implica la incorporació de cap altre fitxer de codi font. Un cop ja està tot llest per usar-la, caldrà mirar quins mètodes ofereix per generar reals a l'aztar. Per això cal consultar la documentació, com mostra la figura 1.5. Dins l'apartat de resum de mètodes (*Method Summary*) hi ha una llista dels mètodes amb una descripció general. D'entre tots, se'n pot trobar un que sembla generar valors reals a l'atzar, anomenat `nextDouble()`. Si es vol veure amb més concreció què fa el mètode, la documentació proporciona una descripció més detallada (*Method Detail*), en què sempre val la pena fer una ullada. De la informació obtinguda, però, es pot veure que en realitat aquest mètode genera valors a l'atzar entre 0.0 i 1.0. Per tant, si es volen crear valors entre 0 i 100, no fa tota la feina. Caldrà fer una multiplicació del seu resultat per 100. No sempre trobareu mètodes que fan exactament el que voleu, però n'hi ha prou que us estalviïn les feines més complicades.

FIGURA 1.5. Estudiant els mètodes oferts per una classe



El codi per generar dos valors reals a l'atzar quedaria com mostra el codi següent. A nivell de sintaxi, tot és igual que si fos una classe addicional generada per vosaltres mateixos. Comproveu que, efectivament, per a diferents execucions els valors mostrats per pantalla varien a l'atzar.

```
1  ...
2    public void inici() {
3      //Inicialització
4      Random rnd = new Random();
5      //Ús
6      double valorA = rnd.nextDouble() * 100;
7      double valorB = rnd.nextDouble() * 100;
8      System.out.println("S'han generat els valors " + valorA  + " i " + valorB);
9    }
10   ...
```

Una de les característiques essencials de Java és la seva modularitat inherent.

Si feu una mica de memòria, veureu que el procés tot just descrit ja l'heu usat anteriorment en diverses ocasions: cada cop que us calia llegir dades pel teclat i heu usat `Scanner`. Si mireu la documentació de l'API de Java, trobareu aquesta classe dins el mateix *package* `java.util`. De fet, `String` també és una classe incorporada dins de l'API de Java, només que en formar part del *package* `java.lang`, és un cas especial i no cal importar-la explícitament.

Així, doncs, tot i que no n'heu estat conscients, molts dels vostres programes fets fins ara ja eren, en certa manera, modulars!

1.3.2 Inicialització amb paràmetres

Algunes classes de Java tenen una particularitat, i és que a l'hora d'inicialitzar-les cal especificar un conjunt d'informació addicional en forma de paràmetres. Aquest

cas es tractarà aquí només de manera superficial i a nivell d'ús dins del context de les classes ofertes dins l'API de Java, ja que és força habitual, però no a nivell de com poder fer que les classes generades per vosaltres siguin capaces d'acceptar-ne, ja que és més complex. L'objectiu només és que disposeu del rerefons bàsic per poder treballar amb algunes de les classes més senzilles de l'API.

Fins al moment, s'ha dit que per inicialitzar una classe per tal de poder invocar els mètodes que ofereix calia fer:

```
1 NomClasse identificador = new NomClasse();
```

Però hi ha casos en què, per inicialitzar correctament la classe, cal afegir un conjunt de valors entre els parèntesis, de manera idèntica a com es faria en invocar un mètode. Si no es posen aquests paràmetres quan pertoca, hi haurà un error de compilació.

```
1 NomClasse identificador = new NomClasse(paràmtres);
```

De fet, sense dir-ho explícitament ja heu treballat anteriorment amb un cas com aquest, ja que és tot just el que passa amb la classe **Scanner**. Per inicialitzar-la correctament li cal un paràmetre indicant quin sistema d'entrada ha de processar.

```
1 Scanner lector = new Scanner(System.in);
```

En Java, el text `System.in` identifica l'entrada estàndard, la qual per defecte és el teclat.

Per veure si cal o no incloure un paràmetre en inicialitzar una classe, caldrà cercar-lo a la seva documentació. Concretament, a l'apartat anomenat resum de constructors (*Constructor Summary*) indica la sintaxi de la part dreta de la inicialització. El nom d'aquest apartat es deu al fet que, formalment, quan s'inicialitza una classe es diu que s'invoca un dels seus **constructors**. Es pot triar qualsevol de les opcions ofertes a la llista de constructors, però segons quina s'esculli, la classe es pot comportar d'una manera diferent.

La figura 1.6 mostra aquest apartat per a la classe `Random`. En aquest cas, es pot veure que hi ha dues maneres diferents d'inicialitzar la classe abans de cridar els seus mètodes. Es pot fer com sempre, sense paràmetres, `Random()`, o bé usant un valor de tipus `long` com a paràmetre, `Random(long seed)`.

FIGURA 1.6. Constructors de la classe Random

Constructor Summary

Random() Creates a new random number generator.
Random(long seed) Creates a new random number generator using a single long seed.

Repte 4. Modifiqueu el programa anterior per mostrar per pantalla dos valors reals aleatoris de manera que la classe `Random` s'inicialitzi amb un paràmetre de tipus `long`. Aquest nou programa hauria de formar part d'un *package* anomenat `unitat5.apartat1.reptes`. Executeu-lo diverses vegades. Quina és la diferència entre usar un constructor o un altre en inicialitzar la classe?

1.4 Mètodes estàtics

Per norma general, per poder invocar mètodes d'una classe cal inicialitzar-la prèviament. Ara bé, d'entre tots els mètodes proporcionats dins de les classes disponibles a l'API de Java, hi ha un petit subconjunt amb una peculiaritat.

> Els mètodes **estàtics** poden ser invocats sense haver d'inicialitzar la seva classe. S'identifiquen perquè a la documentació apareixen marcats amb la paraula clau **static**.

Novament, aquesta secció se centra en l'ús dins del context de l'API de Java, i no pas en com generar-ne a les vostres classes o quina és la seva justificació dins de Java. Si bé l'explicació és genèrica, se centrarà en les dues classes més importants basades en aquests tipus de mètode, les quals us poden ser de gran ajut per fer els vostres programes: `java.lang.Math` i `java.util.Arrays`. Si mireu la documentació d'aquestes dues classes, veureu que gairebé tots els seus mètodes són estàtics. A la figura 1.7 podeu veure un detall d'alguns dels mètodes de la classe **Math**. Fixeu-vos com estan identificats amb la paraula **static** a l'inici de la seva declaració.

FIGURA 1.7. Mètodes estàtics de la classe Math

static long	**abs**(long a) Returns the absolute value of a long value.
static double	**acos**(double a) Returns the arc cosine of a value; the returned angle is in the range 0.0 through *pi*.
static double	**asin**(double a) Returns the arc sine of a value; the returned angle is in the range -*pi*/2 through *pi*/2.
static double	**atan**(double a) Returns the arc tangent of a value; the returned angle is in the range -*pi*/2 through *pi*/2.

Per invocar aquests mètodes, la sintaxi correcta és posar com a prefix el nom de la classe directament, en lloc de l'identificador fruit de la inicialització. A part d'això, el seu comportament és idèntic a qualsevol altre mètode (com una expressió que avalua a un resultat).

```
1 NomClasse.nomMetode(paràmetres);
```

Per exemple, donat el mètode estàtic `sqrt` definit a la classe `Math`, el qual accepta un únic paràmetre d'entrada de tipus real, per invocar-lo i desar el resultat en una variable només caldria el codi que es mostra tot seguit. En aquest cas, es calcula l'arrel quadrada de 36.

```
1 double resultat = Math.sqrt(36);
```

Fixeu-vos com no cal cap inicialització prèvia i, per invocar-los, s'usa abans de l'identificador del mètode el nom de la classe, i no pas d'una variable.

1.4.1 La classe Math

Aquesta classe pertany al *package* `java.lang`, i per tant pot ser usada sense haver d'importar-la. Ofereix un ventall de mètodes estàtics per realitzar operacions matemàtiques avançades, que no es poden dur a terme amb cap dels operadors aritmètics de què disposa Java (suma, resta, multipliació, divisió i mòdul).

Tot seguit es mostren alguns dels mètodes més rellevants, associats a operacions matemàtiques relativament comunes i simples.

Arrodoniment

Un mètode interessant és el que permet arrodonir valors reals.

`round(valor)`, només té com a paràmetre d'entrada un valor real (ja sigui de tipus `float` o `double`) i avalua sempre un enter (`int`). L'arrodonimet es fa a l'alça o a la baixa depenent de si la seva part decimal és superior o igual a 0.5, o menor, respectivament.

Per exemple, suposeu que voleu fer un programa que, donat un valor real a l'atzar entre 0 i 1, mostri el seu arrodoniment (a l'alta o a la baixa, segons si el valor decimal és menor o superior a 0.5). El codi d'aquest programa seria el que es mostra tot seguit. En aquest exemple, fixeu-vos en la diferència en el codi pel que fa a la manera com es crida en els mètodes de les classes `Random` i `Math`.

```
1  package unitat5.apartat1.exemples;
2  import java.util.Random;
3  public class ArrodonirReal {
4    public static void main(String[] args) {
5      ArrodonirReal programa = new ArrodonirReal();
6      programa.inici();
7    }
8    public void inici() {
9      //Inicialització de Random
10     Random rnd = new Random();
11     //Ús per generar un real entre 0 i 1
12     double valor = rnd.nextDouble();
13     System.out.println("El valor real generat és " + valor);
14     //Ús del mètode estàtic. No cal inicialitzar res, es pot usar directament.
15     //No cal importar-la, ja que pertany a java.lang
16     long arrodonit = Math.round(valor);
17     System.out.println("El valor arrodonit és " + arrodonit);
18   }
19 }
```

Elevar a exponent i arrel quadrada

Tot i que d'altres llenguatges sí que disposen d'operador aritmètic per elevar a un exponent (a^b), a Java no n'hi ha. Per això, la classe **Math** ofereix el mètode estàtic per poder dur a terme aquesta operació.

`pow(base, exponent)` accepta dos paràmetres, que només poden ser de tipus `double`, de manera que la seva invocació avalua el primer elevat al segon ($base^{exponent}$).

Junt amb aquest mètode, també es disposa del contrari, en aquest cas només per al càlcul de l'arrel quadrada.

La paraula `sqrt` abreuja *square root*, "arrel quadrada" en anglès.

`sqrt(valor)` només té un paràmetre, que només pot ser un real de tipus `double`, el qual avalua el valor real resultant de fer el càlcul de l'arrel quadrada de `valor`. Cal anar amb molt de compte de no invocar mai aquest mètode amb un paràmetre d'entrada negatiu, ja que el resultat serà sempre incorrecte.

Tot just es mostra un exemple del seu ús, on s'aplica el teorema de Pitàgores per calcular la hipotenusa d'un triangle:

```
1  package unitat5.apartat1.exemples;
2  public class Pitagores {
3    public static void main (String[] args) {
4      Pitagores programa = new Pitagores();
5      programa.inici();
6    }
7    public void inici() {
8      double base = 4.5;
9      double altura = 10;
10     //S'eleva al quadrat cada costat del triangle
11     double sumaQuadrats = Math.pow(base,2) + Math.pow(altura, 2);
12     double hipotenusa = Math.sqrt(sumaQuadrats);
13     System.out.println("El valor de la hipotenusa és " + hipotenusa);
14   }
15 }
```

Màxim i mínim

Per acabar aquest petit mostrari, és possible calcular ràpidament el màxim o el mínim entre dos valors numèrics usant la classe **Math**, ja que també ofereix mètodes per fer-ho.

`max(a, b)` usa dos paràmetres d'entrada de qualsevol tipus primitiu numèric (`int`, `long`, `float` o `double`) i avalua el valor màxim d'entre tots dos. Cal tenir en compte que els dos paràmetres d'entrada sempre han de ser del mateix tipus, i el tipus del valor resultant és el mateix que el dels dos paràmetres d'entrada. Si es comparen dos `long`, el resultat també és un `long`, per exemple.

`min(a, b)` és el mateix que l'anterior, però per al càlcul del mínim.

Tot seguit es mostra un petit programa que mostra com invocar-los.

```
1  package unitat5.apartat1.exemples;
2  public class MinMax {
3    public static void main (String[] args) {
4      MinMax programa = new MinMax();
5      programa.inici();
6    }
7    public void inici() {
8      double resReal = Math.max(1.2, 4.5);
9      System.out.println("El valor màxim és " + resReal);
10     int resEnter = Math.min(-8, 20);
11     System.out.println("El valor mínim és " + resEnter);
```

```
  }
}
```

1.4.2 La classe Arrays

L'existència de la classe **Arrays** no fa menys important dominar els esquemes de manipulació d'*arrays*.

Aquesta classe pertany al *package* `java.util` (com `Random` i `Scanner`). Ofereix tot de mètodes estàtics per fer operacions típiques amb *arrays* (ordenacions, còpies, cerques, omplir rangs de posicions amb certs valors, etc.). Això la fa molt interessant, ja que estalvia molta feina i permet generar codi de manera senzilla. Val la pena conèixer a grans trets quins mètodes conté i saber invocar-los correctament. Ara bé, per usar-los cal tenir present que alguns d'aquests manipulen els paràmetres d'entrada de tipus compost, en aquest cas *arrays*. Per tant, en finalitzar la invocació, l'*array* original haurà canviat.

Tot seguit es llisten algunes de les tasques que es poden assolir usant mètodes estàtics d'aquesta classe. Certament, un cop els domineu, veureu que això fa el codi dels programes molt més senzill que no pas quan heu de codificar aquestes tasques vosaltres mateixos!

1. Ordenació

Poder ordenar les dades dins un *array* pot ser molt còmode per tal de mostrar-les per pantalla i perquè les interpreteu. Per tant, és una operació comuna. **Arrays** ofereix un mètode per dur a terme aquesta operació fàcilment.

`sort(array)` només té un paràmetre d'entrada, que pot ser un *array* de qualsevol tipus de dades primitiu o de cadenes de text. En finalitzar la invocació, els seus elements estaran ordenats (per valor numèric o alfabèticament, segons el tipus de dades). Aquest mètode no avalua res (retorna `void`), ja que treballa directament sobre el paràmetre d'entrada, modificant-ne el contingut.

L'exemple següent mostra com ordenar molt fàcilment un *array* de cadenes de text. Fixeu-vos que aquest mètode no avalua cap resultat, la seva crida simplement transforma el contingut del paràmetre d'entrada.

```
package unitat5.apartat1.exemples;
import java.util.Arrays;
public class OrdenaArrayText {
  public static void main (String[] args) {
    OrdenaArrayText programa = new OrdenaArrayText();
    programa.inici();
  }
  public void inici() {
    String[] array = {"Un", "Dos", "Tres", "Quatre", "Cinc"};
    Arrays.sort(array);
    System.out.println("Els elements ordenats són:");
    for (int i = 0; i < array.length; i++) {
      System.out.println(array[i]);
    }
  }
}
```

2. Cerca

Un dels esquemes més típics d'operació amb *arrays* és, de fet, cercar on hi ha una posició amb un valor concret, per la qual cosa també és util disposar d'un mètode que ho permeti.

`binarySearch(array, clau)` treballa a partir de dos paràmetres d'entrada: un *array* de qualsevol tipus primitiu o `String`, que ha d'haver estat prèviament ordenat, i el valor a cercar, que és la clau de la cerca. Si hi ha un element amb el valor cercat, el mètode avalua el número de la posició on se situa la primera aparició de la clau (un enter), o un valor negatiu si no se n'ha trobat cap. El valor negatiu permet esbrinar la posició d'inserció, dins l'*array*, del valor no trobat, per tal que l'*array* continui ordenat. Per esbrinar la posició d'inserció cal calcular `(-(valor negatiu) - 1)`.

Per poder invocar-lo correctament, les posicions de l'*array* i el valor a cercar han de ser del mateix tipus exactament. També és indispensable que l'*array* tingui els seus elements ordenats, ja que en cas contrari no es pot garantir la correctesa de la cerca. Finalment, sempre heu d'anar amb compte de comprovar si realment s'ha trobat la clau i, per tant, si el mètode ha avaluat un valor positiu o negatiu, ja que en el segon cas es tracta d'una posició invàlida dins l'*array*. En l'exemple següent es pot veure això:

```
1  package unitat5.apartat1.exemples;
2  import java.util.Arrays;
3  public class CercaArray {
4    public static void main(String[] args) {
5      CercaArray programa = new CercaArray();
6      programa.inici();
7    }
8    public void inici() {
9       int[] array = {2, 5, 8, 1, 4, 2, 5, 3, 10};
10     //Primer cal ordenar
11     Arrays.sort(array);
12     int pos = Arrays.binarySearch(array, 5);
13     System.out.println("Hi ha un 5 a la posició " + pos);
14     pos = Arrays.binarySearch(array, 6);
15     System.out.println("Hi ha un 6 a la posició " + pos);
16   }
17 }
```

3. Còpia

Si feu memòria, recordareu que una de les propietats dels tipus compostos en Java és que no es poden fer còpies simplement amb una assignació. El que s'obté són dues variables amb noms diferents que permeten accedir a les mateixes dades. Si es vol una còpia d'un *array*, cal declarar-ne un de nou i copiar-hi tots els valors de l'original, posició per posició. Una altra opció és usar el mètode que ofereix la classe **Arrays**.

`copyOfRange(array, posInici, posFi)` permet crear còpies d'un conjunt de posicions d'un *array* (totes, o només una part). Aquest *array* pot contenir qualsevol tipus primitiu o cadenes de text. Requereix tres paràmetres, l'*array* amb les posicions a copiar, la posició origen i la posició destinació. En invocar-lo, aquest avalua un nou *array* amb (`posFi - posInici`) posicions, en cadascuna

de les quals hi ha copiats els valors entre aquestes posicions (des de `posInici` fins a `posFi - 1`).

Cal anar amb molt de compte que les posicions origen i destinació siguin vàlides (positives i dins del rang de posicions). També s'ha de complir que la posició destinació sigui més gran que l'origen. En cas contrari, es produirà un error en l'execució del programa. Finalment, recordeu que per emmagatzemar el resultat caldrà assignar-lo a una variable de tipus *array*.

La millor manera de veure-ho és amb un exemple, en el qual el resultat de fer l'operació de còpia queda emmagatzemat a la variable `copia`.

```
package unitat5.apartat1.exemples;
import java.util.Arrays;
public class CopiaArray {
  public static void main(String[] args) {
    CopiaArray programa = new CopiaArray();
    programa.inici();
  }
  public void inici() {
    int[] origen = {2, 5, 8, 1, 4, 2, 5, 3, 10, 8};
    //Si es vol copiar de la posició 1 a la 5
    int[] copia = Arrays.copyOfRange(origen, 1, 6);
    System.out.println("Els elements de la còpia són: ");
    for (int i = 0; i < copia.length; i++) {
      System.out.println(copia[i]);
    }
  }
}
```

4. Igualtat

Comparar dos *arrays* implica recorre'ls i anar comprovant posició per posició si totes són iguals. També hi ha un mètode que ja permet fer aquesta comprovació fàcilment.

`equals(array1, array2)` té dos paràmetres d'entrada, que són dos *arrays* de qualsevol tipus de dades primitiu o cadena de text. La seva invocació avalua un booleà que és `true` si els dos són iguals i `false` si no és el cas.

Els valors emmagatzemats als dos *arrays* han de ser del mateix tipus de dades (no es pot comparar un *array* d'enters i un de reals, per exemple). L'exemple següent mostra el funcionament d'aquest mètode.

```
package unitat5.apartat1.exemples;
import java.util.Arrays;
public class ComparaArray {
  public static void main(String[] args) {
    ComparaArray programa = new ComparaArray();
    programa.inici();
  }
  public void inici() {
    double[] origen = {4.5, 1.2, 3.0, 1.4, 3.5};
    //Es fa una còpia exacta
    double[] copia = Arrays.copyOfRange(origen, 0, origen.length);
    boolean iguals = Arrays.equals(origen, copia);
    System.out.println("Són iguals? " + iguals);
    //Es modifica la còpia
    copia[3] = 4.6;
    iguals = Arrays.equals(origen, copia);
    System.out.println("I ara, són iguals? " + iguals);
```

```
  }
}
```

5. Transformació a text

Finalment, una tasca molt típica és mostrar el contingut d'un *array* per pantalla (com es veu ja en els exemples recents). Per això, és útil transformar-lo a una cadena de text.

`toString(array)`, donat un *array* qualsevol com a paràmetre d'entrada, avalua una cadena de text en la qual s'enumeren tots els seus valors. El format de la cadena resultant és una llista de valors separats per comes, entre claus: "[valor1, valor2, ..., valorN]".

Vegeu un exemple tot seguit.

```
package unitat5.apartat1.exemples;
import java.util.Arrays;
public class MostraArray {
  public static void main(String[] args) {
    MostraArray programa = new MostraArray();
    programa.inici();
  }
  public void inici() {
    int[] array = {2, 5, 8, 1, 4, 2, 5, 3, 10, 8};
    String text = Arrays.toString(array);
    System.out.println("L'array conté els valors: " + text);
  }
}
```

Repte 5. Creeu una classe que pertanyi al *package* `unitat5.apartat1.reptes`, el qual faci el següent. Es genera un *array* de 10 posicions i cadascuna d'elles s'inicialitza amb un valor a l'atzar entre 1 i 10, i es mostren per pantalla. Llavors, per cada valor escrit entre les posicions 0 a 4, se cerca si aquest és en alguna de les posicions entre la 5 i la 9. Cada cop que troba un dels valors, ho anuncia per pantalla. Useu tots els mètodes de la classe **Arrays** que sigui possible.

1.5 Documentació de programes en Java

Quan la mida dels programes creix i aquests es fan complicats, compostos per conjunts de classes dins de diversos *packages*, i cadascuna amb els seus mètodes, és important disposar d'alguna manera de poder saber fàcilment per a què serveix cada cosa. Algun mecanisme que faci fàcil avaluar si es poden reutilitzar en altres programes, o reprendre el fil d'un programa ja fet amb anterioritat si més endavant voleu fer esmenes al seu codi font.

Precisament, un aspecte força important, però sovint desatès, durant el desenvolupament d'un programa és documentar correctament el seu funcionament. Cal mantenir un document en el qual s'expliqui com funcionen tots els mòduls i de què es componen (en el cas de Java: *packages*, classes i mètodes), que ha d'estar actualitzat sempre a la darrera versió del programa. Sempre heu de tenir

present que, si bé mentre l'esteu desenvolupant us semblaran evidents tots els aspectes relatius al seu funcionament, passat un temps us n'oblidareu. Llavors, si més endavant hi heu de tornar per fer alguna esmena o alguna modificació, pot resultar més feina tornar a entendre què feia cada bloc de codi que no pas la tasca pròpiament encomanada.

Sempre heu de tenir ben present, com a bon hàbit de programació, que es pot donar el cas que, de fet, no sigueu vosaltres mateixos els qui heu d'entendre els vostres programes, sinó terceres persones. El ventall de possibilitats sota les quals es pot donar aquesta circumstància pot anar des d'un company de treball que ha d'esmenar un bocí de codi que vàreu fer vosaltres fins a un professor que està avaluant el vostre exercici i necessita que sigui fàcil d'entendre i saber què fa cada part. Simplement, poseu-vos en el lloc d'haver d'entendre centenars de línies de codi font que no heu fet vosaltres per fer-vos una idea de per a què serveixen. En programes complexos, no és una tasca gens senzilla.

Encara que no és tan evident com generar codi font, facilitar la tasca de comprensió del codi també és una part integral del procés de desenvolupament d'un programa.

Un exemple molt clar de la utilitat de disposar d'una bona documentació és la de l'API de Java, on s'enumeren totes les classes, biblioteques i mètodes disponibles, explicant amb detall, per a cada cas, per a què serveixen els seus paràmetres d'entrada i el valor de retorn.

1.5.1 Javadoc

Hi ha moltes aproximacions per dur a terme un procés de documentació acurada d'un programa. De fet, en moltes ocasions pot arribar a ser més complicat mantenir la documentació actualitzada que generar el codi font. Això es deu al fet que es tracta de generar un document addicional que cal modificar cada cop que hi ha canvis al codi. Això fa que el procés sigui farragós.

Java intenta pal·liar aquest fet mitjançant un mecanisme que permet generar documentació automàticament a partir del text escrit al codi font, de manera que només cal gestionar un únic fitxer per tenir la informació actualitzada.

El **javadoc** és una eina auxiliar proporcionada pel *kit* de desenvolupament de Java (JDK), que permet generar automàticament documentació relativa a classes Java a partir de comentaris dins del seu codi font.

Els IDE solen disposar d'una opció per integrar dins el seu entorn l'execució de l'eina javadoc.

L'eina *javadoc* processa els fitxers de codi font de les classes d'un programa, de manera que cerca certes paraules clau distintives inserides als comentaris i a partir d'aquestes generar automàticament documentació en format HTML. Això pot estalviar-vos molta feina si mentre aneu fent el codi font del programa sou

acurats i ja aneu creant aquests comentaris, usant les paraules clau corresponents. Codificació i documentació es converteixen en tasques paral·leles.

De fet, la documentació de l'API de Java va estar generada en el seu moment usant aquesta eina. Totes les classes que venen amb les biblioteques de Java es van programar comentant el seu codi font d'acord al format adient. En acabar, només va caldre executar *javadoc* per crear l'HTML amb tota la documentació.

1.5.2 Sintaxi general

Tot el text amb la informació requerida per generar documentació usant *javadoc* s'escriu dins de comentaris al codi font d'acord al format emprat per escriure múltiples línies, que és el que es mostra tot seguit. No es poden usar dobles contrabarres (que serveixen per comentar una línia individual).

```
/** Línia 1 de comentari.
  * Línia 2 de comentari.
  * ...
  * Línia N del comentari.
**/
```

Aquests comentaris es poden inserir en les línies immediatament abans tant de la declaració de la classe (**public class ...**) com dels mètodes. El que s'escrigui en les diferents línies del comentari serà l'explicació detallada d'aquella classe o mètode a la documentació final.

Per exemple, l'esquelet de classe següent aportaria la documentació de la classe pròpia i dels seus dos mètodes.

```
package unitat5.apartat2.exemples;
/** Tot just aquest text serà la documentació de la classe.
  * El que estigui escrit s'inclourà directament a l'HTML resultant.
  * Aquesta és una classe ben documentada.
**/
public class ClasseComentada {
  /** Aquest és el seu primer mètode.
    * Només es mostra l'esquelet, no cal codi de moment.
    * És important veure com funcionen els comentaris per poder usar el javadoc
        .
  **/
  public void unMetode() {
    ...
  }
  /** Un altre mètode, més escàs en documentació.
  **/
  public void unAltreMetode() {
    ...
  }
}
```

Tot el text escrit als comentaris es transforma en text normal en HTML, en un mateix paràgraf. Si es volen afegir efectes especials (negretes, cursiva. etc.) o estructures de text una mica més complexes (com llistes ordenades o enumerades), en el text d'aquests comentaris és possible usar comandes en HTML.

Per exemple, si se sap que l'ordre HTML `<b>...</b>` serveix per fer que un text aparegui en negreta, i que l'ordre `<ol>...</ol>` permet crear una llista enumerada, es podria fer:

```
/** Aquest és un mètode extra.
    * Ara conté una <b>llista d'ítems</b>.
    * <ol>
    *  <li> Ítem 1.
    *  <li> Ítem 2.
    *  <li> Ítem 3.
    *  <li> Ítem 4.
    * </ol>
  **/
  public void unTercerMetode() {
    ...
  }
```

En usar ordres cal anar amb compte, ja que heu de tenir present que el text que heu escrit s'acabarà incorporant a un document HTML general. Per tant, és millor limitar-se a les ordres més simples, i sobretot, mai usar ordres vinculades a encapçalaments (`<h1>...</h1>`, etc.).

Repte 6. Proveu d'executar el *javadoc* des del vostre entorn de desenvolupament per generar la documentació de l'exemple. Tingueu en compte que perquè funcioni no cal que hi hagi instruccions dins dels mètodes, els podeu deixar buits. Només cal que hi hagi comentaris seguint aquest format.

1.5.3 Paraules clau

Hi ha alguns aspectes d'una classe o un mètode que són especialment rellevants, com el seu autor, paràmetres d'entrada o valor de retorn. A part d'una descripció general, val la pena disposar d'una llista clara de quins són i el valor esperat, de manera que destaquin més en el document HTML resultant. Per això, hi ha un seguit de paraules clau que es poden usar un cop finalitzat el text, de manera que s'indica en el *javadoc* aquesta informació addicional.

Totes les paraules clau queden identificades perquè es precedeixen d'un símbol "@", de manera que no puguin ser confoses per text normal, part de la descripció. Després de la paraula clau, es pot escriure una línia de text qualsevol que es considera la seva informació associada.

Tot seguit veureu algunes de les paraules clau més usades.

- **@author**. Aquesta paraula clau només s'usa després del text associat a la descripció de la classe. Indica qui n'ha estat l'autor. Per exemple:

```
package unitat5.apartat2.exemples;
/** Tot just aquest text serà la documentació de la classe.
  * El que estigui escrit s'inclourà directament a l'HTML resultant.
  * Aquesta és una classe ben documentada
  * @author IOC
**/
```

```
7 public class ClasseComentada {
8 ...
9 }
```

- **@param**. Aquesta paraula clau només s'usa després del text associat a la descripció d'un mètode i descriu un dels seus paràmetres. Caldrà afegir-ne tantes paraules clau d'aquest tipus, en diferents línies consecutives, com paràmetres té el mètode. El text ha de començar per l'identificador del paràmetre d'entrada que es vol descriure, i tot seguit s'escriu la seva descripció, on caldria explicar per a què serveix.

```
1 /** Un mètode amb paràmetres.
2    * @param unParametre Aquest paràmetre serveix per fer una cosa.
3    * @param unaltre Parametre Aquest paràmetre diferent serveix per fer una
         altra cosa.
4   **/
5   public void unMetodeParametritzat(int unParametre, int altreParametre) {
6     ...
7   }
```

- **@returns**. Aquesta paraula clau només s'usa després del text associat a la descripció d'un mètode i descriu què avalua el mètode (el valor retornat). Normalment s'escriu després de les paraules clau `@params`, si n'hi ha. Si un mètode no retorna res (`void`), no cal incloure'l. El text associat és directament la descripció.

```
1 /** Un mètode amb paràmetres.
2    * @param unParametre Aquest paràmetre serveix per fer una cosa.
3    * @param unaltre Parametre Aquest paràmetre diferent serveix per fer una
         altra cosa.
4    * @returns Un valor que depèn dels dos paràmetres.
5   **/
6   public int unMetodeParametritzat(int unParametre, int altreParametre) {
7     ...
8   }
```

- **@see**. Aquesta paraula clau es pot usar tant després de la descripció d'un mètode com d'una classe. Serveix per referir-se a la pàgina de documentació d'altres classes dins el mateix programa, en els casos de programes modulars compostos per diferents classes. Bàsicament diu: "Per a més informació, veure també la documentació de...". No serveix per referir-se a qualsevol classe, només funciona amb classes de les quals es disposa el codi font comentat en aquest format. Dins d'un IDE, normalment seran classes dins del mateix projecte.

Com a text associat, cal posar el nom qualificat de la classe a què es refereix.

```
1 /** Tot just aquest text serà la documentació de la classe.
2   * El que estigui escrit s'inclourà directament a l'HTML resultant.
3   * Aquesta és una classe ben documentada.
4   * @author IOC
5   * @see utilitats.arrays.CalculsArrays
6 **/
7 public class ClasseComentada {
8 ...
9 }
```

1.6 Solució dels reptes proposats

Repte 1

```java
public class MitjanaMaxima {
  public static void main(String[] args) {
    MitjanaMaxima programa = new MitjanaMaxima();
    programa.inici();
  }
  public void inici() {
    double[] arrayA = {2, 10.5, 6.3, 4.9, 0, 7.1, 8.3};
    double[] arrayB = {9.4, 6.0, 2.2, 1.0, 8.1};
    //Per cridar els mètodes cal inicialitzar la classe que els conté
    CalculsArrayReals calculador = new CalculsArrayReals();
    //Un cop fet, cal cridar-los usant com a prefix l'identificador
    double mitjanaA = calculador.calcularMitjana(arrayA);
    double mitjanaB = calculador.calcularMitjana(arrayB);
    if (mitjanaA > mitjanaB) {
      System.out.println("arrayA té una mitjana més alta.");
    } else if (mitjanaA < mitjanaB) {
      System.out.println("arrayB té una mitjana més alta.");
    } else {
      System.out.println("Les dues mitjanes són iguals!");
    }
  }
}
```

Repte 2

```java
//------- Fitxer RegistreNotes.java -------
package unitat5.apartat1.reptes;
public class RegistreNotes {
  public static void main(String[] args) {
    RegistreNotes programa = new RegistreNotes();
    programa.inici();
  }
  public void inici() {
    double[] notes = {2.0, 5.5, 7.25, 3.0, 9.5, 8.25, 7.0, 7.5};
    //Per cridar els mètodes cal inicialitzar la classe que els conté
    CalculsArrayReals calculador = new CalculsArrayReals();
    //Un cop fet, cal cridar-los usant com a prefix l'identificador
    double max = calculador.calcularMaxim(notes);
    double min = calculador.calcularMinim(notes);
    double mitjana = calculador.calcularMitjana(notes);
    System.out.println("La nota màxima és " + max + ".");
    System.out.println("La nota mínima és " + min + ".");
    System.out.println("La mitjana de les notes és " + mitjana + ".");
  }
}
```

```java
//------- Fitxer CalculsArrayReals.java -------
package unitat5.apartat1.reptes;
public class CalculsArrayReals {
  public double calcularMaxim(double[] array) {
    double max = array[0];
    for (int i = 1; i < array.length; i++) {
      if (max < array[i]) {
        max = array[i];
      }
    }
    return max;
  }
  public double calcularMinim(double[] array) {
    double min = array[0];
```

```
    for (int i = 1; i < array.length; i++) {
      if (min > array[i]) {
        min = array[i];
      }
    }
    return min;
  }
  public double calcularMitjana(double[] array) {
    double suma = 0;
    for (int i = 0; i < array.length; i++) {
      suma = suma + array[i];
    }
    return suma/array.length;
  }
}
```

Repte 3

```
//——————— Fitxer RegistreNotes.java ———————
package unitat5.apartat1.reptes;
import utilitats.arrays.CalculsArraysReals;
public class RegistreNotes {
  //El codi no canvia
  ...
}
```

```
//——————— Fitxer CalculsArrayReals.java ———————
package utilitats.arrays;
public class CalculsArrayReals {
  //El codi no canvia
  ...
}
```

Repte 4

```
package unitat5.apartat1.reptes;
import java.util.Random;
public class RealsAleatoris {
  public static void main(String[] args) {
    RealsAleatoris programa = new RealsAleatoris();
    programa.inici();
  }
  public void inici() {
    //Inicialització. Per exemple, s'usa el valor 100 com "arrel" (seed).
    Random rnd = new Random(100L);
    //Ús
    double valorA = rnd.nextDouble() * 100;
    double valorB = rnd.nextDouble() * 100;
    System.out.println("S'han generat els valors " + valorA  + " i " + valorB);
  }
}
```

Repte 5

```
package unitat5.apartat1.reptes;
import java.util.Arrays;
import java.util.Random;
public class CercaValors {
  public static void main(String[] args) {
    CercaValors programa = new CercaValors();
    programa.inici();
  }
  public void inici() {
    int[] array = new int[10];
    Random rnd = new Random();
```

```
12      //Omplim l'array
13      for (int i = 0; i < array.length; i++) {
14        array[i] = rnd.nextInt(11);
15      }
16      //Es divideix l'array en dos
17      int[] arrayValors = Arrays.copyOfRange(array, 0, 5);
18      int[] arrayCercar = Arrays.copyOfRange(array, 5, array.length);
19      Arrays.sort(arrayCercar);
20      System.out.print("Valors a cercar: ");
21      System.out.println(Arrays.toString(arrayValors));
22      System.out.print("Array on se cerca: ");
23      System.out.println(Arrays.toString(arrayCercar));
24      //Es fan les cerques
25      for (int i = 0; i < arrayCercar.length; i++) {
26        int valor = arrayValors[i];
27        int pos = Arrays.binarySearch(arrayCercar, valor);
28        if (pos >= 0) {
29          System.out.println("A la posició " + pos + " hi ha el valor " + valor);
30        }
31      }
32    }
33  }
```

Repte 6

Les passes per dur a terme la tasca depenen de l'entorn de desenvolupament usat.

2. Creació d'una aplicació modular. El joc de combats a l'arena

Durant el procés d'aprendre a programar es van presentant diferents conceptes que es consideren importants, des d'aspectes més teòrics o metodològics (què és un tipus de dades, programació estructurada, disseny descendent, etc.), fins aquells molt lligats a la sintaxi del llenguatge (inicialització de variables, invocació de mètodes, sentències, etc.). Quan s'escriuen programes complexos, és imprescindible saber combinar totes aquestes peces per arribar a bon port. Malauradament, sovint us podeu trobar que resulta factible entendre cada concepte de manera individual, i resoldre problemes simples, però no és senzill veure com encaixar totes les peces en problemes més complexos.

En aquesta unitat es planteja resoldre un problema amb un grau de complexitat alt en explicar el funcionament del joc de combats a l'arena. L'objectiu és que us serveixi d'exemple aplicat, i mostrar com totes aquestes petites peces que heu anat aprenent dins el camp de la programació es poden combinar per poder generar un programa de certa envergadura, així com aprendre alguns criteris que es consideren encertats a l'hora de prendre algunes decisions. Per crear aquest programa, es farà especial èmfasi en el concepte de modularitat, de manera que el seu codi font es compongui de diverses classes i se'n pugui veure clarament la justificació.

2.1 El joc de combats a l'arena

El programa que serveix com a fil argumental d'aquest apartat és un joc, en el qual el jugador es va enfrontant amb diversos adversaris en una arena. Cada combat es divideix en rondes, a l'inici de les quals el jugador i el seu adversari trien secretament una estratègia a seguir. Cada ronda pot seguir una estratègia diferent. Segons les estratègies triades per cadascú, el combat s'anirà resolent més favorablement cap a un o cap a l'altre, fins que finalment es consideri que un dels dos ha estat derrotat. Si es derrota l'adversari, s'atorga una puntuació al jugador. Si el jugador és derrotat, acaba la partida. L'objectiu final del jugador és sobreviure deu combats, assolint la màxima puntuació possible en el procés.

Tant per mostrar dades a l'usuari com per introduir les ordres del jugador, s'usa només text.

2.1.1 Descripció detallada del programa

En tractar-se d'un problema més complex, val la pena dedicar un espai a descriure detalladament el problema i veure en què consisteix exactament el joc, en el qual aprofitareu la capacitat d'un ordinador per resoldre ràpidament el tractament d'unes dades per implementar un sistema de combat entre dos lluitadors una mica elaborat, inspirat en els sistemes emprats als CRPG (*Computer Role Playing Game* "joc de rol per ordinador", en anglès), però sense arribar ni de bon tros al nivell dels jocs moderns.

Podeu trobar una mica més d'informació sobre què és un CRPG a l'adreça http://goo.gl/cxZpb.

Atributs dels lluitadors

Per descriure tots els lluitadors, tant el jugador com els seus adversaris, aquests disposen d'un seguit d'atributs que indiquen el seu estat en tot moment. Alguns d'aquests atributs serveixen per establir com progressa el combat i poden veure modificats els seus valors. Tot seguit s'enumeren:

- **Nom**: el nom del lluitador. Per al jugador és "Aventurer", mentre que per als adversaris es referirà a criatures fantàstiques ("Nan", "Ogre", "Hidra", etc.)
- **Nivell:** indicador general de la capacitat de combat del lluitador.
- **Punts:** els punts que ha acumulat el lluitador fins al moment.
- **Punts de Vida (PV):** l'energia del lluitador actual, que pot variar al llarg del combat. Quan arriba a 0 o menys, es considera derrotat.
- **Punts de Vida Màxims:** valor màxim que poden tenir els punts de vida en qualsevol moment.
- **Atac:** la seva capacitat de dur a terme amb èxit estratègies ofensives. S'usa per resoldre el resultat d'una ronda de combat.
- **Defensa:** igual que l'anterior, però per a estratègies defensives.

Combat entre lluitadors


Un combat a mort entre dos lluitadors

El programa es basa en què el jugador va realitzant un combat rere l'altre contra diferents adversaris. Per guanyar, ha de sobreviure a deu combats. Fins que no acaba un combat, i s'ha decidit si el jugador l'ha guanyat o l'ha perdut, no es comença un de nou.

A l'inici de cada combat, es mostra l'estat actual del jugador, el valor actual de tots els seus atributs, i se li pregunta contra quin adversari vol lluitar. El jugador ha d'escriure el nom d'un adversari. Si aquest nom no es troba entre el dels adversaris disponibles en el joc, iniciarà un combat contra un triat a l'atzar entre tots els adversaris disponibles del seu mateix nivell o un de diferència. Això evita que,

per sorpresa, es trobi que ha de lluitar contra un adversari massa poderós per a ell, impossible de guanyar.

Si el nom pertany a algun adversari disponible, llavors s'enfronta contra ell. En aquest cas, no hi ha cap restricció de nivell. El jugador pot triar lluitar contra adversaris molt més o menys poderosos que ell.

Aquest plantejament està disposat de manera que, d'entrada, un nou jugador no sap el nom de cap adversari, ja que no es proporciona cap llista (a menys que hagi fet el programa o vist el codi font, és clar). La intenció és que vagi descobrint nous noms d'adversaris a mesura que va jugant partides, o parlant amb amics que també juguin al joc.

Resolució d'una ronda de combat

Cada combat es divideix en un seguit de rondes, en cadascuna de les quals el jugador ha de triar quina estratègia vol usar. Al principi de cada ronda es mostra l'estat actual tant del jugador com del seu adversari, de manera que sigui possible avaluar quina via d'acció li pot convenir més dur a terme. Llavors, el jugador tria l'estratègia entre quatre possibles: Atacar, Defensar, Engany i Maniobra. Un cop triada, l'adversari en triarà la seva i es decidirà el resultat de la ronda.

El sistema del joc de combats a l'arena està inspirat en el joc de rol de sobretaula "The Mouse Guard RPG" (per David Petersen).

Primer de tot, cal veure per a cada lluitador el grau d'èxit de la seva estratègia. Si ha triat Atacar o Engany, representa que llença tantes monedes com el seu valor d'Atac. Si ha triat Defensar o Maniobra, fa el mateix usant el seu valor de Defensa. El grau d'èxit serà el nombre de cares obtingudes.

En resoldre la ronda, cada lluitador pot rebre un dels efectes següents. La gravetat de cadascun d'ells depèn del grau d'èxit del lluitador mateix o del seu contrincant.

- **Res:** no passa res.
- **Danyat:** el lluitador perd una quantitat de punts de vida igual al grau d'èxit del contrincant.
- **Guarit:** el lluitador recupera tants punts de vida, sense superar mai el valor màxim, com el seu propi grau d'èxit.
- **Penalitzat:** el lluitador veu penalitzat el seu valor d'atac o de defensa (es tria a l'atzar) en tants punts com el grau d'èxit del contrincant. La penalització mai pot fer baixar el valor per sota d'1. Aquest efecte dura fins a la propera ronda múltiple de cinc (5, 10, 15, etc.). Llavors, retorna al seu valor original.

L'efecte que rep cada lluitador depèn de les interaccions entre les estratègies, de manera semblant al joc de pedra, paper, tisores. Depenent de l'estratègia triada i la de l'adversari, el resultat serà diferent. La taula 2.1 mostra el resultat de les interaccions entre estratègies. Per abreujar, "Jug" es refereix al jugador i "Adv" a l'adversari. Un indicador de "x2" vol dir que a l'hora de resoldre aquest efecte, es doblen els èxits assolits pel contrincant.

TAULA 2.1. Resolució d'estratègies

Jug\Adv	Atac	Defensa	Engany	Maniobra
Atac	Jug i Adv: Danyat	Adv: Guarit	Adv: Danyat	Adv: Danyat
Defensa	Jug: Guarit	Jug i Adv: Guarit	Jug: Danyat x2	Jug: Penalitzat
Engany	Jug: Danyat	Adv: Danyat x2	Jug i Adv: Danyat	Jug: Penalitzat
Maniobra	Jug: Danyat	Adv: Penalitzat	Adv: Penalitzat	Jug i Adv: Penalitzat

Exemple de ronda de combat

Per exemple, suposeu que el jugador té ara mateix 10 punts de vida i els seus valors d'atac i defensa són 4 i 3, respectivament. L'adversari té 6 punts de vida i els seus valors d'atac i defensa són 3 i 5 respectivament. Primer de tot, cadascú tria la seva estratègia. El jugador tria Atac mentre que l'adversari tria Maniobra. Això vol dir que, per veure el grau d'èxit, el jugador llençarà tantes monedes com el seu Atac i l'adversari, en haver triat Maniobra, tantes com la seva defensa. El jugador llença 4 monedes i suposeu que treu dues cares. L'adversari en llença 5 i suposeu que n'obté quatre.

Ara cal veure l'efecte de les estratègies. D'acord a la taula, si el jugador tria Atac i l'adversari tria Maniobra, el resultat és que l'adversari rep l'efecte de "Danyat" (*Adv: Danyat*). Se li descompten tants punts de vida com el grau d'èxit del jugador (2). Per tant, ara li queden 6 - 2 = 4 punts de vida i acaba aquesta ronda.

S'inicia una nova ronda on es mostra l'estat dels dos lluitadors i es tria una nova estratègia...

Evidentment, a l'hora de triar l'estratègia, l'ordinador no hauria de fer trampes (ja que coneixerà la del jugador abans de triar-ne la seva). Es pot triar a l'atzar, o seguint alguna tàctica segons el seu estat (defensar més sovint si li queden pocs punts de vida, enganyar si el jugador defensa molt sovint, etc.). Això depèn del grau d'intel·ligència que es vol que tingui l'ordinador.

Resolució de la finalització del combat

El combat finalitza quan, en acabar una ronda, un del lluitadors té 0 o menys punts de vida. Si es tracta del jugador, es considera derrotat. La partida acaba i es mostra la seva puntuació final. Aquesta circumstància inclou també el cas d'empat (ambdós lluitadors han arribat a 0 punts de vida). Si, en cas contrari, és l'adversari el derrotat, al personatge se li atorga certa quantitat de punts, que se sumen als que ja disposa. Els punts atorgats dependran dels punts de l'adversari i, normalment, adversaris més difícils tindran sempre punts.

En atorgar punts al jugador, si aquest arriba o supera un valor associat a una centena (100, 200, 300, etc.), es considera que "puja de nivell" i es fa més poderós. Quan això succeeix, el jugador veu incrementat en un punt el seu nivell, el seu màxim de punts de vida s'incrementa en dos, i el seu atac o defensa, un dels dos triat a l'atzar, s'incrementa en un punt. El jugador també és immediatament guarit, recuperant tots els punts de vida actuals fins a aquest nou màxim.

Un cop atorgats els punts i un possible increment del seu nivell, totes les penalitzacions actuals sobre el jugador desapareixen. Ara bé, a menys que hagi pujat de nivell, aquest no recupera cap punt de vida. Començarà el combat següent amb exactament els mateixos punts amb els quals ha finalitzat aquest.

Si aquest era el desè combat, la partida acaba amb un missatge de felicitació i es mostra la puntuació final. En cas contrari, es torna a iniciar un nou combat.

2.1.2 Identificació de les dades a tractar

De la descripció del problema, les dades principals que cal tractar són les dels dos lluitadors, el jugador i els seus adversaris, que seran les mateixes. Afortunadament, la descripció del problema ofereix una visió clara de quina mena de valors cal manipular (Nom, Nivell, Vida, etc.). En aquest cas, atès que són força valors, tots vinculats entre ells, el més factible és usar un *array*, de manera que es gestioni un per al jugador i un per a l'adversari. D'aquesta manera, amb un parell de variables és senzill disposar de totes les dades vinculades a tots dos. Cada posició de l'*array* pot representar cadascun dels atributs.

Per tal de triar el tipus de dades a desar, pràcticament tots els atributs dels lluitadors són enters, per la qual cosa es poden usar *arrays* d'enters per representar-los. L'únic cas especial que trenca aquesta solució simple és el nom, que seria una cadena de text. Per resoldre això es pot usar l'estratègia següent. Dins l'*array* s'usa un enter per identificar el nom, de manera que cada lluitador sempre tingui un identificador diferent. Llavors, a part, es pot usar una taula de traducció d'identificadors a cadenes de text. L'identificador de tipus enter de cada lluitador diu quin és l'índex dins aquesta taula de traducció on hi ha el seu nom.

Si a simple vista no us queda clar com dur a terme això, no us amoïneu per ara. Quan arribi el moment de mostrar el codi font per fer això, ja es veurà millor.

La taula 2.2 mostra un resum d'una primera proposta d'estructura de l'*array* que representa un lluitador, segons les posicions d'aquest. Ara bé, sempre heu de ser conscients que, a mesura que es va codificant el programa, es pot donar el cas que us adoneu que cal modificar-la.

Taula 2.2. Estructura de les dades associades a un lluitador

0	1	2	3	4	5	6
Identificador	Nivell	Punts	Vida	Vida Màx	Atac	Defensa

2.1.3 Disseny descendent

En aquest cas, l'objectiu principal del problema plantejat és veure com crear aplicacions complexes de manera modular. Per tant, no es durà a terme un disseny descendent complet fins al darrer detall, sinó que aquest servirà per fer un esquema clar de quines són les accions que ha de dur a terme el programa i, en alguns casos, en quin ordre. Per tant, aquest apartat també té un paper de suport a l'hora de presentar-vos el problema perquè l'entengueu.

De la descripció del problema general, se'n podrien extreure els subproblemes enumerats tot seguit. Recordeu, però, que potser aquesta no és l'única solució vàlida, és una proposta d'interpretació possible de l'enunciat. Hi poden haver altres descomposicions vàlides.

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

Aquesta llista ja dóna una bona idea del conjunt de tasques que cal fer. En aquest cas, a mesura que es vagi resolent cada subproblema, si es considera que encara és massa complex o resulta que és un mètode massa llarg, ja es faran noves descomposicions en el mateix moment. Aquesta és una estratègia acceptable per a programes complexos, ja que la descomposició es pot fer molt complicada, en ser difícil veure realment tots els detalls i tenir una idea clara de la mida o complexitat dels mètodes resultants. Però al menys, sempre heu de tenir la disciplina de fer una primera aproximació, per generar el codi font amb una idea clara de per on començar.

Abans de seguir, val la pena fer alguns comentaris. Els subproblemes 4.II i 4.III s'han considerat diferents ja que, si us hi fixeu, hauran de dur a terme tasques força diferents. En el cas del jugador, es pregunta directament a l'usuari, mentre que en el cas de l'adversari l'ordinador és qui l'ha de generar d'alguna manera (per exemple, simplement a l'atzar). En canvi, per al cas dels subproblemes 4.I.a i b, de ben segur que faran el mateix. Només canviaran les dades a tractar. Per tant és un cas clar de parametrització d'un mètode. Per acabar, aquest plantejament també reaprofita subproblemes, ja que el 4.I.a i el 2.I són exactament el mateix.

2.1.4 Mòduls

Si es vol considerar una aproximació modular, un cop es coneixen les tasques que ha de dur a terme el programa en forma de subproblemes, el pas següent seria agrupar-les d'acord al tipus d'accions que porten a terme. Cada conjunt serà un mòdul diferent. En aquest cas, atès que el programa es fa en Java, ja s'usarà directament una organització en classes i *packages*.

Com a punt de partida, caldria escollir un nom de *package* general per a tot el programa. Aquest serà `joc.arena`. La classe principal anirà aquí.

A continuació, cal escollir si es vol usar una jerarquia de *packages* que parteixi de la base per ordenar totes les classes o no. Per a aquest cas, sol ser una bona política dividir les parts vinculades amb la interfície d'usuari de les que estan lligades a la manipulació de les dades del programa. En separar els aspectes relacionats amb la presentació de les dades del seu tractament, els canvis en els mòduls d'un programa(per exemple, passar d'una interfície textual a una gràfica) no afecten el codi dels mòduls de l'altre. Aquesta divisió es pot fer usant dos *packages*: `joc.arena.regles` i `joc.arena.interficie`.

Ara és el moment de dividir les tasques que ha de fer el programa en classes i triar a quin *package* anirà cadascuna. Aquest procés és relativament subjectiu i s'avé a la visió que té el programador sobre com s'ha d'estructurar el seu programa. En aquest sentit, la decisió és tan personal com decidir de quina manera classificar fotos en carpetes dins d'un ordinador. De totes formes, en aquest procés, el que heu de tenir sempre en compte és que l'objectiu final és organitzar el vostre codi de manera que sigui fàcil d'identificar on trobar cada mètode.

> En aplicar modularitat, cal que cada classe encapsuli un conjunt de tasques clarament relacionades, independentment del nombre de mètodes que al final signifiqui que hi ha a cada classe (molts o pocs).

L'objectiu **no** és distribuir uniformement els mètodes per fer classes de mida similar. Si a un programa s'identifiquen 30 mètodes, l'objectiu no és crear 6 classes perquè hi hagi 5 mètodes a cadascuna.

Per a aquest problema es proposa la divisió següent en mòduls. Al *package* `joc.arena.regles` hi haurà les classes:

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

- **Combat:** per a les tasques vinculades a la resolució d'estratègies enfrontades.

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

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

La solució completa de l'exemple emprat en aquest apartat el podeu trobar a la secció "Annexos" del web

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

Un cop es disposa d'aquesta divisió, cada cop que calgui implementar un subproblema en forma de mètode, caldrà fer-ho a la classe que correspongui d'acord a aquesta distribució de tasques.

2.2 La biblioteca "joc.arena.regles"

Abans de poder mostrar dades per pantalla, cal poder disposar d'elles i haver-les manipulat. Per tant, el que té més sentit és començar per aquest *package* i no pas per `joc.arena.interficie`. De fet, de ben segur que des de les classes per mostrar o entrar dades al programa s'invocaran mètodes de tractament de dades. O sigui, mètodes de classes d'aquest *package*.

Per tant, el primer *package* a tractar és aquest.

2.2.1 La classe Monedes

Aquesta classe agrupa els mètodes vinculats als aspectes aleatoris quan es resol un combat. Bàsicament, això es redueix al llançament d'un cert nombre de monedes per comptar quantes cares s'han tret. Això es pot dur a terme usant la classe `Random`, pertanyent al *package* `java.util` de l'API de Java, que permet generar valors a l'atzar. Com que només es vol mirar si es treu cara o creu, es pot usar el mètode `nextBoolean`, de manera que si s'avalua `true`, es considera cara, i en cas contrari, creu. Dins d'aquest programa, seria la classe més senzilla.

Com es pot veure en el codi font, aquesta classe només disposa d'un mètode, ja que, donat el plantejament del problema, només hi ha una acció vinculada al llançament de monedes. Això no és cap problema. Recordeu que, per generar els mòduls d'un programa, el criteri principal i més important és establir parcel·les diferenciades segons les funcionalitats o temàtiques dels mètodes inclosos. No es tracta d'intentar distribuir-los equitativament en parts iguals.

```java
package joc.arena.regles;
import java.util.Random;
public class Monedes {
  /** Resol el llançament d'un grup de monedes.
   *
   * @param numMonedes Nombre de monedes que s'han llençat.
   * @return Nombre de cares obtingudes.
   */
  public int ferTirada(int numMonedes) {
    Random rnd = new Random();
    int numCares = 0;
    for (int i =0; i < numMonedes; i++) {
      boolean tirada = rnd.nextBoolean();
      if (tirada) {
        numCares++;
      }
    }
    return numCares;
  }
}
```

2.2.2 La classe Lluitador

Aquesta classe és la més important, ja que és la que gestiona la manipulació de l'estat dels lluitadors. I per a la descripció del problema, es pot veure que a un lluitador li poden passar moltes coses... Per tant, aquesta classe es tractarà amb molt de detall, ja que serveix per aprendre moltes coses sobre com fer un programa modular correcte.

Quan cal tractar un conjunt de dades de manera complexa, sempre és interessant dedicar un mòdul íntegrament a totes les operacions que es volen dur a terme amb aquestes. Sobretot si algunes de les operacions s'usaran en molts llocs diferents. Un exemple és el cas de disposar d'un repositori de mètodes per manipular *arrays* o cadenes de text que segueixen un format especial. D'aquesta manera, aquestes manipulacions queden encapsulades sota l'identificador d'un mètode i el codi és més fàcil de seguir.

En aquest cas, hi ha un conjunt de dades molt particulars sobre les quals cal fer unes quantes operacions complexes: l'*array* que representa cada *lluitador*. Res impedeix treballar directament amb índexs de posicions per manipular els seus valors, però és millor crear un mètode per a cada operació que es vol dur a terme. És més fàcil entendre una instrucció on posa `danyar(jugador, 3)` que no pas `jugador[3] = jugador[3] - 2;`. A més a més, moltes d'aquestes operacions ja sorgeixen de la descomposició del problema general (per exemple, subproblemes "Combatre> Resoldre resultats d'estrategues: Llençar monedes, penalitzar lluitador, danyar lluitador, guarir lluitador".

L'estratègia final d'aquesta classe és: si un mètode tindria com a paràmetre un lluitador, ja que ha de dur a terme una tasca segons els seus valors, llavors va dins d'aquesta classe.

Mètodes bàsics de manipulació de dades

D'acord amb la descripció del problema, com a mínim cal poder fer les operacions següents sobre un lluitador, ja que modifiquen el seu estat:

- **Danyar**: restar punts de vida.
- **Guarir**: incrementar punts de vida, fins a un màxim.
- **Penalitzar**: restar punts d'Atac o Defensa a l'atzar, però mai de manera que el valor final quedi per sota d'1.
- **Restaurar**: recuperar-se de les penalitzacions (es fa cada ronda múltiple de 5).
- **Renovar**: recuperar tots els punts de vida i eliminar les penalitzacions (en pujar de nivell).
- **Atorgar puntuació**: sumar punts guanyats per un combat.
- **Pujar de nivell**: dur a terme el procés d'increment d'un nivell.

Un cop arribats a aquest punt, es pot detectar que per fer una operació de restaurar, cal saber quin era el valor original de l'Atac o la Defensa. Per tant, el plantejament inicial de com estructurar l'*array* amb les dades dels lluitadors no és suficient. El fet de trobar-vos que heu de reenfocar part del disseny quan heu arribat a la implementació del codi no és un fet estrany. La taula 2.3 mostra el nou format emprat per representar un lluitador.

TAULA 2.3. Nova estructura de les dades associades a un lluitador

0	1	2	3	4	5	6	7	8
Identificador	Nivell	Punts	Vida	Vida Màx	Atac	Atac Màx	Defensa	Defensa Màx

Hi ha un altre aspecte que val la pena considerar en aquest cas, on cada posició de l'*array* que representa un lluitador té un significat molt concret. Es tracta de no usar directament números per accedir a les posicions de l'*array*, sinó fer-ho a partir de constants. Això té dues funcions. D'una banda, millora la llegibilitat del programa. D'altra banda, si en el futur es volen incloure més dades a l'*array* o canviar algun dels camps existents (per exemple, ara el nivell estarà a la posició 3), només caldrà modificar el valor de la constant i el canvi es propagarà a la resta del codi. Com acabeu de veure, haver de canviar el plantejament de les posicions d'un *array* pot passar perfectament. Cal ser previsors.

Finalment, per aprofitar la capacitat que us proporciona un programa per dur a terme càlculs de certa complexitat, l'atorgament dels punts tindrà una particularitat. Sobre els punts que val cada adversari s'aplica un factor de correcció de 0*5 punts per cada nivell de diferència. D'aquesta manera, adversaris proporcionalment més difícils valen més punts, però lluitar repetides vegades contra adversaris de nivell massa baix no reporta cap punt. Per veure-ho, estudieu el mètode `atorgarPunts`.

```java
package joc.arena.regles;
import java.util.Random;
public class Lluitador {
  //Format
  //Nom:Nivell:XP:PV:Max PV:Atac:Max Atac:max Defensa
  public final static int ID = 0;
  public final static int NIVELL = 1;
  public final static int PUNTS = 2;
  public final static int VIDA = 3;
  public final static int VIDA_MAX = 4;
  public final static int ATAC = 5;
  public final static int ATAC_MAX = 6;
  public final static int DEFENSA = 7;
  public final static int DEFENSA_MAX = 8;
  /** Infligeix dany a un lluitador, restant punts de vida, fins a un mínim de
      0.
   *
   * @param lluitador Lluitador que rep el dany
   * @param punts Punts de vida que perd
   */
  public void danyar(int[] lluitador, int punts) {
    if (lluitador[VIDA] > punts) {
      lluitador[VIDA] = lluitador[VIDA] - punts;
    } else {
      lluitador[VIDA] = 0;
    }
  }
  /** Guareix un lluitador, que recupera punts de vida. Mai pot superar
   * el màxim possible.
   *
   * @param lluitador Lluitador a guarir
   * @param punts Punts de vida recuperats
   */
  public void guarir(int[] lluitador, int punts) {
    lluitador[VIDA] = lluitador[VIDA] + punts;
    if (lluitador[VIDA] > lluitador[VIDA_MAX]) {
      lluitador[VIDA] = lluitador[VIDA_MAX];
    }
  }
  /** Aplica una penalització al lluitador. Es fa a l'atzar entre el valor
   * d'atac i el de defensa. Se li resta un punt, fins a un valor mínim d'1.
   *
   * @param lluitador Lluitador a penalitzar
   * @param grau Grau de penalització
   */
  public void penalitzar(int[] lluitador, int grau) {
    //Es tria quina habilitat penalitzar
    Random rnd = new Random();
    int penalitzar = ATAC;
    if (rnd.nextBoolean()) {
      penalitzar = DEFENSA;
    }
    //Es penalitza. Mínim baixa fins a 1
    lluitador[penalitzar] -= grau;
      if (lluitador[penalitzar] < 1) {
        lluitador[penalitzar] = 1;
      }
  }
  /** Renova un lluitador, anul·lant totes les penalitzacions i danys.
   *
   * @param lluitador Lluitador a renovar
   */
  public void renovar(int[] lluitador) {
    restaurar(lluitador);
    lluitador[VIDA] = lluitador[VIDA_MAX];
  }
  /** Restaura els valors d'atac i defensa del lluitador als valors originals.
   *
   * @param lluitador Lluitador a restaurar
```

```
69      */
70    public void restaurar(int[] lluitador) {
71      lluitador[ATAC] = lluitador[ATAC_MAX];
72      lluitador[DEFENSA] = lluitador[DEFENSA_MAX];
73    }
74    /** Resol l'atorgament de punts a l'aventurer en derrotar un adversari. La
75     * quantitat de punts depèn de la diferencia de nivells entre els dos. Si es
76     * guanyen prou punts, s'avisa si cal pujar de nivell.
77     *
78     * @param aventurer Aventurer
79     * @param adversari Adversari derrotat
80     * @returns Si s'ha pujat de nivell (cada 100 punts)
81     */
82    public boolean atorgarPunts(int[] aventurer, int[] adversari) {
83      //Es calcula el multiplicador
84      float multiplicador = 0;
85      int numMultiplicadors = adversari[NIVELL]  - aventurer[NIVELL] + 2;
86      for (int i = 0; i < numMultiplicadors; i++) {
87        multiplicador += 0.5;
88      }
89      //Punts finals a atorgar
90      int puntsAdversari = llegirPunts(adversari);
91      int puntsAtorgats = Math.round(puntsAdversari*multiplicador);
92      //Puja de nivell?
93      aventurer[PUNTS] += puntsAtorgats;
94      int nouNivell = 1 + aventurer[PUNTS]/100;
95      if (nouNivell > aventurer[NIVELL]) {
96        return true;
97      }
98      return false;
99    }
100   /** Resol un increment d'un nivell, augmentant a l'atzar atac o defensa i dos
101    * punts de vida màxims. A més a més, el lluitador es guareix totalment.
102    *
103    * @param lluitador Lluitador que puja de nivell.
104    */
105   public void pujarNivell(int[] lluitador) {
106     lluitador[NIVELL]++;
107     Random rnd = new Random();
108     if (rnd.nextBoolean()) {
109       //S'incrementa atac
110       lluitador[ATAC_MAX]++;
111     } else {
112       //S'incrementa defensa
113       lluitador[DEFENSA_MAX]++;
114     }
115     lluitador[VIDA_MAX] += 2;
116     //Es deixa nou de trinca
117     renovar(lluitador);
118   }
119 }
```

Mètodes vinculats a l'estat del lluitador

Atès que aquesta classe inclou tots els mètodes que depenen de l'estat d'un lluitador per fer la seva feina, també cal incloure, no només els que modifiquen el seu estat, sinó també els mètodes el resultat dels quals depèn d'aquest estat. Aquests inclouen els que fan les operacions següents:

- Calcular el grau d'èxit d'Atac, ja que depèn del valor d'Atac del lluitador.
- El mateix per a la defensa.
- Triar una estratègia a l'atzar, ja que es pot usar l'estat del lluitador per

prendre certes decisions. Aquí es farà que si els punts de vida de l'adversari són molt baixos, és més probable que decideixi defensar.

```
//Mètodes d'accions vinculades a l'estat del lluitador
  /** Resol una tirada d'atac d'un lluitador. Es llencen tantes monedes com
   * el seu valor d'atac.
   *
   * @param lluitador Lluitador que fa la tirada
   * @return El nombre de cares obtingudes
   */
  public int tirarAtac(int[] lluitador) {
    Monedes monedes = new Monedes();
    return monedes.ferTirada(lluitador[ATAC]);
  }
  /** Resol una tirada de defensa d'un lluitador. Es llencen tantes monedes com
   * el seu valor de defensa.
   *
   * @param lluitador Lluitador que fa la tirada
   * @return El nombre de cares obtingudes
   */
  public int tirarDefensa(int[] lluitador) {
    Monedes monedes = new Monedes();
    return monedes.ferTirada(lluitador[DEFENSA]);
  }
  /** Donat un lluitador, tria a l'atzar quina estratègia usar en una
   * ronda de combat.
   *
   * @param lluitador Lluitador que tria l'acció
   * @return Acció triada
   */
  public int triarEstrategiaAtzar(int[] lluitador) {
    Random rnd = new Random();
    int limitDefensa = 3;
    //Si li queda poca vida, defensa el 50% dels cops
    if (lluitador[VIDA] < 2) {
      limitDefensa = 1;
    }
    int accio = rnd.nextInt(10);
    if ((accio >= 0)&&(accio < limitDefensa)) {
      return Combat.ATAC;
    } else if ((limitDefensa >= 3)&&(accio < 6)) {
      return Combat.DEFENSA;
    } else if ((accio >= 6)&&(accio < 8)) {
      return Combat.ENGANY;
    } else {
      return Combat.MANIOBRA;
    }
  }
```

Mètodes per facilitar la lectura de les dades

Finalment, quan es treballa amb conjunts de dades amb una funció molt especial, com és aquest cas, pot valer la pena també incloure mètodes que serveixin com a dreceres per fer lectures de les dades que contenen. Aquests no fan res d'especial que no es podria fer accedint directament a l'*array* per índex, però poden fer el codi de la resta de classes més aclaridor, i a més a més, en cas de documentar la classe usant el *javadoc*, serveixen com una llista de les dades a les quals es pot accedir individualment.

```
//Mètodes per facilitar la lectura de dades
  /** Diu l'identificador d'un lluitador. Usar un mètode facilita la lectura
   * del codi, més que accedir a posicions d'un array.
   *
   * @param lluitador Lluitador de qui es vol llegir l'identificador
   * @return Identificador del lluitador
   */
  public int llegirId(int[] lluitador) {
    return lluitador[ID];
  }
  /** Diu quin és el nivell de lluitador. Usar un mètode facilita la lectura
   * del codi, més que accedir a posicions d'un array.
   *
   * @param lluitador Lluitador de qui es vol llegir el nivell
   * @return
   */
  public int llegirNivell(int[] lluitador) {
    return lluitador[NIVELL];
  }
  /** Diu quins punts val el lluitador. Usar un mètode facilita la lectura
   * del codi, més que accedir a posicions d'un array.
   *
   * @param lluitador Lluitador de qui es volen llegir els punts
   * @return
   */
  public int llegirPunts(int[] lluitador) {
    return lluitador[PUNTS];
  }
  /** Diu quina vida té el lluitador. Usar un mètode facilita la lectura
   * del codi, més que accedir a posicions d'un array.
   *
   * @param lluitador Lluitador de qui es vol llegir la vida
   * @return Vida
   */
  public int llegirVida(int[] lluitador) {
    return lluitador[VIDA];
  }
  /** Diu quina vida màxima té el lluitador. Usar un mètode facilita la lectura
   * del codi, més que accedir a posicions d'un array.
   *
   * @param lluitador Lluitador de qui es vol llegir la vida màxima
   * @return Vida
   */
  public int llegirVidaMax(int[] lluitador) {
    return lluitador[VIDA_MAX];
  }
  /** Diu quin atac té el lluitador. Usar un mètode facilita la lectura
   * del codi, més que accedir a posicions d'un array.
   *
   * @param lluitador Lluitador de qui es vol llegir l'atac
   * @return Grau d'atac
   */
  public int llegirAtac(int[] lluitador) {
    return lluitador[ATAC];
  }
  /** Diu quina defensa té el lluitador. Usar un mètode facilita la lectura
   * del codi, més que accedir a posicions d'un array.
   *
   * @param lluitador Lluitador de qui es vol llegir la defensa
   * @return Grau de defensa
   */
  public int llegirDefensa(int[] lluitador) {
    return lluitador[DEFENSA];
  }
  /** Diu si un lluitador és mort o no. O sigui, si els seus punts de vida
   * son 0 ara mateix. Usar un mètode facilita la lectura del codi, més que
   * accedir a posicions d'un array.
   *
   * @param lluitador Lluitador a comprovar
```

```
70     * @return Si es considera mort (true) o no (false)
71     */
72    public boolean esMort(int[] lluitador) {
73      return (lluitador[VIDA] == 0);
74    }
```

2.2.3 La classe Bestiari

Aquesta classe s'encarrega de tots els aspectes vinculats a la generació dels lluitadors, tant de les dades inicials del jugador com la dels adversaris triats pel jugador (veure si hi ha el que s'ha demanat, i si no és el cas, triar-lo a l'atzar). En aquest cas, s'ha escollit que els adversaris no es generin a l'atzar, sinó que les seves dades ja existeixin dins el codi font, de manera que, donat un adversari amb un nom concret, sempre sigui igual. Per emmagatzemar-los, s'usa un *array* bidimensional: un *array* on en cada posició hi ha l'*array* que descriu els valors d'un adversari. Atès que és on estan definits els noms dels lluitadors, també es gestiona la traducció dels identificadors dels lluitadors al seu nom.

En aquesta classe també es mostra una possibilitat quan s'usen mètodes d'una altra classe sovint dins del codi font. En lloc d'inicialitzar constantment la classe per poder invocar els seus mètodes, hi ha l'opció de declarar la variable on es duu a terme la inicialització com a global. Com a variable global té un àmbit igual a tota la classe, pot ser usada des de qualsevol lloc per invocar mètodes de l'altra classe. Això és tot just el que passa amb la variable `lluitador`. Fixeu-vos-hi.

```
1  package joc.arena.regles;
2  import java.util.Random;
3  public class Bestiari {
4    //Taula de traducció d'identificadors a noms
5    private String[] noms = {"Aventurer",
6                             "Nan", "Kobold",
7                             "Orc", "Profund",
8                             "Bruixot maligne", "Ogre",
9                             "Guerrer del caos", "Troll",
10                            "Elemental terrestre", "Hidra"};
11   //Jugador: ID = 0
12   private int[] jugador = {0,  1,  0, 10, 10, 3, 3, 3, 3};
13   //Adversaris possibles al joc
14   private int[][] adversaris = {
15                                {1,  1, 25,  8,  8, 3, 3, 3, 3},
16                                {2,  1, 30, 10, 10, 4, 4, 2, 2},
17                                {3,  2, 35, 12, 12, 4, 4, 3, 3},
18                                {4,  2, 40, 14, 14, 3, 3, 4, 4},
19                                {5,  3, 45, 15, 15, 3, 3, 5, 5},
20                                {6,  3, 50, 16, 16, 5, 5, 2, 2},
21                                {7,  4, 55, 15, 15, 4, 4, 4, 4},
22                                {8,  4, 60, 18, 18, 3, 3, 5, 5},
23                                {9,  5, 70, 22, 22, 4, 4, 6, 6},
24                                {10, 5, 80, 30, 30, 8, 8, 2, 2}
25                                };
26   //Inicialització usant una variable global
27   private Lluitador lluitador = new Lluitador();
28   /** Genera un nou jugador
29    *
30    * @return Un array amb les dades d'un jugador inicial
31    */
32   public int[] generarJugador() {
```

```
33      lluitador.renovar(jugador);
34      return jugador;
35    }
36    /** Donat un nom, genera l'adversari corresponent. Si aquest nom no existeix,
37     * es genera a l'atzar.
38     *
39     * @param nomAdv Nom de l'adversari a obtenir
40     * @return El lluitador amb aquest nom, o null si no existeix
41     */
42    public int[] cercarAdversari(String nomAdv) {
43      for (int i = 0; i < adversaris.length; i++) {
44       int id = lluitador.llegirId(adversaris[i]);
45       String nom = traduirIDANom(id);
46       if (nom.equalsIgnoreCase(nomAdv)) {
47         lluitador.renovar(adversaris[i]);
48         return adversaris[i];
49       }
50      }
51      return null;
52    }
53    /**Donat un nivell, genera l'adversari corresponent a l'atzar. Es tracta
54     * d'un adversari que sigui d'aquest nivell almenys.
55     *
56     * @param nivell Nivell proper al de l'adversari a obtenir
57     * @return Un adversari
58     */
59    public int[] triarAdversariAtzar(int nivell) {
60      Random rnd = new Random();
61      int[] adversari = null;
62      boolean cercar = true;
63      while (cercar) {
64        int i = rnd.nextInt(adversaris.length);
65        adversari = adversaris[i];
66        int nivellAdv = lluitador.llegirNivell(adversari);
67        int dif = nivell - nivellAdv;
68        if ((dif >= -1)&&(dif <= 1)) {
69          cercar = false;
70        }
71      }
72      //Es deixa a l'adversari nou de trinca, llest per lluitar
73      lluitador.renovar(adversari);
74      return adversari;
75    }
76    /** Transforma un identificador de lluitador al seu nom.
77     *
78     * @param id Identificador
79     * @return La cadena de text amb el nom.
80     */
81    public String traduirIDANom(int id) {
82      if ((id >= 0) && (id < noms.length)) {
83        return noms[id];
84      }
85      return "DESCONEGUT";
86    }
87  }
```

2.2.4 La classe Combat

Aquesta classe s'encarrega dels aspectes vinculats a la resolució d'una ronda de combat, donat l'estat actual dels dos lluitadors (punts de vida, Atac i Defensa) i l'estratègia triada per aquesta ronda. Bàsicament, sobre ella recau el pes del subproblema "Combatre>Resoldre resultats d'estràtegia" resultant del disseny

descendent. Com que aquest encara es divideix en altres subproblemes, per fer les seves tasques li caldrà fer invocacions sobre altres mètodes (concretament, de la classe `Lluitador`).

Per indicar cada estratègia s'usa un valor enter, de manera que fer comparacions és molt més senzill que no pas amb cadenes de text. Per facilitar la lectura del codi, el valor associat a cada estratègia s'assigna a una constant. Fixeu-vos com mitjançant invocacions a mètodes de la classe `Lluitador` es modifica l'estat de cada lluitador segons la resolució de la ronda. Si s'han triat els noms dels mètodes de manera adient, aquest sistema deixa molt clar què succeeix en cada cas sense haver d'anar a inspeccionar el codi font de `Lluitador`.

A més a més, amb ja certa previsió amb vista a mostrar dades per pantalla, s'inclou un mètode que serveix per transformar una acció donada a la seva representació en format text. Es tracta del mètode `estrategiaAText`.

El codi d'aquesta classe seria el següent:

```
1  package joc.arena.regles;
2  public class Combat {
3    //Constants que indiquen possibles accions de combat
4    public static final int ATAC = 0;
5    public static final int DEFENSA = 1;
6    public static final int ENGANY = 2;
7    public static final int MANIOBRA = 3;
8    /** Donat el codi d'una estratègia, el converteix a un text.
9     *
10    * @param acció Codi de l'estratègia
11    * @return Text associat
12    */
13   public String estrategiaAText(int accio) {
14     switch(accio) {
15       case ATAC: return "ATAC";
16       case DEFENSA: return "DEFENSA";
17       case ENGANY: return "ENGANY";
18       case MANIOBRA: return "MANIOBRA";
19     }
20     return "DESCONEGUDA";
21   }
22   /** Obté en grau d'èxit corresponent segons l'acció triada pel lluitador.
23    *
24    * @param ll Lluitador que fa l'acció.
25    * @param accio Acció triada
26    * @return Grau d'èxit de l'acció
27    */
28   public int calcularGrauExit(int[] ll, int accio) {
29     Lluitador lluitador = new Lluitador();
30     switch(accio) {
31       case ATAC:
32       case ENGANY:
33         return lluitador.tirarAtac(ll);
34       default:
35         return lluitador.tirarDefensa(ll);
36     }
37   }
38   /** Resol una ronda d'accions entre dos lluitadors, d'acord amb les estratè
         gies
39    * individuals de cadascú.
40    *
41    * @param jug Estat del Jugador
42    * @param accioJug Estratègia triada pel Jugador
43    * @param adv Estat de l'Adversari
44    * @param accioAdv Estratègia triada per l'Adversari.
45    */
```

```
46    public void resoldreEstrategies(int[] jug, int accioJug, int[] adv, int
          accioAdv) {
47      int exitJug = calcularGrauExit(jug, accioJug);
48      int exitAdv = calcularGrauExit(adv, accioAdv);
49      Lluitador lluitador = new Lluitador();
50      if ((accioJug == ATAC)&&(accioAdv == ATAC)) {
51        //Jug i Adv: Danyat
52        lluitador.danyar(jug, exitAdv);
53        lluitador.danyar(adv, exitJug);
54      } else if ((accioJug == ATAC)&&(accioAdv == DEFENSA)) {
55        //Adv: Guarit
56        lluitador.guarir(adv, exitAdv);
57      } else if ((accioJug == ATAC)&&(accioAdv == ENGANY)) {
58        //Adv: Danyat
59        lluitador.danyar(adv, exitJug);
60      } else if ((accioJug == ATAC)&&(accioAdv == MANIOBRA)) {
61        //Adv: Danyat
62        lluitador.danyar(adv, exitJug);
63      } else if ((accioJug == DEFENSA)&&(accioAdv == ATAC)) {
64        //Jug: Guarit
65        lluitador.guarir(jug, exitJug);
66      } else if ((accioJug == DEFENSA)&&(accioAdv == DEFENSA)) {
67        //Jug i Adv: Guarit
68        lluitador.guarir(adv, exitAdv);
69        lluitador.guarir(jug, exitJug);
70      } else if ((accioJug == DEFENSA)&&(accioAdv == ENGANY)) {
71        //Jug: Danyat x2
72        lluitador.danyar(jug, exitAdv*2);
73      } else if ((accioJug == DEFENSA)&&(accioAdv == MANIOBRA)) {
74        //Jug: Penalitzat
75        lluitador.penalitzar(jug, exitAdv);
76      } else if ((accioJug == ENGANY)&&(accioAdv == ATAC)) {
77        //Jug: Danyat
78        lluitador.danyar(jug,exitAdv);
79      } else if ((accioJug == ENGANY)&&(accioAdv == DEFENSA)) {
80        //Adv: Danyat x2
81        lluitador.danyar(adv, exitJug*2);
82      } else if ((accioJug == ENGANY)&&(accioAdv == ENGANY)) {
83        //Jug i Adv: Danyat
84        lluitador.danyar(jug, exitAdv);
85        lluitador.danyar(adv, exitJug);
86      } else if ((accioJug == ENGANY)&&(accioAdv == MANIOBRA)) {
87        //Jug: Penalitzat
88        lluitador.penalitzar(jug, exitAdv);
89      } else if ((accioJug == MANIOBRA)&&(accioAdv == ATAC)) {
90        //Jug: Danyat
91        lluitador.danyar(jug, exitAdv);
92      } else if ((accioJug == MANIOBRA)&&(accioAdv == DEFENSA)) {
93        //Adv: Penalitzat
94        lluitador.penalitzar(adv, exitJug);
95      } else if ((accioJug == MANIOBRA)&&(accioAdv == ENGANY)) {
96        //Adv: Penalitzat
97        lluitador.penalitzar(adv, exitJug);
98      } else if ((accioJug == MANIOBRA)&&(accioAdv == MANIOBRA)) {
99        //Jug i Adv: Penalitzat
100       lluitador.penalitzar(adv, exitJug);
101       lluitador.penalitzar(jug, exitAdv);
102     } else {
103       //No s'hauria de donar aquest cas...
104     }
105   }
106 }
```

2.3 La biblioteca "joc.arena.interficie"

Un cop es disposa del codi font de totes les classes que gestionen les dades a manipular, ja és possible generar les que les obtenen o les mostren a l'usuari.

2.3.1 La classe EntradaTeclat

Donada la descripció del problema general, només hi ha dos casos on l'usuari ha d'entrar dades usant el teclat. Per indicar contra quin adversari vol lluitar en iniciar-se un combat i per dir l'estratègia a seguir en una ronda de combat. Per tant, només cal incloure dos mètodes:

- `triarAdversari`, associat al subproblema "Triar l'adversari".
- `preguntarEstrategia`, associat al subproblema "Combatre>Triar estratègia del jugador".

Per al cas de l'estratègia, es donarà a triar amb un menú amb quatre opcions, cadascuna associada a una lletra: `[A]tacar`, `[D]efensar`, `[E]ngany` i `[M]aniobra`.

Un fet interessant d'aquesta classe és que, atès que a la classe `Combat` les quatre estratègies possibles estan indicades mitjançant constants, cal traduir la lletra que ha escrit l'usuari al valor de la constant associada. Per accedir a una constant declarada a una altra classe, cal usar la mateixa sintaxi que per invocar un mètode estàtic, però usant l'identificador de la contant: `NomClasse.NOM_CONSTANT`. Això es pot veure al codi del mètode `preguntarEstrategia`.

```java
package joc.arena.interficie;
import java.util.Scanner;
import joc.arena.regles.Bestiari;
import joc.arena.regles.Combat;
public class EntradaTeclat {
  /** Tria l'adversari del jugador segons la seva resposta.
   *
   * @return Cadena de text amb la resposta
   */
  public int[] triarAdversari(int nivell) {
    System.out.print("Contra quin adversari vols lluitar en aquest combat? ");
    Scanner lector = new Scanner(System.in);
    String resposta = lector.nextLine();
    Bestiari bestiari = new Bestiari();
    int[] adversari = bestiari.cercarAdversari(resposta);
    if (adversari == null) {
      System.out.println("Aquest enemic no existeix. Es tria a l'aztar.");
      adversari = bestiari.triarAdversariAtzar(nivell);
    }
    return adversari;
  }
  /** Pregunta a l'usuari quina estratègia vol usar, d'entre
   * les quatre possibles.
   *
   * @return Accio a dur a terme, d'acord a les constants de la classe Combat.
   */
```

```
27    public int preguntarEstrategia() {
28      Scanner lector = new Scanner(System.in);
29      System.out.println("Quina estrategia vols seguir aquesta ronda?");
30      System.out.println("[A]tacar\t[D]efensar\t[E]ngany\t[M]aniobra");
31      System.out.println("—————————");
32      boolean preguntar = true;
33      int accio = -1;
34      while (preguntar) {
35        System.out.print("Accio: ");
36        String resposta = lector.nextLine();
37        if ("A".equalsIgnoreCase(resposta)) {
38          accio =  Combat.ATAC;
39          preguntar = false;
40        } else if ("D".equalsIgnoreCase(resposta)) {
41          accio = Combat.DEFENSA;
42          preguntar = false;
43        } else if ("E".equalsIgnoreCase(resposta)) {
44          accio = Combat.ENGANY;
45          preguntar = false;
46        } else if ("M".equalsIgnoreCase(resposta)) {
47          accio = Combat.MANIOBRA;
48          preguntar = false;
49        } else {
50          System.out.print("Acció incorrecta...");
51        }
52      }
53      return accio;
54    }
55  }
```

2.3.2 La classe SortidaPantalla

En aquesta classe s'agruparien els mètodes en els quals cal més d'una única instrucció per mostrar informació per pantalla. Concretament es tractaria dels mètodes associats als subproblemes "Anunciar inici de combats>Mostrar estat del jugador" i "Combatre>Mostrar l'estat dels lluitadors: Mostrat estat del jugador, Mostrar estat de l'adversari" del resultat d'aplicar disseny descendent. No es tracta d'una classe molt complexa.

```
1   package joc.arena.interficie;
2   import joc.arena.regles.Bestiari;
3   import joc.arena.regles.Lluitador;
4   public class SortidaPantalla {
5     /** Mostra per pantalla el missatge d'inici del Joc
6      *
7      */
8     public void mostarBenvinguda() {
9       System.out.println("Benvingut al Joc de l'Arena");
10      System.out.println("==========================");
11      System.out.println("Escull amb astúcia la teva estratègia per sobreviure
            ...");
12    }
13    /** Mostra l'estat d'un lluitador per pantalla.
14     *
15     * @param ll LLuitador a visualitzar
16     */
17    public void mostrarLluitador(int[] ll) {
18      Lluitador lluitador = new Lluitador();
19      Bestiari bestiari = new Bestiari();
20      int id = lluitador.llegirId(ll);
21      String nom = bestiari.traduirIDANom(id);
22      System.out.print(nom);
```

```
    System.out.print("\tNivell: " + lluitador.llegirNivell(ll));
    System.out.print(" (punts: " + lluitador.llegirPunts(ll) + ")");
    System.out.print("\tVIDA: " + lluitador.llegirVida(ll));
    System.out.print(" (" + lluitador.llegirVidaMax(ll) + ")");
    System.out.print("\tATAC: " + lluitador.llegirAtac(ll));
    System.out.println("\tDEFENSA: " + lluitador.llegirDefensa(ll));
  }
  /** Mostra l'estat actual del jugador contra el seu adversari.
   *
   * @param jugador Jugador
   * @param adversari Adversari
   */
  public void mostrarVersus(int[] jugador, int[] adversari) {
    System.out.print("JUGADOR: ");
    mostrarLluitador(jugador);
    System.out.println("VS");
    System.out.print("ADVERSARI: ");
    mostrarLluitador(adversari);
  }
}
```

2.4 La classe principal

Quan es genera un programa modular estructurat d'acord a una certa jerarquia de *packages*, la classe principal se sol ubicar en el *package* arrel, tota sola. Recordeu que aquesta és la que s'encarrega de resoldre el problema general a partir de la invocació de mètodes de tota la resta de classes dels altres *packages*.

Per a aquest programa, el seu codi seria el següent. Observeu com, novament, la seva estructura és molt propera al primer nivell de descomposició del disseny descendent.

```
package joc.arena;
import joc.arena.regles.Bestiari;
import joc.arena.regles.Combat;
import joc.arena.regles.Lluitador;
import joc.arena.interficie.EntradaTeclat;
import joc.arena.interficie.SortidaPantalla;
public class JocArena {
  public static final int MAX_COMBAT = 10;
  private EntradaTeclat entrada = new EntradaTeclat();
  private SortidaPantalla sortida = new SortidaPantalla();
  private Lluitador lluitador = new Lluitador();
  private Combat combat = new Combat();
  private Bestiari bestiari = new Bestiari();
  public static void main(String[] args) {
    JocArena programa = new JocArena();
    programa.inici();
  }
  public void inici() {
    sortida.mostarBenvinguda();
    int[] jugador = bestiari.generarJugador();
    int numCombat = 0;
    boolean jugar = true;
    while (jugar) {
      numCombat++;
      //Abans de cada combat es restaura al jugador
      lluitador.restaurar(jugador);
      //Inici d'un combat
      System.out.println("*** COMBAT " + numCombat);
      System.out.print("Estat actual del jugador: ");
```

```
30        sortida.mostrarLluitador(jugador);
31        System.out.println("*************************");
32        //S'obté l'adversari
33        int[] adversari = entrada.triarAdversari(lluitador.llegirNivell(jugador))
              ;
34        //Combat
35        combatre(jugador, adversari);
36        //Fi
37        jugar = fiCombat(jugador, adversari);
38        if (numCombat == MAX_COMBAT) {
39          System.out.println("Has sobreviscut a tots els combats. Enhorabona!!");
40        }
41      }
42      System.out.print("Estat final del jugador: ");
43      sortida.mostrarLluitador(jugador);
44    }
45    /** Resol totes les rondes d'un combat.
46     *
47     * @param jugador Estat del jugador
48     * @param adversari Estat de l'adversari
49     */
50    public void combatre(int[] jugador, int[] adversari) {
51      boolean combatre = true;
52      int numRonda = 0;
53      while (combatre) {
54        numRonda++;
55        if (numRonda%5 == 0) {
56          //A les rondes múltiples de cinc es restauren l'atac i la defensa
57          lluitador.restaurar(jugador);
58          lluitador.restaurar(adversari);
59        }
60        System.out.println("——— RONDA " + numRonda);
61        sortida.mostrarVersus(jugador, adversari);
62        System.out.println("————————————————————————");
63        int accioJug = entrada.preguntarEstrategia();
64        int accioAdv = lluitador.triarEstrategiaAtzar(adversari);
65        System.out.print("Has triat " + combat.estrategiaAText(accioJug));
66        System.out.println(" i el teu enemic " + combat.estrategiaAText(accioAdv)
              );
67        combat.resoldreEstrategies(jugador, accioJug, adversari, accioAdv);
68        if (lluitador.esMort(jugador)||lluitador.esMort(adversari)) {
69          combatre = false;
70        }
71      }
72    }
73    /** Resol la finalització del combat.
74     *
75     * @param jugador Estat del jugador
76     * @param adversari Estat de l'adversari
77     * @returns Si el jugador ha de seguir jugant (true) o no (false)
78     */
79    public boolean fiCombat(int[] jugador, int[] adversari) {
80      if (lluitador.esMort(jugador)) {
81        //Has perdut (Nota: tambe inclou el cas que tots dos moren alhora)
82        System.out.println("Has estat derrotat... :-(");
83        return false;
84      }
85      System.out.println("Has guanyat el combat :-)");
86      boolean pujarNivell = lluitador.atorgarPunts(jugador, adversari);
87      if (pujarNivell) {
88        System.out.println("Has pujat de nivell!!");
89        lluitador.pujarNivell(jugador);
90      }
91      return true;
92    }
93  }
```

2.5 Simplificació d'algorismes complexos usant recursivitat

A l'hora de crear programes complexos, un dels aspectes que diferencia el bon programador de l'aficionat és la seva capacitat de fer algorismes eficients. O sigui, que siguin capaços de resoldre el problema plantejat en el mínim de passes. En el cas d'un programa, això significa la necessitat d'executar el mínim nombre d'instruccions possible. Certament, si el resultat ha de ser exactament el mateix, sempre serà millor fer una tasca en 10 passes que no pas en 20, intentant evitar passes que en realitat són innecessàries. Per tant, l'etapa de disseny d'un algorisme és força important i cal pensar bé una estratègia eficient. Ara bé, normalment, els algorismes més eficients també són més difícils de pensar i codificar, ja que no sempre són evidents.

Un exemple molt senzill d'això és la resolució del problema següent. Suposeu que una amiga apunta un número entre el 0 i el 99 en un full de paper i vosaltres l'heu d'endevinar. Cada cop que contesteu, us dirà si el valor que heu dit és més gran o més petit que el que heu d'endevinar. Quina estratègia seguiríeu per assolir-ho? Cal pensar un algorisme a seguir per resoldre aquest problema.

Una aproximació molt ingènua podria ser anar dient tots els valors un per un, començant pel 0. està clar que quan arribeu al 99 l'haureu endevinat. En el millor cas, si havia escrit el 0, encertareu a la primera, mentre que en el pitjor cas, si havia escrit el 99, necessitareu 100 intents. Si estava pel mig, potser amb 40-70 n'hi ha prou. Aquest seria un algorisme que fa el fet i és molt senzill, però no gaire eficient. Anar provant valors a l'atzar en lloc de fer això tampoc millora gran cosa el procés, i ve a ser el mateix.

De ben segur, si mai heu jugat a aquest joc, el que heu fet és ser una mica més astuts i començar per algun valor del mig. En aquest cas, per exemple, podria ser el 50. Llavors, en cas de fallar, un cop sabeu si el valor secret és més gran o més petit que la vostra resposta, en l'intent següent provar un valor més alt o més baix, i anar fent això repetides vegades.

Generalment, la millor estratègia per endevinar un número secret entre 0 i N seria primer provar N/2. Si no s'ha encertat, llavors si el número secret és més alt s'intenta endevinar entre (N/2 + 1) i N. Si era més baix, s'intenta endevinar el valor entre 0 i N-1. Per a cada cas, es torna a provar el valor que hi ha al bell mig del nou interval. I així successivament, fent cada cop més petit l'interval de cerca, fins a endevinar-lo. En el cas de 100 valors, això garanteix que, en el pitjor dels casos, en 7 intents segur que s'endevina. Això és una millora molt gran respecte al primer algorisme, on calien 100 intents, i per tant, aquest seria un algorisme més eficient. Concretament, sempre s'endevinarà en log2(N) intents com a màxim.

Si us hi fixeu, l'exemple que tot just s'acaba d'explicar, en realitat, no és més que un esquema de cerca dins una seqüència de valors, com pot ser dins d'un *array*, partint de la condició que tots els elements estiguin ordenats de més petit a més gran. De fet, fins ara, per fer una cerca d'un valor dins d'un *array* s'ha usat el sistema "ingenu", mirant una per una totes les posicions. Però si els elements estan

El mètode `binarySearch` de la classe `Arrays` fa una cerca dicotòmica.

ordenats prèviament, es podria usar el sistema "astut" per dissenyar un algorisme molt més eficient, i fins a cert punt, més "intel·ligent".

L'algorisme basat en aquesta estratègia es coneix com **cerca binària** o **dicotòmica**.

Per tant, per què no aplicar aquest coneixement per millorar el mètode `cercarAdversari`, a la classe `Bestiari`?

2.5.1 Aplicació de la recursivitat

Malauradament, sovint us trobareu que explicar de paraula la idea general d'una estratègia pot ser senzill, però traduir-la a instruccions de Java ja no ho és tant. Atès que cal anar repetint unes passes en successives iteracions, està més o menys clar que el problema plantejat per fer cerques eficients es basa en una estructura de repetició. Però no es recorren tots els elements i l'índex no s'incrementa un a un, sinó que es va canviant a valors molt diferents per cada iteració. No és un cas evident. Precisament, aquest exemple no s'ha triat a l'atzar, ja que és un cas en què us pot anar bé aplicar un nou concepte que permet facilitar la definició d'algorismes complexos on hi ha repeticions.

> La **recursivitat** és una forma de descriure un procés per resoldre un problema de manera que, al llarg d'aquesta descripció, s'usa el procés mateix que s'està descrivint, però aplicat a un cas més simple.

De fet, potser sense adonar-vos-en, ja s'ha usat recursivitat per descriure com resoldre un problema. Per veure què vol dir exactament la definició formal tot just descrita, es repetirà el text en qüestió, però remarcant l'aspecte recursiu de la descripció:

Una definició recursiva: les inicials del sistema operatiu GNU volen dir "GNU is Not Unix". Font: The GNU Art Gallery

"Generalment, la millor estratègia per **endevinar** un número secret entre 0 i N seria primer provar N/2. Si no s'ha encertat, llavors si el número secret és més alt s'intenta **endevinar** entre (N/2 + 1) i N. Si era més baix, s'intenta **endevinar** el valor entre 0 i N-1. Per a cada cas, es torna a provar el valor que hi ha al bell mig del nou interval. I així successivament, fins a endevinar-lo."

O sigui, el procés d'endevinar un número es basa en el procés d'intentar endevinar un número! Això sembla fer trampes, ja és com usar la mateixa paraula que es vol definir a la seva pròpia definició. Però fixeu-vos en un detall molt important. Els nous usos del procés d'"endevinar" són casos més simples, ja que primer s'endevina entre N valors possibles, després entre N/2 valors, després entre N/4, etc. Aquest fet no és casual i d'ell depèn poder definir un procés recursiu de manera correcta.

2.5.2 Implementació de la recursivitat

La implementació de la recursivitat dins del codi font d'un programa es fa a nivell de mètode.

Un **mètode recursiu** és aquell que, dins del seu bloc d'instruccions, té alguna invocació a ell mateix.

El bloc de codi d'un mètode recursiu sempre es basa en una estructura de selecció múltiple, on cada branca és d'algun dels dos casos possibles descrits tot seguit.

D'una banda, en el **cas base**, que conté un bloc instruccions dins de les quals no hi ha cap crida al mètode mateix. S'executa quan es considera que, a partir dels paràmetres d'entrada, el problema ja és prou simple com per ser resolt directament. En el cas de la cerca, seria quan la posició intermèdia és exactament el valor que s'està cercant, o bé quan ja es pot decidir que l'element a cercar no existeix.

D'altra banda, hi ha el **cas recursiu**, que conté un bloc d'instruccions dins de les quals hi ha una crida al mètode mateix, atès que es considera que encara no es pot resoldre el problema fàcilment. Ara bé, valors usats com a paràmetres d'aquesta nova crida han de ser diferents als originals. Concretament, han de ser uns valors que tendeixin a apropar-se al cas base. En el cas de la cerca, es correspon a la cerca sobre la meitat dels valors originals, ja sigui cap a la meitat inferior o superior. Aquest és un cas en què l'interval de posicions on es farà la nova cerca es va apropant al cas base, ja que tard o d'hora, crida rere crida, l'espai de cerca s'anirà reduint fins que, o bé es troba l'element, o queda clar que no hi és.

Dins de l'estructura de selecció sempre hi ha d'haver almenys un cas base i un de recursiu. Normalment, els algorismes recursius més senzills en tenen un de cada. És imprescindible que els casos recursius sempre garanteixin que successives crides van aproximant els valors dels paràmetres d'entrada a algun cas base, ja que, en cas contrari, el programa mai acaba i es produeix el mateix efecte que un bucle infinit.

En Java, en cas d'un bucle infinit en fer crides recursives es produeix un error de *Stack Overflow* ("desbordament de pila", en anglès).

Càlcul recursiu de l'operació factorial

Com exemple del funcionament d'un mètode recursiu, es començarà amb un cas senzill. Es tracta del càlcul de l'anomenada operació factorial d'un valor enter positiu. Aquesta és unària i s'expressa amb l'operador exclamació (per exemple, 4!, 20!, 3!). El resultat d'aquesta operació és la multiplicació de tots els valors des de l'1 fins a l'indicat (7! = 1*2*3*4*5*6*7). Normalment, la definició matemàtica d'aquesta operació es fa de manera recursiva:

- 0! = 1 (cas base)
- n! = n*(n - 1)! (cas recursiu)

Així, doncs, fixeu-vos que el cas recursiu realitza un càlcul que depèn d'usar la pròpia definició de l'operació, però quan ho fa és amb un nou valor inferior a l'original, de manera que es garanteix que, en algun moment, es farà una crida recursiva que desembocarà en el cas base. Quan això passi, la cadena de crides recursives acaba. Una manera de veure això és desenvolupant pas per pas aquesta definició:

1. 4! = 4*(4 - 1)! = 4*(3)!
2. 4*3! = 4*(3*(3 - 1))! = 4*3*(2)!
3. 4*3*2! = 4*3*(2*(2 - 1))! = 4*3*2*(1)!
4. 4*3*2*1! = 4*3*2*(1*(1 - 1))! = 4*3*2*1*(0)!
5. 4*3*2*1*0! = 4*3*2*1*(1) = 24

La seva implementació en Java seria la següent. Ara bé, en aquest codi s'han afegit algunes sentències per escriure informació per pantalla, de manera que es vegi amb més detall com funciona un mètode recursiu. Veureu que, inicialment, es porten a terme un seguit d'invocacions del cas recursiu, un rere l'altre, fins que s'arriba a una crida que executa el cas base. És a partir de llavors quan, a mesura que es van executant les sentències `return` del cas recursiu, realment es va acumulant el càlcul. Una altra manera de veure-ho és depurant el programa.

```
1  package unitat5.apartat2.exemples;
2
3  public class Factorial {
4
5    public static void main(String[] args) {
6      Factorial programa = new Factorial();
7      programa.inici();
8    }
9
10   public void inici() {
11     System.out.println(factorial(4));
12   }
13
14   /** Mètode recursiu que calcula l'operació factorial
15    *
16    * @param n Operador
17    * @return Resultat de n!
18    */
19   public int factorial (int n) {
20     if (n == 0) {
21       //Cas base: Se sap el resultat directament
22       System.out.println("Cas base: S'avalua a 0");
23       return 1;
24     } else {
25       //Cas recursiu: Per calcular-lo cal invocar el propi metode
26       //El valor del nou parametre d'entrada ha de variar de manera que
27       //es vagi aproximant al cas base
28       System.out.println("Cas recursiu " + (n - 1) + ": S'invoca el factorial")
             ;
29       int res = n*factorial(n - 1);
30       System.out.println("Cas recursiu " + (n - 1) + ": Resultat = " + res);
31       return res;
32     }
33   }
34
35 }
```

Càlcul recursiu de la cerca dicotòmica

Tot seguit es mostra el codi de l'algorisme recursiu de cerca dicotòmica sobre un *array*. Observeu atentament els comentaris, els quals identifiquen els casos base i recursius. En aquest cas, hi ha més d'un cas base i recursiu. Si voleu veure amb més detall com funciona, el podeu depurar per veure com van evolucionant els valors dels paràmetres d'entrada en successives invocacions als casos recursius.

```java
package unitat5.apartat2.exemples;

public class CercaDicotomica {

  public static void main(String[] args) {
    CercaDicotomica programa = new CercaDicotomica();
    programa.inici();
  }

  public void inici() {
    int[] array = {2, 4, 6, 8, 10, 12, 14, 16, 18, 20};
    int cercaDivuit = cercaDicotomica(array, 0, array.length-1, 18);
    int cercaCinc = cercaDicotomica(array, 0, array.length-1, 5);
    System.out.println("Cerca del 18: " + cercaDivuit);
    System.out.println("Cerca del 5: " + cercaCinc);
  }

  /** Cerca dicotòmica recursiva sobre un array d'enters.
   *
   * @param array On es fa la cerca
   * @param inici Posició inicial de la cerca
   * @param fi Posició final
   * @param valor Valor a cercar
   * @return Índex on és el valor, o -1 si no existeix
   */
  public int cercaDicotomica(int[] array, int inici, int fi, int valor) {
    if (inici > fi) {
      //Cas base: No s'ha trobat el valor
      return -1;
    }
    //Es calcula la posició central entre els dos índexs de cerca
    int pos = inici + (fi - inici) / 2;
    if (array[pos] > valor) {
      //Cas recursiu: Si el valor es menor que la posició que s'ha mirat
      //llavors cal seguir cercant per la part "dreta" de l'array
      return cercaDicotomica(array, inici, pos - 1, valor);
    } else if (array[pos] < valor) {
      //Cas recursiu: Si el valor és més gran que la posició que s'ha mirat
      //llavors cal seguir cercant per la part "esquerra" de l'array
      return cercaDicotomica(array, pos + 1, fi, valor);
    } else {
      //cas base: És igual, per tant, s'ha trobat
      return pos;
    }
  }
}
```

Pràcticament qualsevol problema que es pot resoldre amb un algorisme recursiu també es pot resoldre amb sentències d'estructures de repetició. Però molt sovint la seva implementació serà molt menys evident i les interaccions entre instruccions força més complexes que l'opció recursiva (un cop s'entén aquest concepte, és clar).

2.5.3 Recursivitat al joc de combats a l'arena

Un cop coneixeu la base genèrica per generar un algorisme recursiu, és el moment d'aplicar-lo per incloure una cerca dicotòmica dins del joc proposat. En aquest cas, es tractaria de millorar el mètode `cercarAdversari`, de manera que, en lloc de fer la cerca posició per posició, es faci de manera més eficient. La cerca es fa sobre els noms dels adversaris, que han d'estar prèviament ordenats alfabèticament. Per tant, per poder dur a terme aquesta millora, cal garantir que la llista de noms està ordenada i que els valors dels identificadors als *arrays* amb els atributs de tots els adversaris continuen encaixant amb la nova llista de noms. Fer això manualment pot ser una mica pesat, sobretot si en el futur s'afegeixen nous adversaris. L'ordenació es pot dur a terme amb codi tot just al principi del programa.

Per les característiques modulars del programa, el nou mètode de cerca i el d'ordenació anirien dins de la classe `Bestiari`. Tot seguit es mostra el nou codi que cal afegir. En el cas de `cercarAdversari`, aquest reemplaçaria el ja existent. La resta són nous.

Estudieu atentament el codi següent, que presenta la solució al que es planteja. Presteu especial atenció a com es fa la comparació entre cadenes de text mitjançant el mètode `compareTo` de la classe `String`. Aquest indica si una cadena de text és igual, o té un ordre alfabètic inferior o superior a una altra. També tingueu en compte que les majúscules i minúscules es consideren lletres diferents i afecten l'ordre de les paraules. Per tant, per evitar que això succeeixi, sempre es passa tot a minúscula abans de comparar, amb el mètode `toLowercase`.

```
1  //Mètodes per recursivitat a la classe Bestiari
2    /** Donat l'array bidimensional amb els adversaris, els ordena pel nom
3     * associat a cada identificador. Recordar que un array bidimensional es
4     * considera un "array d'arrays"
5     */
6    public void ordenarAdversaris() {
7      for (int i = 0; i < adversaris.length; i++) {
8        for (int j = i + 1; j < adversaris.length; j++) {
9          int idI = lluitador.llegirId(adversaris[i]);
10         String nomI = traduirIDANom(idI);
11         nomI = nomI.toLowerCase();
12         int idJ = lluitador.llegirId(adversaris[j]);
13         String nomJ = traduirIDANom(idJ);
14         nomJ = nomJ.toLowerCase();
15         if (nomI.compareTo(nomJ) > 0) {
16           //I > J. Cal intercanviar
17           int midaArray = adversaris[i].length;
18           int[] temp = new int[midaArray];
19           copiarArray(adversaris[i], temp);
20           copiarArray(adversaris[j], adversaris[i]);
21           copiarArray(temp, adversaris[j]);
22         }
23       }
24     }
25   }
26   /** Mètode auxiliar per copiar els valors d'un array a un altre.
27    *
28    * @param origen Array origen
29    * @param desti Array destí
30    */
```

```
31    public void copiarArray(int[] origen, int[] desti) {
32      for (int i=0; i < desti.length; i++) {
33        desti[i] = origen[i];
34      }
35    }
36    /** Inici de la cerca recursiva. Adapta la crida normal d'acord a la mida
37     * de la llista d'adversaris.
38     *
39     * @param nomAdv Nom a cercar
40     * @return Adversari (null si no es troba)
41     */
42    public int[] cercarAdversari(String nomAdv) {
43      return cercarAdversariRecursivament(nomAdv.toLowerCase(), 0, adversaris.
          length - 1);
44    }
45    /** Cerca recursiva real
46     *
47     * @param nomAdv Nom a cercar
48     * @param inici Posició inicial de la cerca
49     * @param fi Posició final de la cerca
50     * @return Adversari trobat (o null si no es troba)
51     */
52    public int[] cercarAdversariRecursivament(String nomAdv, int inici, int fi) {
53      int diferencia =  fi - inici;
54      if (diferencia < 0) {
55        //No s'ha trobat
56        return null;
57      } else {
58        int posicio = inici + diferencia/2;
59        int id = lluitador.llegirId(adversaris[posicio]);
60        String nom = traduirIDANom(id);
61        nom = nom.toLowerCase();
62
63        int compara = nom.compareTo(nomAdv);
64        if (compara == 0) {
65          //Són iguals. Trobat!
66          return adversaris[posicio];
67        } else if (compara < 0) {
68          //El nom de la llista és mes petit. Cal cercar pels posteriors
69          return cercarAdversariRecursivament(nomAdv, posicio + 1,fi);
70        } else {
71          //El nom de la llista és mes gran. Cal cercar pels anteriors
72          return cercarAdversariRecursivament(nomAdv, inici, posicio - 1);
73        }
74      }
75    }
```

Per garantir que els noms estan ordenats alfabèticament, caldrà cridar el mètode d'ordenació tan bon punt s'iniciï el programa, al mètode `inici` a la classe `JocArena`.

```
1   public void inici() {
2       //PART RECURSIVA
3       bestiari.ordenarAdversaris();
4       //De fet, fora de l'ordenació, tot el codi es manté idèntic
5       sortida.mostarBenvinguda();
6       int[] jugador = bestiari.generarJugador();
7       //etc.
8     }
```

Amb aquestes esmenes, s'ha aplicat recursivitat per millorar l'eficiència del programa. És clar que, per trobar adversaris dins una llista de 10 elements no serà pas evident a simple vista, però fer programes eficients és important per tal de permetre la gestió de quantitats molt més grans d'informació.