Introducció

Un cop coneixeu els tres tipus d'estructura de control (seqüencial, selecció i repetició), ja disposeu de totes les eines bàsiques per poder implementar algorismes dins els vostres programes d'ordinador, usant la tècnica de la programació estructurada. La seva aplicació correcta us permet generar un codi polit i ordenat, relativament fàcil d'entendre per una tercera persona. Ara bé, el punt realment més important a l'hora de dur a terme un programa d'ordinador és la definició del seu algorisme. Si ja d'entrada no dissenyeu un algoritme correcte, el programa mai funcionarà correctament, independentment del fet que s'usin correctament els principis de la programació modular o que el codi font no tingui errors de sintaxi.

Per a programes senzills, el disseny d'algorismes es pot realitzar de manera *ad hoc*, pensant les diferents passes ordenades que cal seguir en llenguatge natural, que després poden ser traduïdes a instruccions del codi font. Ara bé, per a programes complexos, en els quals hi ha moltes passes interrelacionades, pot ser fàcil perdre's, o crear una descripció enrevessada o difícil de seguir. Hi ha tants detalls que us resultarà impossible tenir-los tots presents alhora al cap. El resultat és que resulta molt més fàcil equivocar-se o, fins i tot, un cop se sap que hi ha un error en el disseny, intentar esbrinar on es troba aquest. Per tant, és útil disposar d'una metodologia de treball que doni un cop de mà en aquest aspecte. En aquesta unitat es presentarà una metodologia força popular i útil a l'hora de dissenyar algorismes complexos: el **disseny descendent**. En poques paraules, aquest es basa en l'estratègia, aplicable a qualsevol camp i no només a la programació, de dividir problemes complexos en un conjunt de problemes més simples i fàcils d'atacar i entendre.

A l'apartat "Descomposició de problemes" s'exposen els principis del disseny descendent aplicat dins el context de la creació d'algorismes. Per poder dur-lo a terme, es presenta un nou tipus de bloc d'instruccions que podeu usar dins els vostres programes, els **mètodes**. Si bé aquests ja havien estat introduïts amb anterioritat, només s'havien vist des del punt de vista de la seva invocació per manipular cadenes de text. Aquí veureu amb una mica més de detall com funcionen realment, la seva utilitat i com en podeu definir de nous, inventats per vosaltres, en els vostres programes. De moment, però, aquest apartat només ofereix una visió general sobre com la seva existència està vinculada a l'aplicació del disseny descendent d'algorismes, sense veure tots els seus detalls.

Els detalls més complexos de la declaració i ús de mètodes es detalla a l'apartat "Parametrització de mètodes". Aquí es presenta la seva utilitat com a mecanisme per manipular i produir dades, de manera que el seu ús resulti molt més versàtil i puguin ser invocats amb resultats similars als descrits per operar amb cadenes de text. Això permet crear un recull d'eines reusables dins els vostres programes per transformar informació, de manera que s'eviti haver d'escriure codi repetit.

Heu de tenir en compte que la declaració de mètodes i la seva invocació és una eina usada amb assiduïtat als programes de qualsevol llenguatge de programació, per la qual cosa comprendre-la i saber quan usar-la és de capital importància per a un programador. En el món real, és complicat dur a terme programes complexos que no incloguin mètodes. Per tant, convé que a partir d'ara sempre que feu un programa intenteu aplicar els mecanismes explicats en aquest mòdul per tal de dividir el codi en diferents mètodes, i si pot ser, reaprofitar-los en diferents llocs dins el programa.

Resultats d'aprenentatge

En finalitzar aquesta unitat, l'alumne/a:

1. Analitza els conceptes relacionats amb la programació modular
2. Analitza els avantatges i la necessitat de la programació modular
3. Aplica el concepte d'anàlisi descendent en l'elaboració de programes
4. Modularitza correctament els programes realitzats.
5. Realitza correctament les crides a funcions i la seva parametrització.
6. Té en compte l'àmbit de les variables en les crides a les funcions

1. Descomposició de problemes

Igual que amb la immensa majoria de tasques amb cert grau de complexitat dins el món real, la creació d'un programa requereix un pas previ on cal reflexionar sobre què és exactament el que voleu fer i com assolireu la vostra fita. És molt poc recomanable afrontar aquesta tasca ja seient directament davant de l'ordinador, obrint l'entorn de treball i començant a escriure línies de codi. Aquesta opció només és realment factible quan disposeu d'una certa experiència programant i trobeu que el problema que heu de resoldre, o bé ja l'heu tractat amb anterioritat, o s'assembla molt a un altre que ja heu resolt. Però quan us enfronteu amb un problema nou és imprescindible una etapa en la qual estudiar el problema, les dades que voleu tractar exactament i les tasques que ha de dur a terme l'ordinador per fer-ho (o sigui, l'algorisme del programa).

Malauradament, la capacitat dels humans per copsar problemes complexos és limitada, ja que, en general, només som capaços de mantenir una visió simultània d'uns pocs elements. A aquest fet cal afegir que la presa d'una decisió sobre quina passa cal dur a terme dins la descripció d'un procés sempre té implicacions sobre futures passes. Per tant, quan el procés que cal realitzar és llarg o es basa en la manipulació de molts elements diferents, és molt fàcil, no ja simplement equivocar-se, sinó tan sols saber per on començar.

Un cop arribats a aquest punt, es fa evident que resultaria útil disposar d'alguna estratègia que permeti fer front a la resolució de problemes amb diferents graus de complexitat. Una de les més populars en tots els camps, i que de ben segur useu sovint en el vostre dia a dia, potser sense adonar-vos, és considerar que un problema complex en realitat no és més que l'agregació d'un conjunt de problemes més simples, cadascun d'ells més fàcils de resoldre. Per tant, si sou capaços d'entendre i resoldre tot aquest conjunt de problemes simples, també podreu ser capaços de resoldre el problema complex.

Com diu la frase que s'atribueix a Filip II de Macedònia: "Divide et impera" (divideix i venceràs). Font: Tilemahos Efthimiadis

En conseqüència, i partint d'aquesta premissa, el primer pas per poder dur a terme una tasca complexa serà trobar com descompondre-la en d'altres més simples, que llavors s'aniran resolent un per un.

1.1 Disseny descendent

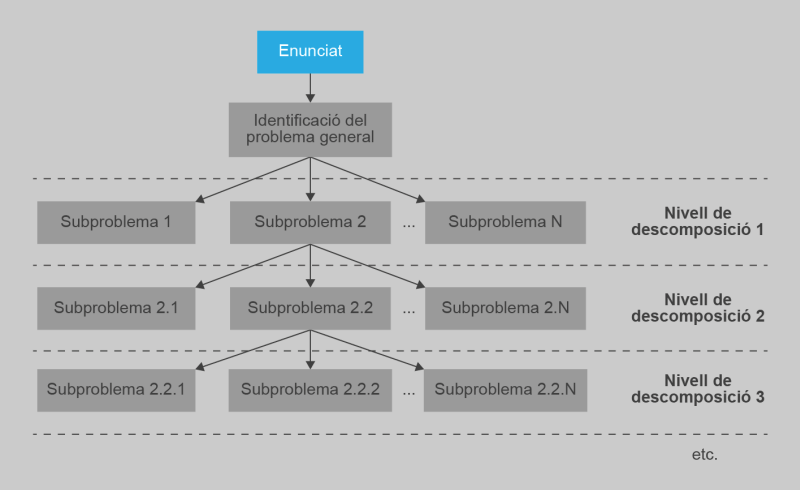
Hi ha dues estratègies bàsiques per resoldre la descomposició d'un problema: el disseny descendent i l'ascendent. Aquest apartat se centra en la primera, en ser la més utilitzada per norma general, i la més fàcil d'aplicar a l'hora de crear programes del nivell que abasten aquests materials.

En el context de la programació, el disseny ascendent normalment s'aplica dins el camp de l'orientació a objectes.

El **disseny descendent** (*top-down*, en anglès) és la tècnica que es basa en partir d'un problema general i dividir-lo en problemes més simples, denominats subproblemes. D'entre tots aquests, els considerats encara massa complexos es tornen a dividir en nous subproblemes. S'anomena descendent perquè partint del problema gran es passa a problemes més petits als quals donarà solució individualment.

L'esquema d'aplicació d'aquesta estratègia es mostra a la figura 1.1, en la qual es veu la raó del nom *descendent*, i s'aprecia com, partint de la definició del problema general, extreta de tasca final que voleu assolir, es crea una estructura jeràrquica de subproblemes en diferents nivells. El nombre de nivells a què cal arribar dependrà de la complexitat del problema general. Per a problemes no massa complexos, hi haurà prou amb un o dos nivells, però per resoldre problemes molt complexos pot caldre un gran nombre de successives descomposicions. També val la pena remarcar que, tot i que és recomanable que la complexitat dels subproblemes d'un mateix nivell sigui aproximadament equivalent, n'hi pot haver que quedin resolts completament en menys nivells que en d'altres.

Figura 1.1. Esquema d'aplicació de disseny descendent, d'acord als nivells de descomposició del problema



Un punt important a tenir en compte en aplicar aquesta descomposició és que cadascun dels subproblemes no es genera arbitràriament, sinó que es planteja com un objectiu parcial, amb entitat pròpia, per resoldre el seu problema de nivell superior. Un cop assolits tots aquests objectius parcials, es considera resolt el total.

Els objectius finals d'aplicar aquesta estratègia són:

- Establir una relació senzilla entre problemes plantejats i el conjunt de tasques a fer per resoldre'ls.
- Establir més fàcilment les passes per resoldre un problema.
- Fer més fàcil d'entendre aquestes passes.
- Limitar els efectes de la interdependència que un conjunt de passes té sobre un altre conjunt.

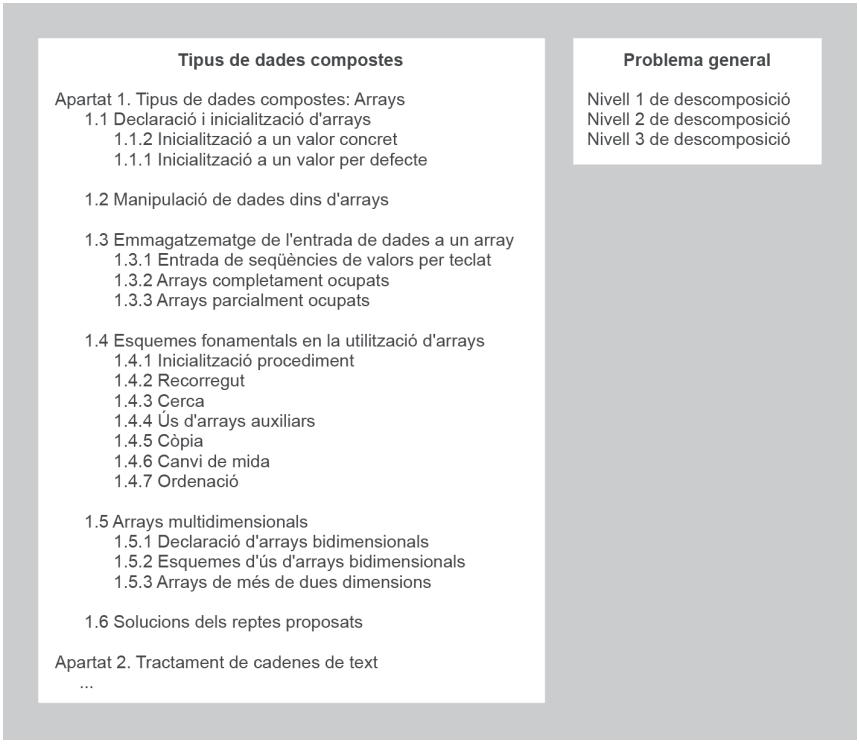
1.1.1 Exemples d'aplicació de disseny descendent

Com sempre, la millor manera de veure l'aplicació de disseny descendent és mitjançant exemples. Per començar, aquests se centraran en la resolució de problemes d'àmbit general, que permetin veure amb claredat el procés sigui quin sigui el context, abans d'entrar en el cas concret de la generació d'un programa.

Una unitat formativa

Un cas força directe d'aplicació de disseny descendent és l'escriptura d'un document de certa complexitat, com per exemple, un llibre o una unitat formativa de l'IOC. Es tracta d'un cas directe ja que l'estructura d'un document d'aquest tipus és evidentment jeràrquica, basada en capítols, seccions, subseccions, etc. Així, doncs, tot just davant vostre ara mateix teniu el resultat directe d'aplicar disseny descendent sobre un problema.

Figura 1.2. Descomposició d'una Unitat Formativa seguint el disseny descendent



En un cas com aquest, quan es planteja que cal fer una unitat formativa, tot i que es parteix ja d'unes directrius o idea general d'allò que es vol explicar (per exemple, uns objectius d'aprenentatge), estareu d'acord que no seria gaire assenyat seure ja immediatament davant d'un processador de text i posar-se a escriure, atacant frontalment el problema. Aquesta aproximació normalment porta a no saber ben bé per on començar, o simplement posar fi a un text incomprensible, amb fins i tot explicacions repetides. És més eficient començar amb una etapa de disseny en la qual s'estableixi un primer nivell d'objectius parcials en la redacció: una divisió inicial en apartats. Aquest primer nivell de descomposició de ben segur

que encara serà massa genèric, però ja ha dividit el problema inicial en d'altres més petits.

Un cop arribats a aquest punt, per cada apartat, es van fent successives divisions en seccions, subseccions, etc. partint de conceptes més generals que es volen explicar cap a conceptes més específics. Un cop es considera que s'ha arribat a un concepte prou específic com per poder ser explicat de manera autocontinguda i de manera relativament fàcil d'entendre pel lector, ja no cal descompondre més. Evidentment, un cop tractats tots els apartats, és possible trobar-se que alguns apartats o seccions estan dividits en més subapartats que d'altres. Això no és problema. Ara bé, el que sí que ha de ser cert sempre és que cada subapartat es correspongui a una temàtica concreta, amb una entitat i sentit propi (d'acord al seu títol), i mai es tracta d'un "calaix de sastre" on s'expliquen moltes coses diferents poc relacionades entre elles. Per exemple, la figura 1.2 mostra la descomposició d'una unitat d'acord a aquests criteris.

Com es pot veure, en aquest cas, el resultat d'aplicar la descomposició us dóna com a resultat l'índex de la unitat. Estareu d'acord que és molt més senzill editar un document partint d'un índex preestablert, amb noms de seccions autoexplicatius sobre allò que han de tractar, que no pas actuant de manera improvisada. Addicionalment, aquest procés de descomposició assoleix una altra fita molt important que va més enllà de facilitar l'etapa de redacció del text. El document resultant també resulta molt més fàcil de seguir i entendre per part dels futurs lectors.

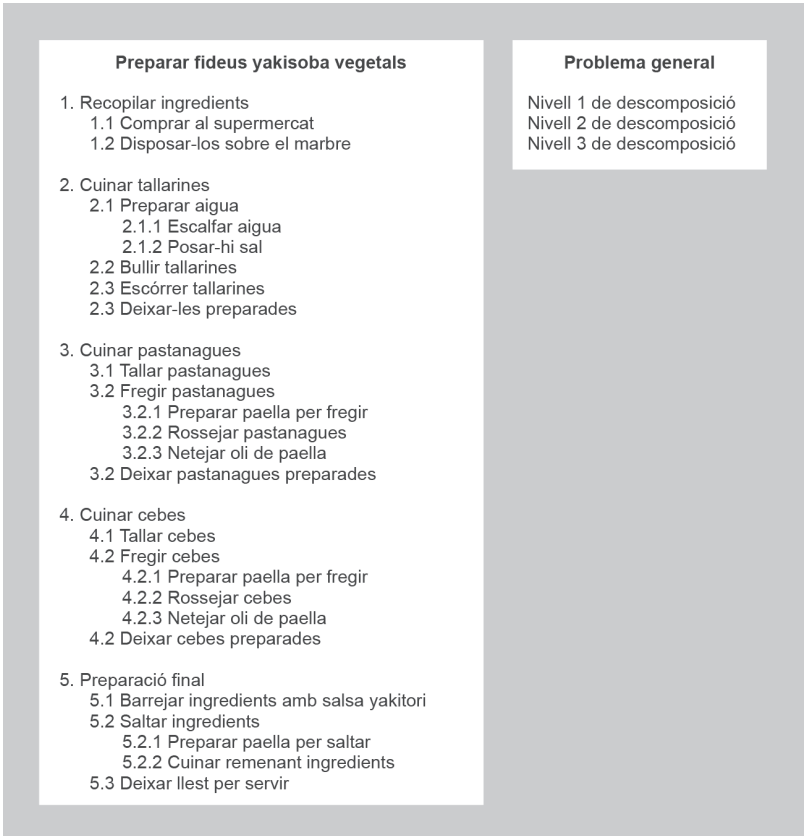
Una recepta de cuina

Si bé l'exemple del document de text és pràcticament dels més directes que hi ha per il·lustrar com la divisió d'un problema en d'altres més petits és de gran ajut, hi ha un petit detall a tenir en compte. La majoria de documents de text, i és el cas de les unitats formatives, no descriuen pas algorismes, que és al cap i a la fi el que haureu de fer en un programa. Ara bé, hi ha prou que el text descrigui una seqüència de passes per dur a terme una fita perquè ja es converteixi en algorisme. Per exemple, un manual d'instruccions per muntar un moble o una recepta de cuina. En aquests casos, per fer la redacció, el procés general es divideix igualment en apartats i seccions que es corresponen a tasques individuals i concretes dins del procés general.

Una recepta de cuina és un algorisme, i es pot descompondre usant disseny descendent (Font: jetalone)

Per tant, el que es proposa fer, a mode d'exemple, és dur a terme un procés de descomposició en subproblemes d'una recepta de cuina de fideus japonesos *yakisoba* (焼きそば) sense carn. En un cas com aquest, el concepte de problema complex és relatiu, ja que tot depèn de les habilitats i coneixements culinaris del lector. Per no simplificar massa l'exemple, se suposarà que alguns aspectes com fregir o saltar no es consideren tasques simples, i cal tenir ben present el procés de preparació. La figura 1.3 mostra una proposta d'esquema de descomposició usant disseny descendent. El format emprat per establir el nivell de descomposició és el mateix que en l'exemple anterior. Estudieu-la atentament i dediqueu uns moments a reflexionar si vosaltres ho hauríeu fet d'una altra manera.

Figura 1.3. Descomposició de la preparació d'una recepta de cuina



Atès que aquest exemple sí que està expressant la descomposició d'un procés, hi ha alguns aspectes a tenir en compte. D'una banda, noteu com la nomenclatura usada per identificar cada subproblema indica clarament què es vol assolir, de manera que tant vosaltres com un tercer observador pot tenir una idea clara d'allò que cal resoldre per dur a terme la tasca final. Això és molt important. Ara bé, d'altra banda, si bé amb aquest identificador se sap "què" cal resoldre, no se sap "com" es resol cada subproblema. En aquest sentit, es considera que un subproblema és una abstracció sobre part del procés complet. Definir l'algorisme que resol cada subproblema individual serà ja l'etapa següent.

Per exemple, un cop es té clara la descomposició, ja es podria decidir cercar l'algorisme per resoldre el subproblema "Preparar oli per fregir", que podria ser:

1. Agafar ampolla d'oli de gira-sol.
2. Omplir paella fins a un terç.
3. Posar foc al màxim.
4. Mentre l'oli no s'escalfi.
5. Esperar.

Noteu com, per resoldre aquest punt, no és necessari saber absolutament res de la resta del procés general. Això indica que cada subproblema del nivell més baix planteja un seguit de tasques totalment autocontigudes. Aquest procés

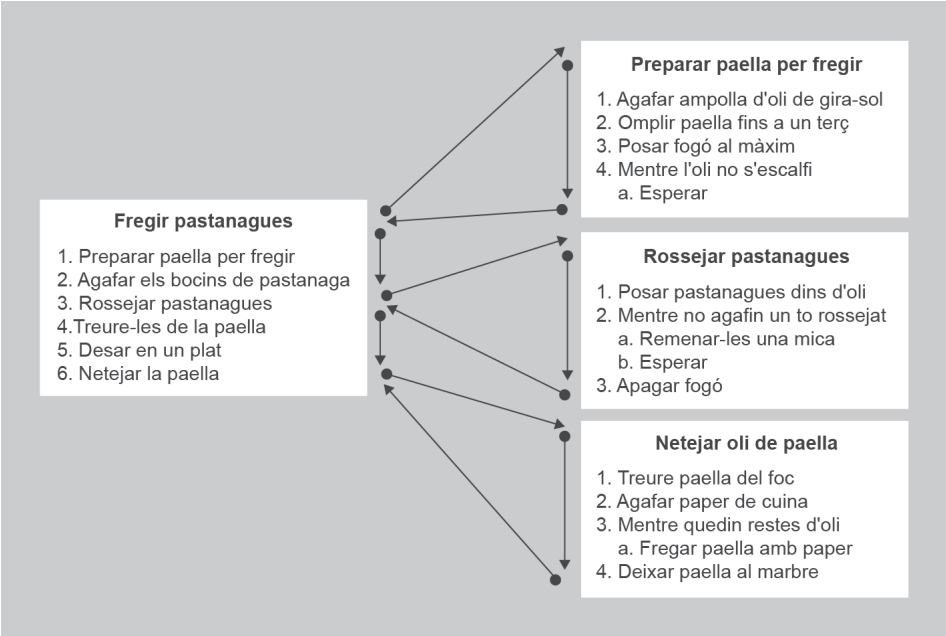
llavors s'aniria repetint per cada subproblema, normalment començant resolent els problemes més senzills (nivells inferiors), i anant a poc a poc resolent els més complexos (nivells superiors), fins a arribar al problema general, que és el de nivell més alt.

Aquest ordre recomanat es deu al fet que, durant aquest procés, per resoldre subproblemes de nivells superiors, és possible referir-se a subproblemes de nivells inferiors. La millor manera de veure això és veient com es resoldria, per exemple, el subproblema "Fregir pastanagues":

1. **Preparar oli per fregir**.
2. Agafar els bocins de pastanaga.
3. **Rossejar pastanagues**.
4. Treure-les de la paella.
5. Desar en un plat.
6. **Netejar la paella**.

Noteu com, per resoldre aquest subproblema, en els punts 1, 3 i 6 precisament s'està referint a subproblemes de nivell inferior, que es consideren resolts si seguim l'ordre de resolució descrit. També noteu com, per resoldre un subproblema, tant es poden usar subproblemes ja resolts de nivell inferior com passes addicionals que es considerin prou simples. Per tant, des del punt de vista d'ordre de les passes que s'estan seguint, el procés seguiria a grans trets el flux de control de la figura 1.4.

Figura 1.4. Flux de control de l'algorisme basat en disseny descendent per fregir unes pastanagues



En aquesta figura, el punt més important és veure com cada subproblema es considera una entitat estrictament independent i autocontinguda dins de tot el

procés, a la qual s'hi accedeix des d'un altre de nivell superior. Quan es fa, les seves passes són seguides d'inici a fi, i en acabar es continua exactament per on us havíeu quedat en el nivell superior.

1.1.2 Reutilització de subproblemes resolts

Els exemples de la unitat formativa o la recepta de cuina, a simple vista, aparenten seguir un esquema molt similar. Els diferents nivells segueixen una ordenació seqüencial (1, 1.1, 1.1.1 ... 2, 2.1, etc.), de manera que, fet i fet, els subproblemes es van resolent de manera ordenada i un cop resolt el darrer subproblema, la tasca general està pràcticament finalitzada. Això encaixa amb el model estrictament jeràrquic del problema general descompost tal com s'ha exposat inicialment.


Reaprofitant subproblemes resolts us estalvieu reinventar la roda. Font: Derek Ramsey

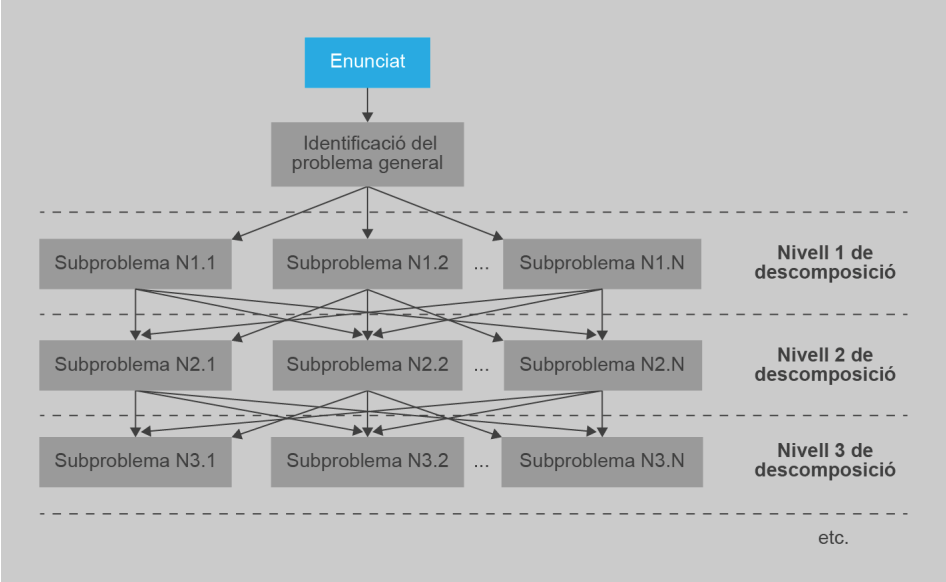
Ara bé, la descomposició mitjançant disseny descendent permet fer ús d'una característica molt útil quan s'usa per dissenyar algorismes. Es tracta de la possibilitat de cercar subproblemes idèntics, o si més no força semblants, i reaprofitar la seva solució en més d'un lloc dins del problema general. Un cop s'han resolt una vegada, no tindria sentit tornar-los a resoldre de nou repetides vegades. Sobre aquesta circumstància, de moment s'estudiarà només el cas de subproblemes exactament iguals.

Per exemple, si us fixeu en la descomposició de la recepta de cuina, podeu observar que hi ha subproblemes repetits. Es tracta de "Preparar paella per fregir" i "Netejar paella". No només s'han descrit ja d'entrada amb noms idèntics, sinó que, si us pareu a pensar, les passes que engloben també ho seran. Els elements que es manipulen per a la seva resolució (paella i oli) i la manera com es fa aquesta manipulació són exactament els mateixos. Per tant, un cop s'han definit les passes per resoldre'l la primera vegada, ja no cal tornar-ho a fer.

Aneu amb compte, ja que per considerar que dues solucions són idèntiques, els elements que intervenen han de ser exactament els mateixos. Així, doncs, "Tallar cebes" i "Tallar pastanagues" són certament subproblemes molt semblants, però no realment idèntics, ja que es manipulen elements diferents.

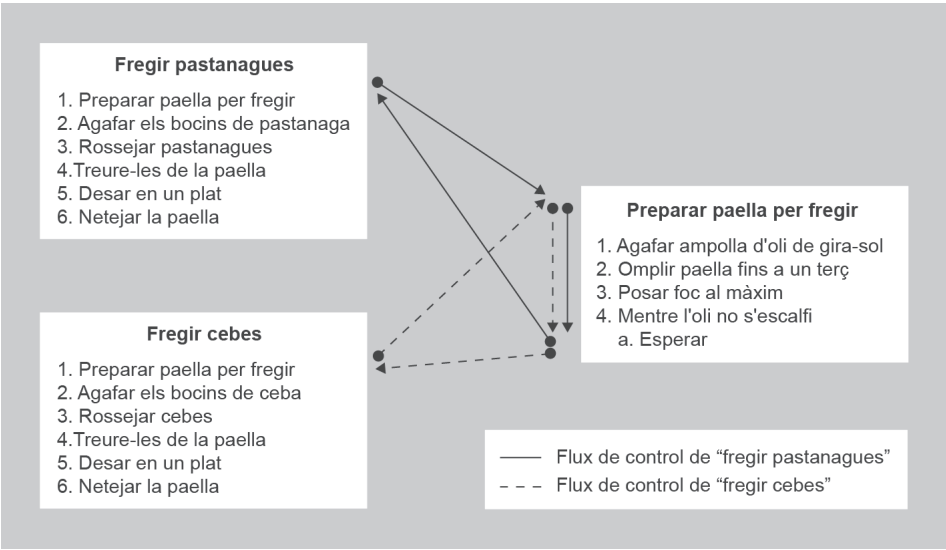
Un cop detectada aquesta característica del disseny descendent, és el moment de matisar la descripció de descomposició en nivells de la figura 1.1. En realitat, el procés és més aviat semblant al que descriu la figura 1.5, la qual indica un canvi de plantejament, ja que qualsevol subproblema d'un nivell donat pot ser part de qualsevol subproblema d'un nivell superior. Per remarcar aquest fet, a la figura cada subproblema no s'enumera usant un índex associat al subproblema de nivell superior on pertany, sinó directament d'acord al nivell on pertany. No hi ha exclusivitat dins la jerarquia.

Figura 1.5. Esquema d'aplicació de disseny descendent amb subproblemes repetits



Aquesta circumstància també fa que, a l'hora de considerar el flux de control de l'algorisme, aquest adopti una forma especial. Per resoldre dues tasques diferents es comparteix un mateix conjunt de passes, tal com mostra la figura 1.6, per al cas tot just esmentat. Aquest fet és important que el tingueu ben present, ja que té conseqüències molt directes a l'hora d'implementar un algorisme quan es tracta d'un programa d'ordinador.

Figura 1.6. Flux de control de l'algorisme basat en disseny descendent per fregir pastanagues o cebes



En conclusió, en descompondre un problema, és especialment encertat intentar fer-ho de manera que es forci l'aparició de subproblemes repetits, i així la seva resolució es pot reaprofitar en diversos llocs.

1.1.3 Aplicació correcta del disseny descendent

Un aspecte que heu de tenir en compte en aplicar disseny descendent és que es tracta d'una estratègia basada en unes directrius generals per atacar problemes complexos, però no és cap esquema determinista que us garanteixi que sempre obtindreu la millor solució. Això vol dir que, partint d'un mateix problema, diferents persones poden arribar a conclusions diferents sobre com dur a terme la descomposició. D'entre totes les solucions diferents possibles, algunes es poden considerar millors que d'altres. De fet, res impedeix, a partir ja una solució concreta, aplicar refinaments que la millorin. Per tant, és interessant poder avaluar si la descomposició que heu fet va per bon camí o no.

Alguns dels criteris en que us podeu basar per fer aquesta avaluació són els següents:

- Si un problema que sembla *a priori* força complex es descompon en molts pocs nivells, potser val la pena fer una segona ullada. Inversament, si un problema no massa complex té massa nivells, potser s'ha anat massa lluny en la descomposició.
- Veure si el nombre de passes incloses a cadascun dels subproblemes no és excessivament gran i és fàcil de seguir i entendre. En cas contrari, potser encara faria falta aplicar nous nivells de descomposició.
- Repassar que els noms assignats als subproblemes siguin autoexplicatius i expressin clarament la tasca que estan resolent. Sense ni tan sols llegir les seves passes, caldria entendre perfectament què s'assoleix en resoldre'ls. En cas contrari, potser la descomposició no està agrupant conjunts de passes realment relacionades entre elles.
- Si absolutament cap dels subproblemes és reutilitzat enlloc, especialment en descomposicions en molts nivells, és molt possible que no s'hagi triat correctament la manera de descompondre alguns subproblemes.
- Vinculat al punt anterior, l'aparició de subproblemes molt semblants o idèntics, però tractats per separat en diferents llocs, també sol ser indici que no s'està aplicant la capacitat de reutilitzar subproblemes correctament.

1.2 Disseny descendent aplicat a la creació de programes

Un cop disposeu d'un marc de referència general sobre com aplicar disseny descendent, és el moment d'aplicar la mateixa tècnica per a la creació d'un programa. Per començar, es mostra l'aplicació de disseny descendent sobre un programa de complexitat baixa i que ja coneixeu, de manera que el resultat de la descomposició sigui molt simple. Aquest serveix com a fil argumental per poder analitzar alguns dels aspectes importants del disseny descendent i veure com és

possible implementar la descomposició en subproblemes resultants, d'acord a la sintaxi del llenguatge Java.

El problema del que es parteix és, o hauria de ser, un conegut vostre: un programa que, a partir d'una llista de 10 valors enters, els mostri per pantalla ordenats.

Abans de començar, és una bona idea pensar quina mena de dades cal manipular i com s'emmagatzemaran, ja que això us permetrà avaluar en cada pas de la descomposició si un problema és massa complex encara o no. En aquest cas cal manipular una llista d'enters, per la qual cosa el més assenyat seria emmagatzemar-la en forma d'un *array*.

Com s'ha vist per al cas general d'aplicació de disseny descendent, una bona manera de descompondre el problema en subproblems és establir quines són les parts diferenciades, o etapes, que el composen. Cada etapa es correspondrà a un subproblema que cal resoldre. El més important en aquest pas és que cada subproblema correspongui sempre a una tasca concreta amb un objectiu a resoldre clarament diferenciat. Ha de ser fàcil assignar-li un nom. En cas contrari, segurament no s'està fent bé la descomposició.

Per a aquest programa, es pot considerar que el divideix en tres parts, o subproblemes, a resoldre:

- llegir la llista d'enters,
- ordenar-la, i
- mostrar-la per pantalla.

No totes les descomposicions han de tenir sempre molts nivells. Si el programa és simple, n'hi haurà pocs.

Un cop arribats a aquest primer nivell de descomposició, és el moment de plantejar-se si cada subproblema és massa complex o no. En aquest cas, se suposa que ja domineu el conjunt d'instruccions que calen, com llegir una llista d'enters de longitud coneguda (desant-los en un *array*), com ordenar un *array* (usant l'algorisme de la bombolla) i com mostrar-lo per pantalla (mitjançant un recorregut). Per tant, es pot considerar que tots els subproblemes plantejats no són excessivament complexos i el procés de descomposició acaba.

1.2.1 Declaració de mètodes

Un cop descompost el problema general, és el moment de crear el programa que el resol mitjançant codi font. Per això, caldrà decidir, per a cadascun dels subproblemes que s'han detectat, quines instruccions cal executar per resoldre'l individualment. Els llenguatges de programació permeten una implementació directa d'aquest procés, en oferir mecanismes per agrupar o catalogar blocs d'instruccions i etiquetar-los amb un identificador, d'acord al seu subproblema associat.

En general dins dels llenguatges de programació, s'anomena una **funció** a un conjunt d'instruccions amb un objectiu comú que es declaren de manera explícitament diferenciada dins del codi font mitjançant una etiqueta o identificador.

Per tant, per cada subproblema a resoldre, dins del vostre codi font s'haurà de definir una funció diferent. En el llenguatge Java, aquests conjunts d'instruccions se'ls anomena **mètodes**, en lloc de funcions, però a efectes pràctics, els podeu considerar el mateix. Aquest terme no és nou, ja que ha estat usat amb anterioritat sota dos contextos diferents, si bé mai s'havia entrat en molt de detall en la seva descripció ni s'havia justificat el seu format.

En algunes parts de la literatura, a les funcions que compleixen certes propietats se les anomena *accions*.

- Quan es parla de **mètode principal**, es tracta d'un conjunt d'instruccions que, etiquetades sota un identificador anomenat *main*, resolen el problema general (o sigui, tot el programa). Atès que fins al moment no s'havia aplicat disseny descendent, no hi havia subproblemes, i per tant en el vostre codi font només hi havia definit aquest únic mètode. No en calia cap altre.
- Quan es parla de la **invocació d'un mètode** sobre valors de certs tipus de dades complexos, com les cadenes de text (`String`), es tracta d'executar un conjunt d'instruccions amb un objectiu comú: transformar la cadena de text o obtenir dades contingudes.

Com podeu veure, tot i no haver entrat en detall, la manera com s'han usat fins al moment els mètodes és coherent amb la definició que tot just s'ha presentat. A partir d'ara començareu a estudiar-los amb més profunditat.

La declaració bàsica d'un mètode es fa usant la sintaxi que es mostra tot seguit. Com podeu veure, el seu format és molt semblant a com es declara el mètode principal (però no exactament igual, alerta!):

```
1 public void nomMetode() {
2     //Aquí dins aniran les seves instruccions
3     //...
4   }
```

Aquesta declaració es pot dur a terme en qualsevol lloc del fitxer de codi font, sempre que sigui entre les claus que identifiquen l'inici i fi de fitxer (`public class NomClasse { ... }`) i fora del bloc d'instruccions mètode principal, o qualsevol altre mètode. Normalment, se sol fer immediatament a continuació del mètode principal. La declaració ha de seguir exactament aquest format. L'única part que podeu modificar és `nomMetode`, que no és més que un identificador, com el d'una variable, i per tant podeu triar el que vulgueu. Tot i així, sempre hauríeu de procurar usar algun text que sigui autoexplicatiu.

Els identificadors dels mètodes segueixen les mateixes convencions de codi que les variables (lowerCamelCase).

D'acord a la descomposició proposada, dins el codi font del programa d'ordenació hi haurà declarats tres mètodes, un associat a cada subproblema. Aquests podrien ser:

```java
public class OrdenarDescendent {
  public static void main(String[] args) {
    //Instruccions del mètode principal (problema general)
    //...
  }
  //Mètode que resol el subproblema de llegir la llista.
  public void llegirLlista() {
    //Instruccions del mètode
    //...
  }
  //Mètode que resol el subproblema d'ordenar la llista.
  public void ordenarLlista() {
    //Instruccions del mètode
    //...
  }
  //Mètode que resol el subproblema de mostrar la llista per pantalla.
  public void mostrarLlista() {
    //Instruccions del mètode
    //...
  }
}
```

1.2.2 Canvis en el mètode principal en declarar altres mètodes

Abans de continuar, cal presentar un canvi necessari en el format dels vostres programes quan es vol declarar altres mètodes, associats a subproblemes, a part del mètode principal.

Per diferenciar els mètodes que resolen subproblemes del mètode principal, i evitar confusions, podeu referir-vos-hi com a *mètodes auxiliars*.

Concretament, per les característiques del llenguatge Java, cal que el mètode principal tingui un format molt concret. En cas contrari, hi haurà un error de compilació en futures passes del procés. Tot el codi que aniria normalment dins el bloc d'instruccions del mètode principal s'ubica en un nou mètode auxiliar, i dins el mètode principal simplement s'invoca aquest nou mètode. De fet, no és imprescindible que conegueu els detalls dels motius pels quals és necessari fer aquest canvi. Simplement podeu usar el codi següent d'exemple com a plantilla per generar els vostres programes, tenint en compte que tot el codi que posaríeu normalment al mètode principal, ara anirà al mètode `inici`.

```java
public class OrdenarDescendent {
  public static void main (String[] args) {
    //Aquí cal usar el nom de la classe que esteu creant.
    OrdenarDescendent programa = new OrdenarDescendent();
    programa.inici();
  }
  public void inici() {
    //Instruccions del mètode principal (problema general)
    //...
  }
  //Resta de mètodes
  //...
}
```

En usar aquest codi com a plantilla, noteu que a la línia següent caldria posar el nom de la classe que esteu editant, en lloc de `OrdenarDescendent`:

```java
OrdenarDescendent programa = new OrdenarDescendent();
```

1.2.3 Accessibilitat de variables dins una classe

En el moment que s'aplica disseny descendent i les parts del codi d'un programa es descomponen amb mètodes, apareix un problema. Els diferents mètodes que definiu serveixen per processar una informació comuna a tots tres, en aquest cas, la llista d'enters, en forma d'*array*. Això vol dir que us caldrà manipular aquesta variable en els diferents mètodes, de manera que els seus valors siguin compartits i accessibles per tots ells. Ara bé, abans de començar, caldrà decidir exactament a on es declararà.

Per prendre correctament aquesta decisió cal fer memòria i recordar el concepte d'àmbit d'una variable: donada una variable, només es considerarà declarada des de la línia on s'ha fet fins a trobar la clau tancada següent (}). Si us hi fixeu, la conseqüència directa d'això és que, atès que cada mètode pren la forma d'un bloc d'instruccions tancat entre claus ({ **...** }), si una variable es declara dins d'algun mètode, sigui quin sigui, aquesta no es considerà declarada en cap dels altres.

Hi ha diferents maneres de solucionar aquest problema. De moment en veureu la més simple. Qualsevol dada que hagi de ser accedida en més d'un subproblema per tal de resoldre'l, caldrà declarar-la com una variable global.

> Una **variable global** és una variable que pot ser accedida des de qualsevol instrucció dins un mateix fitxer de codi font. El seu àmbit és tot el fitxer.

En contraposició a les variables globals, hi ha les variables *locals*, que són les que heu usat fins ara: variables amb un àmbit local en un bloc concret de codi.

En Java, la sintaxi per declarar una variable global és molt semblant a la que s'ha vist fins ara, només varia el fet que cal afegir la paraula clau `private` abans de la declaració i el lloc on declarar-la. Aquest darrer punt és, de fet, el més important si voleu que una variable es consideri global.

```
1 private tipus nomVariable = valorInicial;
```

En aquest cas, la declaració s'ha de fer fora de tots els mètodes, però dins del bloc de claus que delimita la classe, exactament igual que quan declareu constants. Per exemple, si es vol declarar un *array* d'enters anomenat **llistaEnters**, es podria fer:

```
1  public class OrdenarDescendent {
2    //Variable global
3    private int[] llistaEnters = new int[10];
4    public static void main (String[] args) {
5      OrdenarDescendent programa = new OrdenarDescendent();
6      programa.inici();
7    }
8    public void inici() {
9      //Instruccions del mètode principal (problema general)
10     //...
11   }
12   //Mètode amb les instruccions per llegir la llista.
13   public void llegirLlista() {
14   }
```

```
15    //Mètode amb les instruccions per ordenar la llista.
16    public void ordenarLlista() {
17    }
18    //Mètode amb les instruccions per mostrar la llista per pantalla.
19    public void mostrarLlista() {
20    }
21  }
```

En ser global, la variable `llistaEnters` serà accessible des de qualsevol instrucció dins del codi. Val la pena remarcar que en Java no és imprescindible declarar-la a l'inici del codi font per poder ser usada lliurement. Per exemple, si es declarés a la darrera línia, tot just després de la declaració del mètode `mostrarLlista`, es continuaria considerant declarada per a tot el fitxer. Això és una lleugera diferència respecte a les variables locals, que només tenen vigència a partir de la línia de codi on s'han declarat. De totes maneres, per convenció, se solen declarar al principi de tot, de manera que el codi queda ordenat: primer variables globals i constants, i després mètodes.

Abans de continuar, és important remarcar que l'ús de variables globals només es considera polit en casos com aquest, on hi ha una dada que ha de ser manipulada en diferents subproblemes. Per a altres dades d'ús limitat a un únic mètode (comptadors o semàfors de bucles, resultats temporals d'operacions, etc.), caldrà declarar-les en el bloc de codi corresponent i mai com una variable global.

1.2.4 Codificació de mètodes

Un cop ja es disposa del mètode principal adaptat, les dades generals del programa declarades com variables globals i la declaració d'un mètode associat a cada suproblema resultant la descomposició del problema general, ja podeu procedir a escriure les instruccions de cada mètode. En aquest aspecte, les claus que delimiten un mètode (`public void nomMetode() { ... }`) conformen el bloc d'instruccions que resol aquell problema concret.

L'ordre en el qual caldrà resoldre'l és per nivells de descomposició, normalment començant pel nivell més baix, ja que és el més intuïtiu. Per les característiques de la descomposició mitjançant disseny descendent, si l'heu aplicat correctament, la resolució de cada mètode hauria de ser una tasca totalment autocontinguda i independent. Per tant, hauríeu de poder resoldre en qualsevol ordre els mètodes associats a problemes d'un mateix nivell de descomposició. Si apareix alguna dependència, és que la descomposició no és correcta.

En el cas de l'exemple, només hi ha dos nivells, el problema general i el primer nivell de descomposició. En el nivell més baix, el mètode `llegirLlista` tindrà les instruccions que llegeixen 10 enters des del teclat i els desen a l'*array*, el mètode **ordenarLlista**, les que ordenen l'*array* i el mètode `mostrarLlista`, les que el mostren per pantalla. Per tant, el codi podria ser:

```java
import java.util.Scanner;
public class OrdenarDescendent {
  int[] llistaEnters = new int[10];
  public static void main (String[] args) {
    OrdenarDescendent programa = new OrdenarDescendent();
    programa.inici();
  }
  public void inici() {
    //Instruccions del mètode principal (problema general)
    //...
  }
  //Mètode amb les instruccions per llegir la llista.
  public void llegirLlista() {
    System.out.println("Escriu 10 valors enters i prem retorn.");
    Scanner lector = new Scanner(System.in);
    int index = 0;
    while (index < llistaEnters.length) {
      if (lector.hasNextInt()) {
        llistaEnters[index] = lector.nextInt();
        index++;
      } else {
        lector.next();
      }
    }
    lector.nextLine();
  }
  //Mètode amb les instruccions per ordenar la llista.
  public void ordenarLlista() {
    for (int i = 0; i < llistaEnters.length - 1; i++) {
      for(int j = i + 1; j < llistaEnters.length; j++) {
        //La posició tractada té un valor més alt que el de la cerca... Els
            intercanviem.
        if (llistaEnters[i] > llistaEnters[j]) {
          //Per intercanviar valors cal una variable auxiliar
          int canvi = llistaEnters[i];
          llistaEnters[i] = llistaEnters[j];
          llistaEnters[j] = canvi;
        }
      }
    }
  }
  //Mètode amb les instruccions per mostrar la llista per pantalla.
  public void mostrarLlista() {
    System.out.print("L'array ordenat és: [ ");
    for (int i = 0; i < llistaEnters.length;i++) {
      System.out.print(llistaEnters[i] + " ");
    }
    System.out.println("]");
  }
}
```

1.2.5 Invocació de mètodes

Un cop resolts tots el mètodes d'un nivell donat, es pot procedir a resoldre els del nivell superior. Ara bé, en fer-ho, recordeu que teniu la possibilitat d'usar la solució de qualsevol subproblema de nivell inferior. Per exemple, aquest era el cas d'aprofitar saber preparar l'oli a la paella per solucionar com fregir les pastanagues a la recepta de cuina.

Dins del codi font, això es fa *invocant* algun dels mètodes que heu codificat. Per fer-ho, només cal posar una instrucció que conté el nom del mètode a invocar, dos

parèntesis i el punt i coma de final de sentència. O sigui:

```
nomMetode();
```

A efectes pràctics, cada cop que s'invoca un mètode, el programa executa les instruccions que hi ha codificades dins aquell mètode, des de la primera fins a la darrera. Quan acaba d'executar la darrera instrucció del mètode, llavors el programa procedeix a executar la línia immediatament posterior a la invocació al mètode.

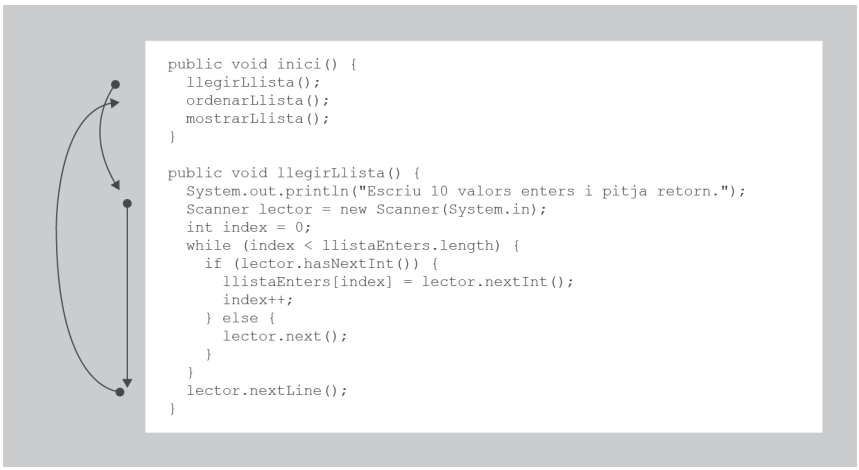
Si torneu a l'exemple, ara ja només us quedaria completar el codi associat al problema general (mètode **inici**). Per fer-ho, és possible invocar a **llegirLLista**, **ordenarLlista** i **mostrarLlista**. Donat aquest fet, el programa final ja seria el següent. Compileu-lo i executeu-lo per veure que és així.

```
import java.util.Scanner;
public class OrdenarDescendent {
  int[] llistaEnters = new int[10];
  public static void main (String[] args) {
    OrdenarDescendent programa = new OrdenarDescendent();
    programa.inici();
  }
  public void inici() {
    llegirLlista();
    ordenarLlista();
    mostrarLlista();
  }
  //Mètode amb les instruccions per llegir la llista.
  public void llegirLlista() {
    System.out.println("Escriu 10 valors enters i prem retorn.");
    Scanner lector = new Scanner(System.in);
    int index = 0;
    while (index < llistaEnters.length) {
      if (lector.hasNextInt()) {
        llistaEnters[index] = lector.nextInt();
        index++;
      } else {
        lector.next();
      }
    }
    lector.nextLine();
  }
  //Mètode amb les instruccions per ordenar la llista.
  public void ordenarLlista() {
    for (int i = 0; i < llistaEnters.length - 1; i++) {
      for(int j = i + 1; j < llistaEnters.length; j++) {
        //La posició tractada té un valor més alt que el de la cerca... Els
            intercanviem.
        if (llistaEnters[i] > llistaEnters[j]) {
          //Per intercanviar valors cal una variable auxiliar
          int canvi = llistaEnters[i];
          llistaEnters[i] = llistaEnters[j];
          llistaEnters[j] = canvi;
        }
      }
    }
  }
  //Mètode amb les instruccions per mostrar la llista per pantalla.
  public void mostrarLlista() {
    System.out.print("L'array ordenat és: [ ");
    for (int i = 0; i < llistaEnters.length;i++) {
      System.out.print(llistaEnters[i] + " ");
    }
    System.out.println("]");
  }
}
```

Un cop ja disposeu de tot el marc de referència sobre com queda distribuït un programa generat per la descomposició del problema usant disseny descendent, és interessant veure també quin és el flux de control de les instruccions quan aquest executa la invocació a un mètode. Això es mostra a la figura 1.7 per al cas de la invocació al mètode `llegirLlista`.

Repte 1: Modifiqueu el programa d'exemple de manera que faci el següent. Després de mostrar la llista ordenada, en una nova línia, ha de dir quants dels valors són inferiors a la meitat del valor més gran emmagatzemat. Apliqueu disseny descendent per afegir aquesta nova tasca, declarant i invocant els nous mètodes que faci falta.

Figura 1.7. Flux de control de les instruccions quan s'invoca el mètode llegirLlista.



1.2.6 Inicialització diferida de variables

La necessitat de declarar variables globals comporta una problemàtica en uns casos molt específics. Fins ara, en el moment de declarar una variable, immediatament li heu assignat un valor inicial. Aquest valor podia ser tant el resultat d'assignar directament un literal, com una expressió o una entrada de dades per part de l'usuari. Ara bé, en declarar una variable com a global, només pot ser inicialitzada directament mitjançant un literal o expressions on s'usen altres variables globals. No hi ha la possibilitat de fer-ho mitjançant un valor que depengui d'una entrada, per teclat o dels arguments del mètode principal. Això significa que la declaració de la variable i l'assignació del valor que es vol tractar realment no es pot fer a la mateixa instrucció. Tot i així, sempre que es declara una variable cal assignar-li un valor inicial.

Per al cas de variables de tipus primitius, resoldre aquest problema és simple. Per convenció, se li assigna inicialment el valor 0 i més endavant ja se sobreescriurà el seu valor amb un altre vàlid. Ara bé, per al cas de variables complexes (com les de tipus *array* o **String**), cal assignar un valor especial que en Java s'anomena **null**. Aquesta cadena de text és una paraula reservada del llenguatge que serveix per dir que, de moment, la variable està declarada però més endavant ja se li assignarà un valor correcte, tan aviat com sigui possible.

> Operar amb qualsevol variable de tipus complex amb un valor `null` assignat **sempre** resultarà en un programa erroni.

Per exemple, suposeu que voleu modificar el programa anterior de manera que, en lloc d'entrar deu valors pel teclat, en podeu entrar un nombre arbitrari. A partir de la seqüència escrita, el primer valor indicarà quants enters cal llegir tot seguit pel teclat. En un cas com aquest, és impossible inicialitzar l'*array* amb una mida concreta, ja que aquesta depèn d'una entrada pel teclat. Però el programa requereix que aquest sigui declarat com una variable global. Per tant, cal diferir la inicialització.

El codi que resol aquesta situació seria el següent. Atès que llegir la seqüència d'enters del teclat ara és un problema més complex, cal llegir la mida i els valors. S'ha aplicat disseny descendent per dividir-lo en dos subproblemes: llegir la mida de la seqüència i la seqüencia pròpiament. Comproveu que funciona en el vostre entorn de treball.

```
1  //Variable global. Array no inicialitzat.
2    private int[] llistaEnters = null;
3    //En aplicar disseny descendent, ara cal declarar "lector" com a global
4    Scanner lector = new Scanner(System.in);
5    public static void main (String[] args) {
6      OrdenarDescendentVariable programa = new OrdenarDescendentVariable();
7      programa.inici();
8    }
9    public void inici() {
10     llegirLlista();
11     ordenarLlista();
12     mostrarLlista();
13   }
14   //Mètode amb les instruccions per llegir la llista.
15   //El primer valor sera la llargària
16   public void llegirLlista() {
17     System.out.println("Escriu una llista de valors enters i prem retorn.");
18     System.out.println("El primer valor indica la mida de la seqüència.");
19     llegirMida();
20     llegirValors();
21   }
22   public void llegirMida() {        //Metode que llegeix el primer valor
23     //Lectura
24     int mida = 0;
25     if (lector.hasNextInt()) {
26       mida = lector.nextInt();
27     } else {
28       lector.next();
29     }
30     llistaEnters = new int[mida];   //Inicialització diferida de l'array
31   }
32   public void llegirValors() {
33     int index = 0;
34     while (index < llistaEnters.length) {
35       if (lector.hasNextInt()) {
36         llistaEnters[index] = lector.nextInt();
37         index++;
38       } else {
39         lector.next();
40       }
41     }
42     lector.nextLine();
43   }                                 //La resta de mètodes no canvien ...
44 }
```

1.3 Un exemple més complex

Un cop ja s'ha vist un exemple senzill d'aplicació de disseny descendent, bàsicament amb l'objectiu d'introduir la sintaxi per a la declaració de mètodes i variables globals en Java, és el moment de proposar-ne un altre de més complex, que requereixi diversos graus de descomposició.

El que es vol fer és un gestor de registre de temperatures preses setmanalment per un observatori. Es pressuposa que el programa es posa en marxa a l'inici de l'any (1 de gener) i al principi de cada setmana. Al llarg de 52 setmanes que té un any, es van enregistrant les temperatures mesurades cada dia de la setmana anterior (o sigui, set en total cada vegada). Cada cop que es fa un registre, sabent que ha passat una setmana, el programa calcula automàticament quin dia i mes és l'actual. A partir d'aquestes dades, és possible consultar en qualsevol moment quina ha estat la temperatura mitjana i la diferència entre el valor màxim i mínim enregistrats. En fer-ho, la data actual també es mostra en pantalla.

Totes aquestes accions es porten a terme usant un menú. Evidentment, l'aplicació ha de ser prou robusta com per tractar casos erronis (per exemple, consultar valors quan encara no hi ha cap data enregistrada, o intentar registrar com a temperatura valors de tipus incorrecte).

Per deixar més clar el comportament esperat, tot seguit es mostra un prototip del que s'esperaria mostrar amb la seva execució:

```
Benvingut al registre de temperatures
─────────────────────────────────────
[RT] Registrar temperatures setmanals.
[MJ] Consultar temperatura mitjana.
[DF] Consultar diferència màxima.
[FI] Sortir.
Opció: MJ
No hi ha temperatures registrades.

Benvingut al registre de temperatures
─────────────────────────────────────
[RT] Registrar temperatures setmanals.
[MJ] Consultar temperatura mitjana.
[DF] Consultar diferència màxima.
[FI] Sortir.
Opció: RT
Escriu les temperatures d'aquesta setmana:
20,5 21,1 21 21,7 20,9 20,6 19,9

Benvingut al registre de temperatures
─────────────────────────────────────
[RT] Registrar temperatures setmanals.
[MJ] Consultar temperatura mitjana.
[DF] Consultar diferència màxima.
[FI] Sortir.
Opció: MJ
Fins avui 8 de gener la mitjana ha estat de 20.814285 graus.

Benvingut al registre de temperatures
─────────────────────────────────────
[RT] Registrar temperatures setmanals.
[MJ] Consultar temperatura mitjana.
[DF] Consultar diferència màxima.
```

```
[FI] Sortir.
Opció: DF
Fins avui 8 de gener la diferència màxima ha estat de 1.8000011 graus.

Benvingut al registre de temperatures
_____________________________________
[RT] Registrar temperatures setmanals.
[MJ] Consultar temperatura mitjana.
[DF] Consultar diferència màxima.
[FI] Sortir.
Opció: FI
```

1.3.1 Descomposició de problema

Un cop s'ha plantejat amb cert detall el problema a resoldre (què ha de fer el programa), és possible iniciar la descomposició mitjançant disseny descendent. Per veure amb més detall el procés, aquesta vegada s'anirà fent a poc a poc i nivell per nivell.

1. Identificació de les dades a tractar

Abans de començar la descomposició, com a pas previ, és interessant establir quina mena de dades cal manipular i com emmagatzemar-les dins el programa. D'aquesta manera, resulta més fàcil avaluar per cada subproblema què ha de dur a terme i si es tracta d'una tasca complexa o no. En aquest cas, cal gestionar d'una llista de temperatures, que es pot emmagatzemar usant un *array* de reals, i una data dins un mateix any, que es pot emmagatzemar usant dos enters, dia i mes. L'*array* haurà de tenir espai per emmagatzemar els valors dels dies a 52 setmanes (52*7 = 364) i caldrà controlar el fet que hi ha posicions "buides" i d'altres amb valors correctes assignats. Per exemple, després de la primera setmana només els 7 primers valors, les posicions 0 a 6, són vàlids. La resta de posicions no tenen valors vàlids assignats.

Quan arribi el moment caldrà considerar si declarar-les com a variables globals, depenent de si aquestes dades s'usen dins de més d'un subproblema o no.

2. Primer nivell de descomposició

El resultat d'aplicar el primer nivell de descomposició pot resultar en molts o pocs subproblemes depenent del grau de granularitat amb què decidiu tractar les tasques que realitza el programa. Inicialment, és recomanable no usar una granularitat alta i mantenir un nivell d'abstracció alt. Normalment, és important no baixar ràpidament de nivell i començar a resoldre problemes molt concrets en una sola passada. Una estratègia per evitar això es plantejar-se quines accions cal emprendre abans de poder-ne dur a terme unes altres.

Partint de la descripció del problema, una possible descomposició en nivells seria la següent, encara força general. El programa bàsicament és una estructura de repetició que va iterant sobre aquestes dues tasques:

- Mostrar menú.
- Tractar ordre.

Aquestes iteracions s'aniran repetint fins a complir la condició que vol finalitzar el programa. D'entrada, es pot decidir que això es durà a terme amb una variable de control de tipus semàfor.

Fins a cert punt, per a un dissenyador novell, seria comprensible proposar ja en el primer nivell resoldre subproblemes tals com el càlcul de les temperatures mínimes i màximes, ja que són aspectes que ressalten clarament a l'enunciat. Però si reflexioneu, us adonareu que per poder dur a terme aquestes tasques hi ha condicions prèvies que abans cal complir: que l'usuari hagi seleccionat una opció. Per tant, això vol dir que gestionar el menú i executar les opcions té relació de jerarquia dins el disseny: primer llegiu l'opció i després l'executeu. Per tant, no es troben en el mateix nivell.

3. Segon nivell de descomposició

Per veure si cal un segon nivell cal avaluar si els subproblemes proposats en el primer nivell són massa complexos encara. Evidentment, depenent de la destresa del programador, el que es considera complex pot ser molt relatiu. En qualsevol cas, i això és independent de l'habilitat del programador, el que cal identificar són tasques clarament diferenciades que cal resoldre per solucionar cada subproblema.

- **Mostrar menú**. Per fer això, bàsicament només cal imprimir un conjunt de text en pantalla i ja està. És una tasca molt simple que es pot dur a terme mitjançant successives invocacions a `System.out.println`. Per tant, no cal descompondre-la més.

- **Tractar ordre**. Cal llegir l'ordre pel teclat i cal analitzar si el que es llegeix es correspon a alguna de les quatre ordres possibles. Això es pot fer amb una estructura de selecció. Llavors, segons el que s'ha llegit, cal fer tasques totalment diferents. Clarament, es tracta d'una tasca complexa que cal descompondre. La manera més lògica de fer-ho, inicialment, podria ser per cada tasca que cal dur a terme.

A partir d'aquesta anàlisi, la descomposició fins al segon nivell quedaria com:

1. Mostrar menú.
2. Tractar ordre.
   (a) Entrar registre de temperatures setmanals.
   (b) Mostrar temperatura mitjana.
   (c) Mostrar diferència màxima.
   (d) Finalitzar execució.

4. Tercer nivell de descomposició

Novament, es fa una iteració sobre els subproblemes de segon nivell per veure si presenten tasques complexes o no. En funció d'això, caldrà decidir si cal seguir descomposant-los.

- **Entrar registre de temperatures setmanals**. Per fer això cal resoldre dues tasques. D'una banda, llegir les temperatures i posar-les a l'*array* de temperatures. A més, també cal anar actualitzant la data actual cada cop que es llegeixen dades (avançar-la 7 dies). Això no és simple, ja que cal controlar el cas de quin dia acaba cada mes (28, 30 o 31 dies).
- **Mostrar temperatura mitjana**. Aquest problema es pot descompondre en dos. D'una banda, es demana mostrar la data de manera que el mes es mostri en format text, partint d'un número. D'altra banda, cal mostrar el càlcul que es demana (sumar tots els valors a l'*array* de temperatures i dividir-los pel seu nombre).
- **Mostrar diferencia màxima**. Aquest cas és exactament igual que l'anterior, només que el càlcul és diferent (cercar amb un únic recorregut els valors màxim i mínim i calcular-ne la diferència).
- **Finalitzar execució**. Bàsicament, seria canviar el valor de la variable de control de tipus semàfor que controla l'estructura de repetició on s'englobaran "Mostrar menú" i "Tractar ordre". És molt simple.

Segons aquesta anàlisi, la descomposició fins al tercer nivell quedaria així:

1. Mostrar menú.
2. Tractar ordre.
   (a) Entrar registre de temperatures setmanals.
      i. Llegir temperatures del teclat.
      ii. Actualitzar data actual.
   (b) Mostrar temperatura mitjana.
      i. Mostrar data actual.
      ii. Calcular temperatura mitjana.
   (c) Mostrar diferència màxima.
      i. Mostrar data actual.
      ii. Calcular diferència màxima.
   (d) Finalitzar execució.

5. Quart nivell de descomposició

Novament, correspon estudiar si cal fer un nou nivell de descomposició segons el grau de complexitat dels subproblemes de tercer nivell. Un punt interessant que ara us trobeu és el fet que es poden localitzar subproblemes repetits. Mostrar la data actual és una tasca que cal fer en llocs diferents. Per tant, només caldrà resoldre aquest subproblema una única vegada.

- **Llegir temperatures del teclat**. Tot i que no es fa en poques línies de codi, sabeu llegir 7 valors de tipus real i assignar-los a un *array*. Per tant, no és una tasca especialment complexa que valgui la pena descompondre més.
- **Actualitzar data actual**. Es tracta d'incrementar el dia i, depenent del mes, amb una estructura de selecció, veure si s'ha avançat a un nou mes. No es compon de passes gaire complexes.
- **Mostrar data actual**. Es tracta de mostrar el dia directament i mostrar cert text segons el valor numèric del més. Això es podria fer amb una estructura de selecció. Per tant, tampoc es compon de passes gaire complexes..
- **Calcular temperatura mitjana**. És un càlcul sobre els valors registrats, fent un recorregut sobre l'*array*. És simple.
- **Calcular diferència màxima**. Exactament un cas molt semblant a l'anterior.

Atès que tots els subproblemes del tercer nivell ja són simples i resolen una tasca molt concreta i autocontinguda, no cal un quart nivell de descomposició. Heu acabat.

1.3.2 Esquelet de la classe

Un cop identificades les dades que es volen tractar i finalitzat el procés de descomposició inicial, és possible crear un esquelet de la classe, només amb la declaració de variables globals i mètodes necessaris. Cada subproblema equival a un mètode que cal declarar. Si l'esquelet està correctament declarat, hauria de ser possible compilar el codi font, tot i que en executar-se no faria absolutament res encara. Només es tracta d'una organització general del codi font.

En aquest exemple l'esquelet quedaria així:

```
1  public class RegistreTemperatures {
2
3    //Constants
4    private static final int MAX_SETMANES = 52;
5
6    //Variables globals
7    private int numTemperatures = 0;
8    private float[] temperatures = new float[MAX_SETMANES * 7];
9    private int dia = 1;
10   private int mes = 1;
11
12   //Mètodes associats al problema general
13   public static void main (String[] args) {
14     RegistreTemperatures programa = new RegistreTemperatures();
15     programa.inici();
16   }
17   public void inici() {
18   }
19
20   //Mètodes associats al primer nivell de descomposició
21   public void mostrarMenu() {
22   }
```

```
23    public void tractarOpcio() {
24    }
25
26    //Mètodes associats al segon nivell de descomposició
27    public void registreTemperaturesSetmanals() {
28    }
29    public void mostrarMitjana() {
30    }
31    public void mostrarDiferencia() {
32    }
33    public void finalitzarExecució() {
34    }
35
36    //Mètodes associats al tercer nivell de descomposició
37    //etc.
38  }
```

Repte 2: Completeu el codi font de l'esquelet de l'exemple amb la declaració dels mètodes al tercer nivell de descomposició. Els seus noms seran: `llegirTemperaturesTeclat`, `incrementarData`, `mostrarData`, `calculaMitjana` i `calculaDiferencia`.

1.3.3 Implementació del tercer nivell de descomposició

Per codificar la descomposició d'aquest problema, també es començarà des dels mètodes associats als subproblemes del nivell més baix de descomposició i s'anirà pujant a poc a poc fins a arribar a la resolució del problema general, que correspon al mètode principal. Per tant, cal codificar els mètodes:

- `llegirTemperaturesTeclat`: registra 7 temperatures, o sigui, llegeix 7 valors reals i els desa a l'*array* de temperatures.
- `incrementarData`: donada una data, suma 7 al seu valor.
- `mostrarData`: mostra la data actual, en format: *Número del dia* de *Nom del mes*.
- `calculaMitjana`: mostra per pantalla la mitjana aritmètica de temperatures del registre.
- `calculaDiferencia`: mostra per pantalla la diferència de temperatures entre els valors màxim i mínim del registre.

Una proposta de com fer el seu codi seria la següent: atès que els mètodes són blocs autocontiguts de codi, un cop incorporats al codi font del programa, és possible compilar-lo perfectament sense problemes, tot i que, evidentment, el programa encara no farà res si s'executa. Afegiu-los i comproveu que és així.

```
1   //Mètodes associats al tercer nivell de descomposició
2     public void llegirTemperaturesTeclat() {
3       System.out.println("Escriu les temperatures d'aquesta setmana:");
4       Scanner lector = new Scanner(System.in);
5       int numLlegides = 0;
6       while (numLlegides < 7) {
7         if (lector.hasNextFloat()) {
```

```
        temperatures[numTemperatures] = lector.nextFloat();
        numLlegides++;
        numTemperatures++;
      } else {
        lector.next();
      }
    }
    lector.nextLine();
  }
  public void incrementarData() {
    //Quants dies té aquest mes?
    int diesAquestMes = 0;
    if (mes == 2) {
      diesAquestMes = 28;
    } else if ((mes == )||(mes == 6)||(mes == )||(mes == 11)) {
      diesAquestMes = 30;
    } else {
      diesAquestMes = 31;
    }
    dia = dia + 7;
    //Hem passat de mes?
    if (dia > diesAquestMes) {
      dia = dia – diesAquestMes;
      mes++;
      //Hem passat d'any?
      if (mes > 12) {
        mes = 1;
      }
    }
  }
  public void mostrarData() {
    System.out.print(dia + " de ");
    switch(mes) {
      case 1:
        System.out.print("Gener"); break;
      case 2:
        System.out.print("Febrer"); break;
      case 3:
        System.out.print("Març"); break;
      case 4:
        System.out.print("Abril"); break;
      case 5:
        System.out.print("Maig"); break;
      case 6:
        System.out.print("Juny"); break;
      case 7:
        System.out.print("Juliol"); break;
      case 8:
        System.out.print("Agost"); break;
      case 9:
        System.out.print("Setembre"); break;
      case 10:
        System.out.print("Octubre"); break;
      case 11:
        System.out.print("Novembre"); break;
      case 12:
        System.out.print("Desembre");
    }
  }
  public void calculaMitjana() {
    float acumulador = 0;
    for(int i = 0; i < numTemperatures; i++) {
      acumulador = acumulador + temperatures[i];
    }
    System.out.print((acumulador / numTemperatures));
  }
  public void calculaDiferencia() {
    //Veure Repte 3, més endavant
    //...
  }
```

Repte 3: Codifiqueu el mètode `calculaDiferencia`. Explicat amb més detall, aquest mètode cerca els valors més alt i més baix d'entre els enregistrats i calcula la diferència entre ells. Un cop calculat, mostra el valor resultant per pantalla, tal com fa `calculaMitjana`.

1.3.4 Implementació del segon nivell de descomposició

Un cop acabada la codificació dels mètodes de nivell més baix de descomposició, cal resoldre el nivell immediatament superior, pas a pas, sense saltar-se cap nivell. Un aspecte interessant en anar resolent nivells superiors és que, si la descomposició ha estat apropiada, la dificultat de codificar tots els subproblemes, independentment del nivell, hauria de ser similar. En el cas dels nivells més baixos, això era degut al fet que es tracta dels problemes que heu considerat més simples, com ja heu vist. En nivell superiors, però, la seva complexitat també serà més baixa ja que es disposa d'una part del problema resolta. Per tant, la codificació d'aquest segon nivell no hauria de resultar haver de fer mètodes molt més complicats o necessàriament amb més codi que el pas anterior. De fet, fins i tot poden ser més senzills.

Els mètodes que estan inclosos en aquest nivell són els associats a les quatre opcions possibles dins el programa:

- `registreTemperaturesSetmanals`: gestiona el porcés de registrar temperatures setmanals: llegir dades, emmagatzemar-les i incrementar la data actual.
- `mostrarMitjana`: mostra per pantalla el missatge de la mitjana de les temperatures: data actual i valor.
- `mostrarDiferencia`: mostra per pantalla el missatge de la diferència màxima de les temperatures: data actual i valor.
- `finalitzarExecució`: gestiona la finalització del programa.

<iocstl textB>La codificació estrictament de nivells inferiors a superiors de manera totalment compartimentalitzada no sempre és possible. ::: A mesura que aneu pujant de nivells podeu trobar alguns casos especials, els quals no són infreqüents durant un procés de disseny descendent, i sobre ell us caldrà reflexionar.

D'una banda, es pot donar el cas en què caldrà aprofitar un mateix mètode de nivells inferiors per codificar més d'un mètode del nivell present. Aquest és el cas de `mostrarMitjana` i `mostrarDiferencia`, ja que ambdós han de mostrar la data actual, i per tant faran ús de `mostrarData`.

D'altra banda, la codificació d'un mètode pot no ser clara *a priori*, i no ho serà fins a solucionar nivells de descomposició superiors. Aquest és el cas de la codificació del mètode `finalitzarExecució`, ja que controla la finalització de

l'execució del programa principal, un mètode de nivell superior que encara no toca implementar. Aquest és un cas en què la resolució ordenada de nivell inferior a superior individualment no és perfecta, ja que cal una visió més general del problema. Quan això passa, cal deixar el codi buit fins que la solució quedi més clara més endavant.

El codi dels quatre mètodes, que preveu les circumstàncies tot just descrites, seria el següent. Afegiu-lo al programa i comproveu que compila correctament.

```
//Mètodes associats al segon nivell de descomposició
  public void registreTemperaturesSetmanals() {
    //Cal controlar si hi haurà espai per a aquests 7 registres
    if ((numTemperatures + 7) >=  temperatures.length) {
      System.out.println("No queda espai per a més temperatures.");
    } else {
      llegirTemperaturesTeclat();
      incrementarData();
    }
  }
  public void mostrarMitjana() {
    if (numTemperatures > 0) {
      System.out.print("\nFins avui ");
      mostrarData();
      System.out.print(" la mitjana ha estat de ");
      calculaMitjana();
      System.out.println(" graus.");
    } else {
      System.out.println("\nNo hi ha temperatures registrades.");
    }
  }
  public void mostrarDiferencia() {
    //Veure Repte 4, tot seguit.
    //...
  }
  public void finalitzarExecució() {
    //Ja es pensarà a resoldre el nivell superior...
  }
```

Repte 4: Codifiqueu el mètode `mostrarDiferencia`.

1.3.5 Implementació del primer nivell de descomposició

En aquest problema, el primer nivell de descomposició es correspon als subproblemes més generals, que engloben totes les funcions del programa. Només cal implementar dos mètodes:

- `mostrarMenu`: mostra el menú principal per pantalla (només imprimeix coses per pantalla).
- `tractarOpcio`: llegeix l'ordre i executa el mètode de segon nivell corresponent.

El codi que porta a terme aquestes tasques és el següent. Fixeu-vos que encara no heu resolt el mètode `finalitzarExecució`, però atès que se n'ha declarat l'esquelet, es pot invocar correctament sense que hi hagi cap error de compilació.

Simplement, ara per ara la seva invocació és equivalent a no fer res (no s'executa cap instrucció).

```
1  //Mètodes associats al primer nivell de descomposició
2    public void mostrarMenu() {
3      System.out.println("\nBenvingut al registre de temperatures");
4      System.out.println("————————————————————————————————————————");
5      System.out.println("[RT] Registrar temperatures setmanals.");
6      System.out.println("[MJ] Consultar temperatura mitjana.");
7      System.out.println("[DF] Consultar diferència màxima.");
8      System.out.println("[FI] Sortir.");
9      System.out.print("Opció: ");
10   }
11   public void tractarOpcio() {
12     Scanner lector = new Scanner(System.in);
13     String opcio = lector.nextLine();
14     if (opcio.equalsIgnoreCase("RT")) {
15       registreTemperaturesSetmanals();
16     } else if (opcio.equalsIgnoreCase("MJ")) {
17       mostrarMitjana();
18     } else if (opcio.equalsIgnoreCase("DF")) {
19       mostrarDiferencia();
20     } else if (opcio.equalsIgnoreCase("FI")) {
21       finalitzarExecució();
22     } else {
23       System.out.println("Opció incorrecta!\n");
24     }
25   }
```

equalsIgnoreCase

Aquest mètode de la classe `String` compara dues cadenes de text ignorant diferències entre majúscules i minúscules.

1.3.6 Implementació del problema general

Finalment, ha arribat el moment d'implementar el problema general, el mètode `inici`. Atès que tots els subproblemes inferiors ja han estat resolts, normalment aquesta tasca serà ja relativament simple. Pel funcionament que s'ha definit per a aquest programa, el que ha de fer bàsicament és mostrar el menú i llegir una ordre de manera indefinida, fins a demanar que finalitzi el programa, o sigui una estructura de repetició controlada per una variable de control amb funcions de semàfor, que canviarà d'estat quan calgui finalitzar les iteracions.

El seu codi seria el següent, on `fi` seria la variable de control. Com podeu veure, aquest és el mètode més senzill de tots!

```
1  public void inici() {
2      while (!fi) {
3        mostrarMenu();
4        tractarOpcio();
5      }
6    }
```

Un cop arribats a aquest punt, toca fer marxa enrere i recordar que hi havia una tasca pendent: codificar `finalitzarExecució`. Les seves instruccions no es podien deduir en el seu moment, ja que depenien de com es resoldria el codi del problema general. Ara que ja sabeu que la finalització del programa depèn d'una variable de control, ja es pot saber que cal canviar el seu valor de manera adient.

Per tant, el seu codi serà:

```
public void finalitzarExecució() {
    fi = true;
  }
```

Un cop codificat, només resta una cosa, i és declarar aquesta variable. Atès que es tracta d'un valor que cal accedir des de dos mètodes diferents, caldrà fer-ho com una variable global.

```
//Variables globals
  private boolean fi = false;
  private int numTemperatures = 0;
  private float[] temperatures = new float[MAX_SETMANES * 7];
  private int dia = 1;
  private int mes = 1;
  ...
```

1.3.7 Millores sobre la solució final

Un cop finalitzat el programa, és el moment de veure si funciona. Evidentment, pot ser que el codi d'algun mètode no hagi estat codificat correctament i sigui necessari corregir-lo. Per al cas dels programes complexos, el depurador és una eina de gran ajut en aquest tasca, complementada amb el fet que, usant mètodes, és molt fàcil identificar la utilitat de cada bloc de codi.

Tot i que el programa funcioni, un cop ja disposeu de tot el codi del programa (ja s'han deduït tots els mètodes, el seu codi, i les variables globals que cal usar), val la pena donar una ullada general per refinar el resultat. Aquest procés de refinament es basa en dos principis: eliminar mètodes massa curts o simplificar els que són encara massa llargs o complexos.

> Abans de procedir a millorar el codi, però, cal que tingueu sempre present la màxima següent: primer cal que el programa funcioni. Després ja pensareu com simplificar el codi.

Eliminació de mètodes

Un cop codificats tots els mètodes, potser hi haurà algun que té molt poques línies -estem parlant d'una o dues. Mai es reaprofita, i només s'usa en un únic lloc. Normalment, quan això passa es deu al fet que s'ha filat massa prim en el procés de descomposició i s'ha considerat com a subproblema una tasca que és molt senzilla i no té prou entitat en si mateixa. Que això succeeixi no vol dir que el procés hagi estat totalment incorrecte, ja que moltes vegades, *a priori*, és impossible saber que això passarà. Molts cops, només un cop codificats tots els mètodes us podeu adonar d'aquest fet i obrar en conseqüència.

Un cas clar d'aquesta circumstància pot ser el mètode `finalitzarExecució`, que només té una línia i només s'invoca en un únic lloc dins el programa. Normalment, no té sentit crear mètodes tan curts. Per tant, no seria incorrecte eliminar-lo i incorporar el seu codi directament allà on s'invoca (en el tractament de l'ordre "FI").

```
1  public void tractarOpcio() {
2      Scanner lector = new Scanner(System.in);
3      String opcio = lector.nextLine();
4      if (opcio.equalsIgnoreCase("RT")) {
5        registreTemperaturesSetmanals();
6      } else if (opcio.equalsIgnoreCase("MJ")) {
7        mostrarMitjana();
8      } else if (opcio.equalsIgnoreCase("DF")) {
9        mostrarDiferencia();
10     } else if (opcio.equalsIgnoreCase("FI")) {
11       //S'ha esborrat el mètode finalitzarExecució i s'ha posat el seu codi
             directament.
12       fi = true;
13     } else {
14       System.out.println("Opció Incorrecta!\n");
15     }
16   }
```

Millora de mètodes

A la secció "Recursos del contingut" del web disposeu d'un annex on s'explica una tècnica per simplificar el codi d'alguns mètodes amb estructures de selecció llargues.

Si al final del procés apareix algun mètode molt llarg o complex, això pot significar el contrari del cas anterior: que no s'ha descomposat prou el problema. Pot valer la pena tornar a aplicar el procés de descomposició *a posteriori*, dividint aquest mètode en d'altres. Tot i que pot semblar que, un cop el programa ja funciona, no val la pena refinar el procés de descomposició (al cap i a la fi, el seu propòsit era simplificar el procés de creació, que ja ha finalitzat), penseu que un altre avantatge del disseny descendent és facilitar la legibilitat del vostre codi. En el món de la programació tampoc us ha de fer mandra de ser polits i endreçats (amb el vostre codi).

Ara bé, de vegades el procés ha estat correcte i simplement la quantitat de línies de codi necessàries per dur a terme la tasca establerta és realment gran. En casos com aquests, igualment, és interessant repassar si el codi es pot millorar simplificant-lo, cercant algun conjunt de codi alternatiu que el faci més curt. Evidentment, això no sempre és possible, però val la pena fer-hi una pensada ara que ja teniu un programa que funciona.

1.4 Solució dels reptes proposats

Repte 1

```
import java.util.Scanner;
public class OrdenarDescendent {
  int[] llistaEnters = new int[10];
  public static void main (String[] args) {
    OrdenarDescendent programa = new OrdenarDescendent();
    programa.inici();
  }
  public void inici() {
    llegirLlista();
    ordenarLlista();
    mostrarLlista();
    comptarMeitatMaxim();
  }
  //Mètode amb les instruccions per llegir la llista.
  public void llegirLlista() {
    System.out.println("Escriu 10 valors enters i pitja retorn.");
    Scanner lector = new Scanner(System.in);
    int index = 0;
    while (index < llistaEnters.length) {
      if (lector.hasNextInt()) {
        llistaEnters[index] = lector.nextInt();
        index++;
      } else {
        lector.next();
      }
    }
    lector.nextLine();
  }
  //Mètode amb les instruccions per ordenar la llista.
  public void ordenarLlista() {
    for (int i = 0; i < llistaEnters.length - 1; i++) {
      for(int j = i + 1; j < llistaEnters.length; j++) {
        //La posició tractada té un valor més alt que el de la cerca... Els
            intercanviem.
        if (llistaEnters[i] > llistaEnters[j]) {
          //Per intercanviar valors cal una variable auxiliar
          int canvi = llistaEnters[i];
          llistaEnters[i] = llistaEnters[j];
          llistaEnters[j] = canvi;
        }
      }
    }
  }
  //Mètode amb les instruccions per mostrar la llista per pantalla.
  public void mostrarLlista() {
    System.out.print("L'array ordenat es: [ ");
    for (int i = 0; i < llistaEnters.length;i++) {
      System.out.print(llistaEnters[i] + " ");
    }
    System.out.println("]");
  }
  //Nou mètode per resoldre el nou subproblema
  public void comptarMeitatMaxim() {
    int valorMaxim = llistaEnters[llistaEnters.length - 1] / 2;
    int i = 0;
    while ((llistaEnters[i] < valorMaxim)&&(i < llistaEnters.length)) {
      i++;
    }
    System.out.println("El nombre de valors inferiors a la meitat del maxim és
        " + i);
  }
}
```

Repte 2

```
...
  //Mètodes associats al tercer nivell de descomposició
  public void llegirTemperaturesTeclat() {
  }
  public void incrementarData() {
  }
  public void mostrarData() {
  }
  public void calculaMitjana() {
  }
  public void calculaDiferencia() {
  }
  ...
```

Repte 3

```
public void calculaDiferencia() {
    float maxima = temperatures[0];
    float minima = temperatures[0];
    for(int i = 1; i < numTemperatures; i++) {
      if (temperatures[i] < minima) {
        minima = temperatures[i];
      }
      if (temperatures[i] > maxima) {
        maxima = temperatures[i];
      }
    }
    System.out.print((maxima – minima));
  }
```

Repte 4

```
public void mostrarDiferencia() {
    if (numTemperatures > 0) {
      System.out.print("\nFins avui ");
      mostrarData();
      System.out.print(" la diferència màxima ha estat de ");
      calculaDiferencia();
      System.out.println(" graus.");
    } else {
      System.out.println("\nNo hi ha temperatures registrades.");
    }
  }
```

2. Parametrització de mètodes

L'aparició de blocs de codi repetit o gairebé igual indica un mal estil de programació.

L'objectiu principal del disseny descendent és oferir una metodologia que us permeti plantejar la creació d'un programa d'una manera molt semblant a com ho faríeu amb qualsevol altre problema de la vida real: dividint problemes complexos en d'altres més simples i fàcils de seguir. Un cop s'ha fet aquesta descomposició, els llenguatges ofereixen un mecanisme per associar a cada subproblema un bloc de codi concret. En Java, es tracta dels mètodes. El benefici d'usar mètodes, en la mesura correcta, és la generació de codi més fàcil d'entendre i on pot ser possible reutilitzar alguns blocs a diferents parts, sense haver de repetir el codi escrit.

Si bé la identificació de subproblemes d'acord a diferents nivells de complexitat és suficient per poder descompondre el programa en bocins de codi amb objectius parcials diferenciats, i fer-lo més fàcil de seguir, no és suficient per oferir un alt grau de reusabilitat. En la majoria de casos, el programa resultant dividit amb mètodes és directament equivalent a trossejar un programa on només hi ha el mètode principal. Això es deu al fet que tal com heu usat els mètodes fins ara, aquests servien per manipular un conjunt de variables, sovint globals, molt específic.

En realitat, a l'hora de plantejar quins subproblemes hi ha, i la seva declaració associada en forma de mètode al codi, a part del seu nom i què fan, també és possible definir certs aspectes vinculats a les dades que han de tractar: els seus paràmetres d'entrada i de sortida.

Un **paràmetre** és un identificador usat dins la descripció d'un procés, el valor del qual en realitat pot variar per diferents aplicacions d'aquest procés.

2.1 Paràmetres d'entrada

Els paràmetres d'entrada són una eina molt útil i extensament usada a l'hora de definir amb més detall les característiques dels subproblemes resultants d'una descomposició usant disseny descendent. Per començar, i com a resum breu d'aquest concepte, podeu quedar-vos amb la definició següent.

Un **paràmetre d'entrada** és un valor que s'estableix immediatament abans de seguir un procés, de manera que indica les dades que ha de tractar o que modifica el seu comportament.

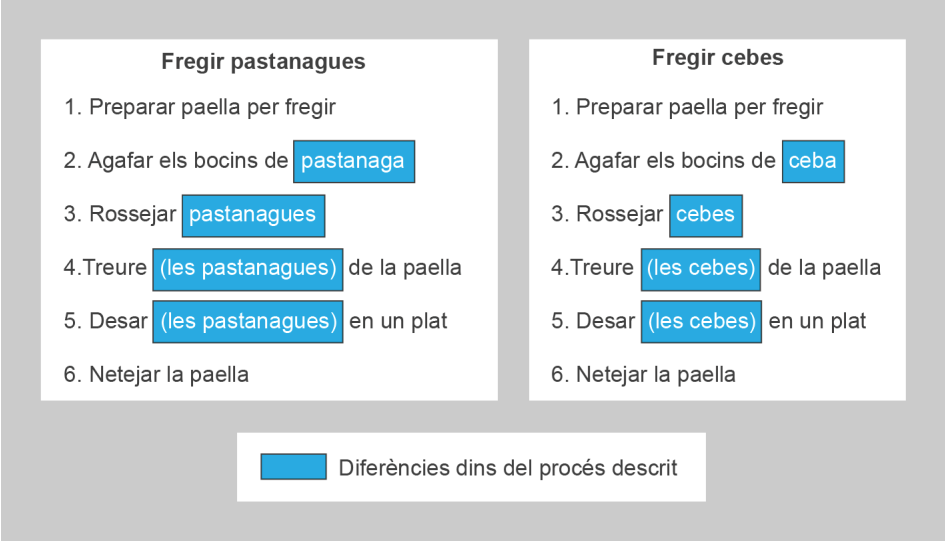
2.1.1 Motivació: definició de problemes semblants

Si es descompon fins a dos nivells el problema de cuinar uns fideus japonesos vegetals, es podrien trobar els subproblemes següents:

1. Recopilar ingredients
   (a) Comprar al supermercat
   (b) Disposar-los sobre el marbre
2. Cuinar tallarines
   (a) Preparar aigua
   (b) Bullir tallarines
   (c) Escórrer tallarines
   (d) Deixar-les preparades
3. Cuinar pastanagues
   (a) Tallar pastanagues
   (b) Fregir pastanagues
   (c) Deixar pastanagues preparades
4. Cuinar cebes
   (a) Tallar cebes
   (b) Fregir cebes
   (c) Deixar cebes preparades
5. Preparació final
   (a) Barrejar ingredients amb salsa yakitori
   (b) Saltar ingredients
   (c) Preparar paella per saltar
   (d) Cuinar remenant ingredients
   (e) Deixar llest per servir

Partint d'aquest plantejament, la figura 2.1 mostra la divisió ja més detallada, fins a un tercer nivell, dels processos de fregir cebes i pastanagues, usats en llocs ben diferents (3 i 4). En aquesta figura hi ha remarcades mitjançant quadres les diferències entre tots dos.

Figura 2.1. Diferències entre dos problemes semblants.



Ja a simple vista es pot veure que, tot i poder descompondre's fins en sis nous subproblemes, es tracta de dos procediments molt semblants. De fet, si analitzeu amb detall els dos casos, us adonareu que només canvia l'aliment que cal fregir, però els subproblemes que cal anar resolent són exactament els mateixos en tots els casos. De fet, si tinguéssiu una certa visió de futur, podríeu arribar a adonar-vos que aquest escenari s'aplicaria fins i tot en descomposicions d'altres procediments semblants que no us cal usar aquí, com, per exemple, fer patates fregides. L'única condició és que, quan parleu de fregir, sempre heu d'estar-vos referint a aliments, és clar.

Dit en paraules informàtiques: donats aquests dos mètodes, només varia el valor de les dades que cal transformar, que sempre seran d'idèntic tipus. Però l'algorisme que cal usar és el mateix.

La conseqüència d'aquesta observació és que seria molt més còmode definir un seguit de passes de manera genèrica. Per exemple, anomenar el procés *fregir aliment*, on només s'usa el terme *aliment* dins de la descripció de les seves passes per referir-se a allò que es vol fregir. Llavors, sempre que es vol fregir un aliment concret, en les explicacions es reemplaça on posa *aliment* per allò que realment es vol fregir (cebes, pastanagues, patates, etc.) i s'aplica el procés tal com està escrit, sense haver de canviar res. Una única descripció serveix per fregir qualsevol cosa, no cal descriure un cas diferent per cada aliment.

Doncs bé, si féssiu això, el terme *aliment* encaixaria amb la definició de *paràmetre d'entrada*. El seu valor s'estableix abans de començar el procés, i segons aquest, condiciona allò que cal tractar (en aquest cas, fregir).

Un fet important en un cas com aquest és que el nombre de paràmetres d'entrada permesos no té perquè estar limitat a un. Podríeu definir tants com us facin falta per cada diferència existent, o per cada dada addicional que pugui variar segons la situació i que cal tenir en compte per realitzar-lo. Per exemple, suposeu que voleu preveure la possibilitat de fregir amb diferents quantitats d'oli. O sigui, solucionar el problema "fregir aliment amb certa quantitat d'oli". En aquest cas, es podrien

usar dos paràmetres d'entrada: l'aliment i la quantitat d'oli. Les passes a seguir continuarien essent gairebé iguals, llevat que la descripció seria genèrica tant a l'hora de referir-se a l'aliment que cal fregir, com pel que fa a quant d'oli cal posar a la paella. Així, doncs, la descripció del procés seria independent tant quant a què cal fregir com pel que fa a la quantitat d'oli a usar. Els valors concrets que cal aplicar dependrien de cada situació donada a l'hora de fregir:

- Fregir patates amb 400 cl d'oli: (aliment = patates, quantitat = 400 cl).
- Fregir cebes amb 200 cl d'oli : (aliment = cebes , quantitat = 200 cl).
- Fregir escalopes amb 400 cl d'oli: (aliment = escalopes, quantitat = 400 cl).
- Etc.

És molt important triar correctament el nom dels subproblemes, perquè siguin descriptius.

Per tant, un cop teniu dividit un problema general en subproblemes, caldrà estudiar quins d'ells són prou semblants per, en realitat, poder considerar que es tracta del mateix mitjançant l'aplicació de paràmetres d'entrada. Normalment, si el nom que s'ha posat als subproblemes és adient, això és fàcil de detectar, ja que en tindran de molt semblants, o fins i tot idèntics. Pot succeir que les divisions en subproblemes no siguin exactament idèntiques com en aquest exemple, però llavors valdrà la pena que reviseu la descomposició i reflexioneu sobre si, en realitat, hauria de ser igual.

A mesura que tingueu experiència, no us caldrà ni tan sols esperar al final i ja de ben segur que anireu detectant aquestes semblances sobre la marxa.

2.1.2 Declaració i ús de mètodes amb paràmetres d'entrada

De la mateixa manera que això estalvia molt d'espai en un llibre de cuina, aplicar aquest sistema en aplicar disseny descendent per descompondre un programa també és molt útil. Això vol dir que aquest fet també s'ha de poder preveure a la declaració i codificació d'un mètode. Per sort, tots els llenguatges de programació que permeten declarar funcions o mètodes preveuen aquesta possibilitat.

Declaració

La llista de paràmetres d'un mètode s'escriu entre els parèntesis en la seva declaració. Aquesta llista pren una estructura semblant a fer un seguit de declaracions de variables separades per comes. La sintaxi exacta seria la següent:

```
1 public void nomMètode (tipusParam1 numParam1, tipusParam2 nomParam2, etc.) {
2   //Codi
3   ...
4 }
```

La declaració de paràmetres d'entrada no és obligatòria. Si no us calen paràmetres d'entrada per un mètode, no cal posar res entre els parèntesis, tal com heu fet fins

ara. Tampoc hi ha límit en el nombre de paràmetres, però normalment és millor no excedir-se. De fet, un alt nombre de paràmetres en el mètode associat a un subproblema pot significar que no s'ha descompost suficientment.

Un cop un mètode té definits un seguit de paràmetres d'entrada, aquests es consideren variables declarades, amb àmbit dins de tot el seu codi. Aquestes variables poden ser usades com qualsevol altra dins el seu codi. Ara bé, tenen una peculiaritat, i és que d'entrada no se'ls assigna cap valor inicial, ja que vindrà donat segons com s'invoqui el mètode, com veureu tot seguit.

El mètode principal (*main*) es declara amb un paràmetre d'entrada: un *array* de *String*.

```
//El mètode mostrarMaxim té dos paràmetres d'entrada, de tipus enter
  public void mostrarMaxim(int a, int b) {
    ...
  }
```

Únicament pel que fa a variables disponibles dins del bloc de codi del mètode, es pot considerar que aquest codi seria equivalent a fer:

```
public void mostrarMaxim() {
    //S'inicialitzen amb un valor inicial d'una manera especial que ja veureu
        aviat
    int a;
    int b;
    ...
  }
```

Invocació

Quan un mètode s'ha declarat amb paràmetres d'entrada, cal tenir-los en compte durant la seva invocació. No hi ha prou a posar només el nom del mètode com fins ara. Per cada paràmetre cal assignar un valor, exactament en el mateix ordre en què s'han declarat a la llista. Aquest valor pot ser el resultat de qualsevol expressió: un literal, una variable, o una expressió més complexa. En qualsevol cas, el tipus resultant ha de coincidir amb el del paràmetre que ocupa el mateix ordre. Sintàcticament, això es fa posant els valors entre els parèntesis de la invocació, separats per comes:

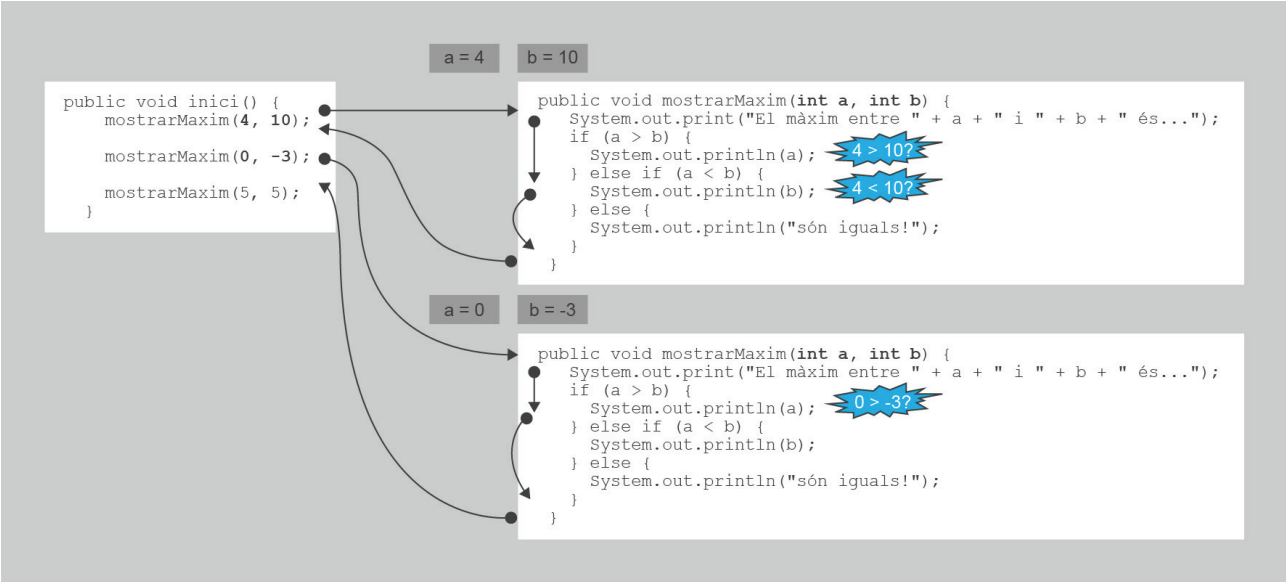
```
nomMetode(valor1, valor2, etc.);
```

Quan s'invoca un mètode amb un conjunt de paràmetres d'entrada declarats, el primer que fa el programa és mirar la llista de paràmetres i declarar, implícitament dins el codi del mètode, tantes variables com paràmetres d'entrada, amb els mateixos noms. Llavors agafa cada valor especificat a la invocació i copia, un per un i en el mateix ordre, aquests valors dins les variables creades a partir de la llista de paràmetres.

La millor manera de tenir una primera idea de com funciona tot plegat és amb un exemple senzill. El codi següent serveix per il·lustrar el comportament d'un mètode amb paràmetres d'entrada. No es tracta d'un programa complex resultat d'aplicar estrictament disseny descendent, només serveix per provar com funciona. Comproveu què fa en el vostre entorn de treball.

La figura 2.2, tot seguit, mostra un esquema del seu flux de control i què està passant a mesura que s'executen les instruccions del programa. Tot i que el codi del mètode `mostrarMaxim` es llista dos cops, només és per claredat, per diferenciar les execucions per cada invocació. En el codi font original només cal definir-lo un cop. Precisament, es tracta d'un cas de reaprofitament de codi. Fixeu-vos que la manera com es defineixen els paràmetres a la invocació del mètode serveix per donar valor a les variables `a` i `b` dins el mètode.

FIGURA 2.2. Flux de control en cridar un mètode i assignació de valors als seus paràmetres




```java
public class MaximParametreEntrada {
  public static void main (String[] args) {
    MaximParametreEntrada programa = new MaximParametreEntrada();
    programa.inici();
  }
  public void inici() {
    mostrarMaxim(4, 10);
    mostrarMaxim(0, -3);
    mostrarMaxim(5, 5);
  }
  //Té dos paràmetres d'entrada, de tipus enter
  //El seu valor inicial depèn dels valors assignats a cada invocació
  public void mostrarMaxim(int a, int b) {
    System.out.print("El màxim entre " + a + " i " + b + " és...");
    if (a > b) {
      System.out.println(a);
    } else if (a < b) {
      System.out.println(b);
    } else {
      System.out.println("són iguals!");
    }
  }
}
```

En aquest exemple els valors usats en invocar el mètode `mostrarMaxim` s'han expressat mitjançant literals, però també es podrien usar sense problemes variables o expressions, tal com mostra el codi llistat a continuació. En aquest cas, les expressions s'avaluen abans de procedir a executar les instruccions del mètode, de manera que sigui possible assignar correctament el valor resultant a cada paràmetre d'entrada. Proveu que fa exactament el mateix que l'exemple anterior.

```java
public class MaximParametreEntrada {
  public static void main (String[] args) {
    MaximParametreEntrada programa = new MaximParametreEntrada();
    programa.inici();
  }
  public void inici() {
    //Usem literals
    mostrarMaxim(4, 10);
    //Usem variables
    int i = 0;
    int j = -3;
    mostrarMaxim(i, j);
    //Usem expressions, amb literals o variables
    mostrarMaxim(2 + 3, i + 8);
  }
  //La resta del codi és igual...
}
```

Repte 1. Feu un programa que cridi diversos cops un mètode amb un únic paràmetre de tipus enter. Aquest mètode escriu per pantalla tants símbols "*" com el valor del paràmetre.

2.1.3 Manipulació dels paràmetres d'entrada

Un cop es comença a executar el codi del mètode, les variables representades pels seus paràmetres d'entrada poden ser usades com qualsevol variable declarada a la primera línia del programa: els seus valors poden ser llegits directament, poden usar-se dins d'expressions més o menys complexes, tenen un àmbit igual tot el mètode, etc. Fins i tot poden veure modificat el seu valor inicial. Ara bé, heu de tenir ben clar que els valors que contenen són còpies de l'original. Per tant, el valor usat a la invocació mai es veurà alterat.

Per exemple, proveu el codi següent i veieu com la variable `i`, usada a la invocació del codi, no es veu alterada en cap moment. Dins del paràmetre d'entrada `a` només hi ha una còpia del valor de `i`. No es tracta d'una associació directa.

```java
//Modifiqueu el valor d'un paràmetre. Afecta a la variable original?
public class ModificaParàmetre {
  public static void main (String[] args) {
    ModificaParàmetre programa = new ModificaParàmetre();
    programa.inici();
  }
  public void inici() {
    int i = 10;
    System.out.println("Abans de cridar el mètode \"i\" val " + i);
    modificarParametre(i);
    System.out.println("Després de cridar el mètode \"i\" val " + i);
  }
  //Té un únic paràmetre d'entrada, de tipus enter
  public void modificarParametre(int a) {
    //Ara hi ha una variable "a" declarada.
    //El seu valor depèn de com s'ha invocat el mètode.
    a = 0;
    System.out.println("Heu modificat el valor a " + a);
  }
}
```

El que ha succeït seria equivalent a haver fet:

```
//Es disposa d'una variable
int i = 10;
//Es crea una còpia de la variable
int a = i;
//Es modifica la còpia
a = 0;
//Quant val i? i encara val 10
```

En el llenguatge Java, però, aquesta norma només es compleix estrictament quan es tracta de tipus primitius. Quan el paràmetre és un *array*, sí que es modifica el valor. Si feu memòria, això es una conseqüència lògica de les seves propietats quan es copien variables de tipus *array*. En aquest cas, no es disposava realment de dues rèpliques idèntiques, sinó de dues variables diferents a partir de les quals accedir exactament a les mateixes dades.

Comproveu aquest fet mitjançant l'execució de l'exemple següent:

```
//Modifiqueu el valor d'un paràmetre. Afecta a la variable original?
public class ModificaParametreCompost {
  public static void main (String[] args) {
    ModificaParametreCompost programa = new ModificaParametreCompost();
    programa.inici();
  }
  public void inici() {
    int[] i = {1, 2, 3, 4, 5};
    System.out.println("Abans de cridar el mètode \"i[3]\" val " + i[3]);
    modificarParametre(i);
    System.out.println("Després de cridar el mètode \"i[3]\" val " + i[3]);
  }
  //Té un únic paràmetre d'entrada, de tipus enter
  public void modificarParametre(int[] a) {
    //Ara hi ha una variable "a" declarada.
    //El seu valor depèn de com s'ha invocat el mètode.
    a[3] = 0;
    System.out.println("Hem modificat el valor a " + a[3]);
  }
}
```

El que ha succeït ara seria equivalent a haver fet:

```
//Es disposa d'una variable de tipus array
int i[] = {1, 2, 3, 4, 5};
//Es crea una còpia de la variable array
int a[] = i;
//Es modifica la còpia
a[3] = 0;
//Quant val i[3]? i[3] val 0!
```

De fet, aquesta circumstància es dóna amb qualsevol tipus de dada compost del Java. L'única excepció són les cadenes de text (variables de la classe `String`). El seu comportament és igual al dels tipus primitius, ja que, recordeu, tenen la propietat de ser immutables. No és possible modificar-ne el contingut.

Repte 2. Feu un programa anomenat `ModificarParametreString` que comprovi que, efectivament, modificar el valor d'un paràmetre d'entrada de la classe `String` no té cap efecte sobre el valor original.

2.2 Paràmetres de sortida

Com en el cas dels paràmetres d'entrada, es partirà d'una definició i a partir d'aquí veureu quina és la utilitat i mode d'ús d'aquest tipus de paràmetres.

> Un **paràmetre de sortida** indica un resultat final obtingut després de realitzar un procés determinat.

2.2.1 Motivació: Definició de problemes que generen un resultat concret

Si us fixeu una mica, us adonareu d'una característica comuna a molts dels casos en què un problema, en qualsevol nivell, es divideix en subproblemes més simples. Sovint, conceptualment, la manera com es resol és realitzant un seguit de tasques on s'obtenen resultats parcials, que al final són combinats o aprofitats dins el procés general per obtenir el resultat final (la resolució del problema).

Revisant la descomposició de la recepta de cuina, simplement en una observació a primer nivell, podreu detectar subproblemes on la seva conclusió repercuteix en l'obtenció d'un resultat parcial tangible, que és manipulat per subproblemes posteriors: diferents aliments cuinats preparats per separat. Partint d'aquests resultats parcials, s'obté el plat final, en combinar-los tots en les tasques associades al darrer subproblema. Aquesta circumstància queda resumida a la taula 2.1.

Taula 2.1. Subproblemes i resultats obtinguts en resoldre'ls

Subproblema	Què fa	resultat obtingut
1. Recopilar ingredients	Obté els ingredients base (tallarines, cebes, etc.)	Els diferents aliments a processar
2. Cuinar tallarines	Dels aliments, processa les tallarines	Tallarines preparades
3. Cuinar pastanagues	Dels aliments, processa les pastanagues	Pastanagues preparades
4. Cuinar cebes	Dels aliments, processa les cebes	Cebes preparades
5. Preparació final	A partir de tallarines, pastanagues i cebes, fa el plat final	Plat final

Més enllà d'una recepta de cuina, segur que podeu pensar en molts casos semblants on els subproblemes a resoldre en realitat són un mecanisme per recopilar un conjunt de resultats parcials, necessaris abans de poder dur a terme la resolució del problema general.

- Per escriure un llibre, cal recopilar-ne els capítols.
- Per enviar un missatge electrònic, cal obtenir l'adreça de destinació o

generar el text a enviar.

- Per muntar una màquina qualsevol, cal obtenir els seus components. Aquests, a la seva vegada, també es generen a partir del muntatge de peces encara més simples.
- Per vendre un producte, cal cercar el producte pròpiament, consultar-ne el preu o crear un tiquet.
- Etc.

Cada mètode ha de ser una eina especialitzada per fer una única tasca concreta. Font: Joan Arnedo

Per tant, en la definició d'un subproblema també pot cobrar sentit indicar que quan aquest es resol, es produeix un resultat concret que podrà ser usat *a posteriori* per altres subproblemes. D'acord a la definició inicial, aquest resultat seria precisament el seu paràmetre de sortida.

Teòricament, es podria preveure una tasca que, en acabar, produeixi més d'un resultat, ja que ha resolt diversos problemes alhora. Per simplificar, només es tractarà el cas de subproblemes que produeixen un únic resultat, i per tant, a molt estirar amb un únic paràmetre de sortida. Per tant, a partir d'ara, sempre que es faci la descomposició d'un problema, caldrà que la plantegeu de manera que això es compleixi. Partireu del principi que si un subproblema produeix més d'un resultat tangible, és que no s'ha descomposat prou encara.

Evidentment, també es pot donar el cas que les tasques associades a un subproblema no donin cap resultat tangible que hagi de ser manipulat *a posteriori*. En aquest cas, no hi haurà cap paràmetre de sortida.

2.2.2 Declaració i ús de mètodes amb un paràmetre de sortida

Novament, dins de la declaració d'un mètode també és possible fer una translació directa d'aquesta propietat per tal de codificar la resolució d'un subproblema. Parlant ja en termes de mètodes dins els vostres programes, un mètode amb un paràmetre de sortida explicita que servirà per generar un resultat, amb un valor concret.

Declaració

Atès que un programa genera i tracta dades, el primer que cal decidir és a quin tipus de dades pertanyerà el valor d'aquest resultat. Un cop decidit, la sintaxi per declarar un mètode amb un paràmetre de sortida és la següent:

```
1 public tipusParamSortida nomMètode (llistaParamEntrada) {
2   //Codi
3   ...
4 }
```

El tipus de paràmetre de sortida pot ser qualsevol, identificat per la paraula clau corresponent, com en el cas d'una variable. Si es vol indicar que no hi ha cap

paràmetre de sortida, s'usa la paraula reservada `void`, tal com heu usat fins ara en tots els mètodes abans d'introduir aquest concepte.

El codi del mètode serà el que correspongui per tal d'obtenir el resultat esperat, d'acord al seu propòsit. En aquest aspecte, res canvia respecte a com heu programat mètodes fins ara. Ara bé, dins d'aquest codi caldrà decidir, d'entre totes les dades que tracta internament, quina es correspon al resultat final. Un cop ho heu establert, la sentència que indica que ja heu acabat la tasca a fer i disposeu del resultat és la següent:

```
1 return resultat;
```

De fet, tan bon punt s'invoca la sentència `resultat`, l'execució del mètode acaba immediatament i ja no s'executa cap instrucció més.

Aquesta normalment s'escriuria la darrera de totes, al final del bloc de codi associat al mètode. On posa `resultat` es pot usar qualsevol expressió que avaluï el tipus declarat pel paràmetre de sortida: un literal, una variable, o una combinació qualsevol d'operands i operadors. Si es vol anar a poc a poc, normalment el més recomanable és que deseu el resultat en una variable i feu `return` d'aquesta (`return nomVariable;`).

Escriure la sentència `return` és indispensable, ja que si no es posa havent declarat un paràmetre de sortida, hi haurà un error de compilació.

> Donat el nom de la sentència usada, sovint al paràmetre de sortida se l'anomena també **valor de retorn** d'un mètode. A partir d'ara s'usarà també aquesta nomenclatura indistintament.

Evidentment, la millor manera de veure el funcionament de tot plegat és amb un exemple senzill. Suposeu que voleu definir un mètode que serveix per llegir correctament un únic valor enter des del teclat, de manera que ho faci definint un paràmetre de sortida. El seu codi podria ser el següent (Llegiu atentament els comentaris del codi, ja que indiquen el conjunt de decisions que cal anar fent per codificar-lo correctament.):

```
1  //1. Quin tipus de valor genera? Un enter (int)
2  public int llegirEnterTeclat() {
3    //2. Es fa el codi que llegeix un únic enter del teclat, com s'ha fet sempre
4    //No canvia absolutament res...
5    Scanner lector = new Scanner(System.in);
6    int enterLlegit = 0;
7    boolean llegit = false;
8    while (!llegit) {
9      llegit = lector.hasNextInt();
10     if (llegit) {
11       enterLlegit = lector.nextInt();
12     } else {
13       System.out.println("Això no és un enter.");
14       lector.next();
15     }
16   }
17   lector.nextLine();
18   //3. Un cop fet, quina variable té el resultat? "enterLlegit"
19   //4. Cal fer "return" damunt seu
20   return enterLlegit;
21 }
```

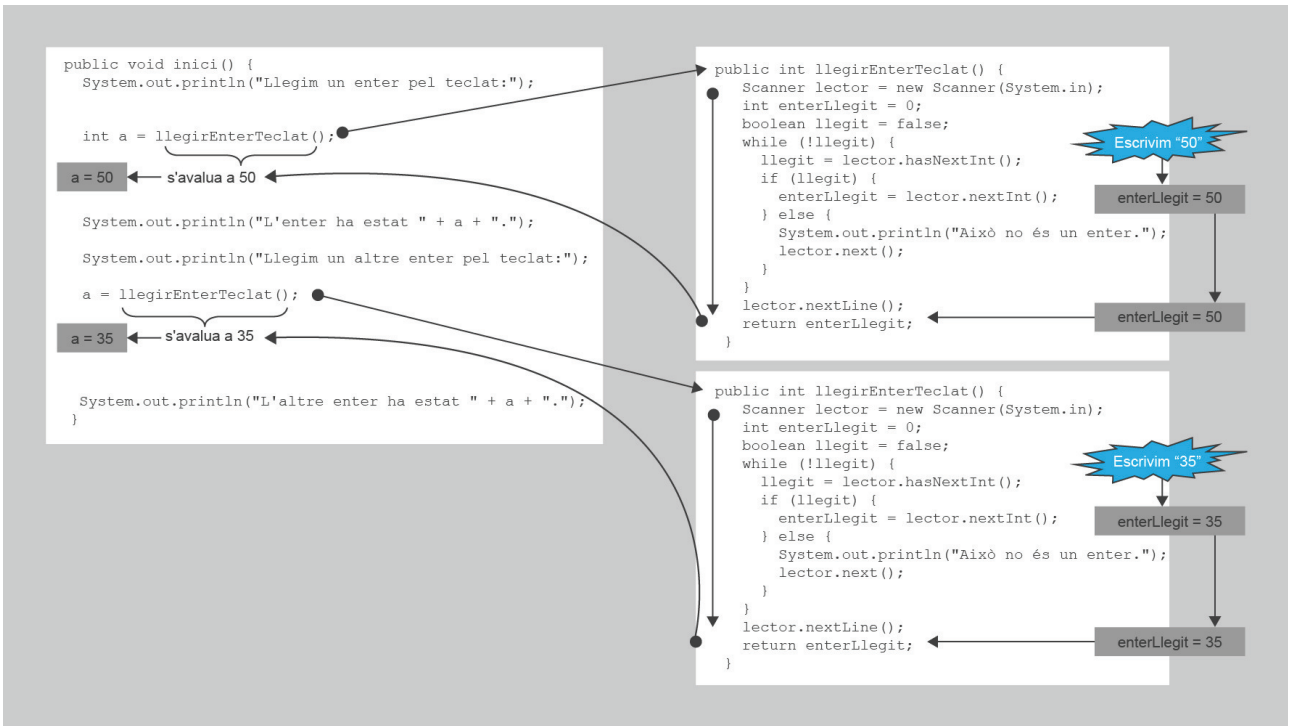
Invocació

Com en tota expressió, si voleu obtenir el resultat calculat pel mètode per consultar-lo *a posteriori*, caldrà que el guardeu en una variable.

L'aspecte més important quan hi ha mètodes amb un valor de retorn definit dins el vostre codi és la invocació, ja que hi ha una variació substancial respecte al model vist fins al moment. La sintaxi per invocar el mètode és exactament la mateixa, posant el nom i la llista de valors per als paràmetres d'entrada (si n'hi ha). En dur-se a terme la invocació, el flux d'execució salta també a les instruccions definides dins del seu bloc de codi. Ara bé, el que canvia és el comportament del mètode, ja que passa a considerar-se una expressió. Aquest avaluarà el valor resultant per a aquella invocació.

Per veure aquest procés, fixeu-vos en el codi següent, en el qual s'invoca el mètode definit anteriorment. Compileu-lo i executeu-lo, per veure què succeeix. La figura 2.3 mostra un esquema de quin és el flux de control associat a la primera invocació i què està succeint en cada moment. Per claredat a la figura, el codi del mètode `llegirEnterTeclat` està repetit per les diferents invocacions, però, novament, tingueu en compte que en el codi font només apareix una única vegada. Simplement serveix per mostrar que el seu resultat pot variar per diferents invocacions.

Figura 2.3. Flux de control en cridar un mètode i avaluació del mètode usant el valor de retorn.




```
1  import java.util.Scanner;
2  public class LlegirEnters {
3    public static void main (String[] args) {
4      LlegirEnters programa = new LlegirEnters();
5      programa.inici();
6    }
7    public void inici() {
8      System.out.println("Llegiu un enter pel teclat:");
9      int a = llegirEnterTeclat();
10     System.out.println("L'enter ha estat " + a + ".");
11     System.out.println("Llegiu un altre enter pel teclat:");
12     a = llegirEnterTeclat();
13     System.out.println("L'altre enter ha estat " + a + ".");
```

```
14    }
15    public int llegirEnterTeclat() {
16      //El codi no varia respecte a l'exemple anterior
17      //...
18    }
19  }
```

Si feu memòria, aquest model d'invocació és exactament igual a la invocació de mètodes sobre una variable de tipus `String`. Això és perquè els mètodes definits per `String` estan declarats amb valors de retorn.

Repte 3. Modifiqueu l'exemple de manera que en lloc d'usar enters, funcioni amb reals (`double`). Assegureu-vos del seu correcte funcionament introduint nombres amb decimals.

2.3 Quan declarar paràmetres d'entrada o sortida

Com s'ha anat mostrant, l'ús de paràmetres, tant d'entrada com de sortida, és especialment útil per tal de convertir problemes molt semblants en un mateix i fer que el procés de descomposició d'un programa sigui conceptualment similar al d'un problema d'àmbit general. Ara bé, en realitat, el seu ús no s'hauria de limitar només a aquest escenari, sinó que hauria de ser generalitzat a qualsevol mètode. De fet, l'ús de paràmetres d'entrada i sortida no es tracta d'un cas especial diferenciat en la declaració de paràmetres, sinó el més habitual quan s'usen dins del codi d'un programa.

D'altra banda, fins ara, les variables globals s'han presentat com a mecanisme per compartir o traspassar dades entre mètodes de diferents nivells de descomposició. En realitat, però, també és el més normal usar paràmetres com a mecanisme per a aquesta tasca, reemplaçant majoritàriament l'ús de variables globals. A partir d'ara, es considerarà que és millor evitar usar-les, excepte en casos molt concrets. Per tant, davant la pregunta "quan declarar paràmetres d'entrada i sortida en un mètode?", la resposta és "sempre que es pugui i tingui sentit", independentment que un mètode en realitat s'invoqui una sola vegada o moltes dins del vostre programa.

2.3.1 Millorant la llegibilitat de codi amb paràmetres

Una part de la utilitat dels mètodes és millorar la comprensió del codi dels programes, de manera que no cal anar seguint i entenent les instruccions en un bloc massa llarg, i d'una sola vegada. Si els identificadors dels mètodes s'han escollit amb criteri, el codi és més fàcil de seguir. Aquest seguiment encara és més senzill si s'usen paràmetres, ja que no deixen cap mena de dubte sobre les dades que està tractant un mètode donat.

Per exemple, fixeu-vos en el codi que es mostra a continuació i plantegeu-vos sincerament les preguntes següents:

- En aquesta invocació, quins valors s'estan tractant a `unMetode`?
- Quin és el resultat per a aquest cas?
- Si el volguéssiu mostrar per pantalla, en quina variable es troba?

```
1 //Són variables globals
2 i = 4;
3 j = 8;
4 k = 12;
5 ...
6 unMetode();
```

De fet, és impossible contestar les preguntes sense saber exactament com funciona aquest mètode. Hauríeu d'anar immediatament al seu codi font i inspeccionar-lo detalladament.

En canvi, compareu-lo amb aquest altre codi:

```
1 //No són variables globals
2 int i = 4;
3 int j = 8;
4 int k = 12;
5 ...
6 k = unMetode(i, j);
```

Sense ni tan sols saber què fa el mètode, no cal saber quin és el seu codi per respondre. Sense cap mena de dubte, els valors que s'estan tractant són 4 i 8, i el resultat estarà emmagatzemat a la variable `k`. Per tant, triar usar paràmetres en lloc de variables globals és especialment positiu per a la llegibilitat del codi.

2.3.2 El principi d'ocultació / encapsulació

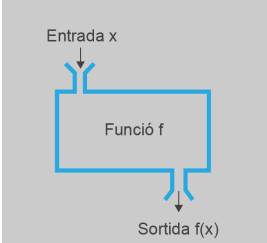
Un motiu per usar de manera generalitzada paràmetres d'entrada i sortida en els vostres mètodes i limitar l'ús de les variables globals és garantir un dels principis més importants dins de la programació.

> El principi d'**encapsulació / ocultació** es defineix formalment com la segregació dels elements en què es descompon un problema general, de manera que se separa la seva interfície contractual de la seva implementació.

Dit en altres paraules, un mètode ha de fer una única tasca molt concreta, de manera autocontinguda, i per poder ser invocat no cal saber realment *com* funciona, només *què* fa. Algú que no hagi programat el mètode ha de ser capaç d'usar-lo sense haver de saber quin és el seu codi o quines variables s'estan usant a dins seu. Si per poder invocar satisfactòriament un mètode es requereix saber

part del seu codi, no s'està seguint aquest principi. De fet, fins a cert punt, aquest principi està lligat als motius exposats anteriorment per evitar l'ús de variables globals.

Per tal de seguir fidelment el principi d'encapsulació / ocultació, pot ser útil establir unes pautes generals de comportament dels mètodes quan es volen considerar en forma parametritzada. L'aspecte principal per assolir aquesta fita és adonar-se que, tot i que a l'hora de declarar un mètode els paràmetres d'entrada i el de sortida són aspectes totalment independents sintàcticament, en la majoria de casos estan molt relacionats dins el seu codi (figura 2.4).



En darrera instància, els mètodes solen encapsular una transformació de les dades d'entrada en dades de sortida

Normalment, els mètodes són utilitzats com un mecanisme per encapsular blocs de codi autocontinguts que realitzen càlculs o transformacions sobre els valors dels paràmetres d'entrada, a partir dels quals generar un resultat final concret. No és casualitat que en molts llenguatges als mètodes se'ls anomeni "funcions", ja que es comporten exactament com funcions matemàtiques: partint d'uns valors d'entrada donats, s'aplica una fórmula (en aquest cas, un procés) mitjançant la qual se'n calcula el resultat.

Exemples de problemes típics que heu vist i que compleixen la condició que, donats uns valors d'entrada, cal calcular un resultat concret, poden ser:

- El màxim (o el mínim) entre un conjunt de valors (per exemple, un *array*).
- La mitjana aritmètica dels valors dins un *array*.
- Donat un valor de nota, transformar-la en cadena de text (per exemple, 5 = Aprovat).
- Saber els dies que té un mes donat.
- Cercar la posició d'un valor concret dins un *array* (si n'hi ha).

2.3.3 Exemples de mètodes parametritzats

A mode d'exemple, tot seguit es mostra la traducció d'alguns dels blocs de codi associats a problemes amb els quals heu treballat en mètodes amb paràmetres d'entrada i un valor de retorn. Dins del codi trobareu en format comentari les explicacions sobre quina mena de dades tracten, i per tant, quins són els seus paràmetres d'entrada i sortida.

En aquests exemples només es mostra el codi associat al mètode que realitza cada tasca, no el programa complet, amb els mètodes principal (`main`) i `inici`.

Calcular el mínim entre dos enters

```
1 //Paràm. entrada: els valors a tractar són dos enters
2   //Paràm. sortida: el mínim entre els dos, un enter
3   public int minim (int a, int b) {
```

```
    //"a" i "b" contenen els valors a tractar
    int min = b;
    if (a < b) {
      min = a;
    }
    //"min" conté el resultat
    return min;
  }
```

Per invocar-lo per calcular el mínim entre `4` i `10`, i desar el resultat en una variable anomenada `res`, es faria:

```
int res = minim(4, 10);
//En aquest cas, "res" valdrà 10
```

Repte 4. Usant aquest mètode, feu un programa que calculi quin és el valor mínim entre els valors 3, 6, 10 i 15, i el mostri per pantalla.

Calcular el màxim dins un array de reals

```
//Paràm. entrada: el valor a processar és un array
  //Paràm. sortida: el màxim entre tots, un real
  public double maxim(double[] array) {
    //"array" conté el valor a tractar
    double max = array[0];
    for (int i = 1; i < array.length; i++) {
      if (array[i] > max) {
        max = array[i];
      }
    }
    //"max" conté el resultat
    return max;
  }
```

Per invocar-lo per calcular el màxim a l'*array* anomenat `valors` i desar el resultat en una variable anomenada `res`, es faria:

```
double[] valors = {1.3, -2.1, 0, 12.0, 4.7};
...
int res = maxim(valors2);
//En aquest cas, "res" valdrà 12.0
```

Calcular la mitjana aritmètica dels valors d'un array

```
//Paràm. entrada: el valor a processar és un array
  //Paràm. sortida: el resultat de fer el càlcul, un real (té decimals)
  public double mitjana(int[] array) {
    //"array" conté el valor a tractar
    int acumulador = 0;
    for (int i = 0; i < array.length; i++) {
      acumulador = acumulador + array[i];
    }
    //El resultat és (acumulador)/(nombre elements)
    //Es pot retornar el resultat d'una expressió
    return acumulador/array.length;
  }
```

Per invocar-lo per calcular la mitjana de l'array `valors` i desar el valor a la variable `res`, es faria:

```
int[] valors = {1, 2, 3, 4, 5};
...
int res = mitjana(valors);
//En aquest cas, "res" valdrà 3
```

Transformar una nota a text

```
//Paràm. entrada: el valor a processar és un real (una nota)
  //Paràm. sortida: una cadena de text
  public String notaAText(double nota) {
    //"nota" conté el valor a tractar
    String text = null;
    if ((nota >= 9)&&(nota <= 10)) {
      text = "Excel·lent";
    } else if ((nota >= 6.5)&&(nota < 9)) {
      text = "Notable";
    } else if ((nota >= 5)&&(nota < 6.5)) {
      text = "Aprovat";
    } else  if ((nota >= 0)&&(nota < 5)) {
      text = "Suspès";
    } else {
      text = "Nota no vàlida";
    }
    //"text" conté el resultat
    return text;
  }
```

Per invocar-lo per saber el text que correspon a la nota `7.8`, desar el valor a la variable `textNota` i mostrar-lo per pantalla, es faria:

```
String textNota = notaAText(7.8);
//En aquest cas, "textNota" valdrà "Notable"
System.out.println(textNota);
```

Repte 5: feu un programa que, donat un conjunt de cinc notes parcials emmagatzemades en un *array* de reals, mostri el text de la nota final (la mitjana de totes cinc). Pista/Nota: tingueu en compte que el paràmetre d'entrada de l'exemple `mitjana` és un *array* d'enters. (No podeu usar el seu codi directament, l'haureu de modificar.)

Calcular els dies d'un mes

```
public static final int[] numDies = {31, 28 , 31, 30, 31, 30, 31, 31, 30, 31,
    30, 31};
  ...

  //Paràm. entrada: el valor a processar és un enter (un núm. de mes)
  //Paràm. sortida: un enter (un núm. de dies)
  public int diesDelMes(int mes) {
    //"mes" conté el valor a tractar
    int dies = 0;
    if ((mes >= 1)&&(mes >= 12)) {
      dies = numDies[mes - 1];
    }
    //"dies" conté el resultat
    return dies;
  }
```

Per invocar-lo per saber els dies del febrer i desar-los a la variable `res`, es faria:

```
int res = diesDelMes(2);
//En aquest cas, "res" valdrà 28
```

Cercar la posició on hi ha un valor concret a un array d'enters

```
//Paràm. entrada: els valors a processar són l'array i què cercar (un enter)
  //Paràm. sortida: un enter (una posició de l'array)
  public int cercarValor(int[] array, int valor) {
    //"array" conté el valor a tractar
    int posicio = 0;
    boolean trobat = false;
    while ((posicio < array.length)&&(!trobat)) {
      if (array[posicio] == valor) {
        trobat = true;
      }
      posicio = posicio + 1;
    }
    //S'ha trobat?
    if (!trobat) {
      posicio = -1;
    }
    //"posició" conté el resultat.
    //Si no s'ha trobat es fa valer -1
    return posicio;
  }
```

Per invocar-lo per trobar si en alguna posició d'un *array* anomenat `valors` hi ha el valor 7 i desar l'índex d'aquesta posició a la variable `res`, es faria:

```
int[] valors = {1, 2, 3, 4, 5};
...
int res = cercarValor(valors, 7);
//En aquest cas, "res" valdrà -1
```

Repte 6. Feu un programa que pregunti un número enter pel teclat. Llavors, donat un conjunt de valors dins un *array*, ha de calcular quants cops apareix el número introduït. Useu mètodes tant per preguntar el número com per cercar el seu nombre d'aparicions. Declareu l'*array* on cal fer la cerca com una variable dins del mètode `inici`, inicialitzada amb valors concrets de la vostra elecció.

2.4 Un exemple de disseny descendent amb mètodes parametritzats

Per saber com usar mètodes basats en paràmetres d'entrada i sortida (el més habitual quan feu programes), tot seguit veureu un nou exemple de procés complet de descomposició d'un problema relativament complex mitjançant disseny descendent. Aquest cop, però, els subproblemes s'analitzaran també des del punt de vista de quines dades són necessàries per a la seva resolució (entrada) i quin resultat concret produeixen (sortida). Aquest cop, però, no es detallaran tots els aspectes de la descomposició. Aquest exemple serveix principalment com a fil argumental per ressaltar alguns aspectes importants de l'ús de mètodes dins un

programa més complex, com la propagació dels valors entre invocacions dels mètodes.

El sistema de joc d'endevinar un número secret és semblant al del Mastermind.

En aquest cas, el programa que es vol realitzar és una versió una mica més avançada del joc d'endevinar un número secret. En aquest cas, es tracta d'endevinar una combinació de 5 lletres a l'atzar. Sempre que es realitza un intent, el programa dóna una pista. Per cada lletra, s'indica si és totalment incorrecta (`.`), si s'ha encertat (`O`), o si aquesta lletra existeix, però no en aquesta posició (`X`). El programa s'executa fins que l'usuari encerta, moment en què es dóna un missatge de felicitació.

Per aclarir aquest funcionament, tot seguit es dóna un exemple d'execució:

```
Escriu 5 lletres minúscules: atzbq
La resposta es [-OX--]. Continua intentant-ho!
Escriu 5 lletres minúscules: btymz
La resposta es [-O-XX]. Continua intentant-ho!
Escriu 5 lletres minúscules: mtzoc
La resposta es [OOX--]. Continua intentant-ho!
Escriu 5 lletres minúscules: mtdzp
La resposta es [OO-OX]. Continua intentant-ho!
Escriu 5 lletres minúscules: mtpze
La resposta es [OOOO-]. Continua intentant-ho!
Escriu 5 lletres minúscules: mtpzx
La resposta es [OOOOO]. Has encertat!
```

2.4.1 Descomposició de problema

Novament, el primer pas a fer, abans de posar-se a generar codi, és seure una estona a pensar i fer una proposta sobre quines dades caldrà tractar i com es podria descompondre el problema.

Identificació de les dades a tractar

En una ullada preliminar, per a aquest problema, les dades més importants que cal tractar en general són bàsicament dues: la paraula secreta i la paraula que respon l'usuari. També, en algun moment, caldrà generar la resposta del programa amb la pista dient quines lletres ha encertat i controlar quan cal acabar l'execució, un cop s'ha encertat.

Primer nivell de descomposició

Abans de poder fer res, caldrà generar la paraula secreta de manera aleatòria. Un cop fet, el programa es dedica a preguntar la resposta de l'usuari i tot seguit analitzar-la per generar una resposta amb la pista, o la felicitació si ha encertat. Per tant, en un primer nivell es podria descompondre com:

1. Generar paraula secreta.

2. Preguntar resposta.
3. Resoldre resposta.

Segon nivell de descomposició

Per veure si cal un segon nivell de descomposició, cal analitzar si els subproblemes definits en el primer nivell es consideren complexos, o si més no, si conceptualment són més fàcils d'entendre si es tornen a descompondre.

- **Generar paraula secreta**. Generar una paraula secreta es pot considerar com la generació repetides vegades d'una única lletra aleatòria.
- **Preguntar resposta**. D'entrada, llegir una cadena de text des del teclat és un problema àmpliament tractat i que ja hauríeu de saber resoldre. Per tant, no és gens complex.
- **Resoldre resposta**. Resoldre la resposta vol dir que, primer de tot, cal generar la pista i després veure si es correspon amb una combinació guanyadora (`OOOOO`) o no. Per tant, es podria dividir en aquests dos subproblemes.

1. Generar paraula secreta.
   (a) Generar lletra aleatòria.
   (b) Muntar paraula.
2. Preguntar resposta.
3. Resoldre resposta.
   (a) Generar pista.
   (b) Donar resposta.

Tercer nivell de descomposició

"System.nanoTime"()
Aquesta és la instrucció que accedeix al temps amb precisió de nanosegons (una milmilionèsima de segon, un segon = 1.000.000.000 nanosegons).

Novament, s'analitzen els nous subproblemes per veure si cal continuar descomponent o no.

- **Generar lletra aleatòria**. Aquest subproblema requereix una mica d'imaginació per plantejar-lo (i una mica de coneixement de quines instruccions té Java que us puguin ajudar). Per tant, segons com es plantegi, serà fàcil o difícil. En aquest exemple es farà de la manera següent. Java permet accedir a l'hora actual amb precisió de nanosegons usant una instrucció especial. Això és un temps tan reduït que sempre s'obtindran valors diferents per cada execució. Per tant, és com generar valors a l'atzar. Atès que ja sabeu com, partint d'un enter, obtenir text (per exemple, en convertir un número de mes en el seu nom), ja s'ha arribat a un problema prou simple.

- **Muntar paraula**. Es tractaria de concatenar 5 lletres per crear un `String`. Aquesta tasca es pot resoldre directament amb l'operador `+` de les cadenes de text.

- **Generar pista**. Caldrà comparar la paraula secreta i la resposta lletra a lletra. Això es pot fer amb una estructura de repetició, que no és massa complexa, i les lletres (els caràcters) es poden comparar fàcilment.

- **Donar resposta**. Es tracta de fer una comparació i mostrar un missatge per pantalla. És clarament una estructura de selecció, la qual cosa és senzilla.

Per tant, ja heu acabat la descomposició.

2.4.2 Implementació del segon nivell de descomposició

Un cop establerta la descomposició a partir de la qual treballar, ja es podria establir una primera versió de l'esquelet de la classe. Per fer això, ara seria necessària una etapa de reflexió addicional, ja que per establir les capçaleres dels mètodes cal tenir ben clars quins seran els seus paràmetres d'entrada i de sortida. Malauradament, això obliga fins a cert punt a tenir una visió general de com ha de funcionar tot el programa i les interaccions entre mètodes.

A causa d'això, en un cas com aquest pot ser millor anar a poc a poc i només aturar-se a pensar la declaració exacta de cada mètode amb els seus paràmetres quan correspongui, seguint pas per pas cada nivell de descomposició. A part d'això, sí que és possible escriure ja el mètodes `main` i l'esquelet del mètode `inici`, ja que sempre segueixen la mateixa estructura.

Normalment, els mètodes que fan càlculs exclusivament a partir d'entrada / sortida (teclat, rellotge intern, etc.) no tenen paràmetres d'entrada.

Donada la descomposició del problema, cal declarar dos mètodes, que es poden dir:

- `generarLletraAleatoria`. Genera una lletra a l'atzar. Per tant, el valor de retorn serà un caràcter. No necessita cap entrada per fer aquest càlcul, ja que tot el que cal s'obté directament d'una instrucció de Java per saber els nanosegons presents.

- `generarPista`. Genera la pista que cal mostrar per pantalla (per exemple, `OXX.O`). Això és un text, per la qual cosa el valor de retorn serà un `String`. Per poder fer aquesta tasca, cal comparar la resposta de l'usuari (un text) i el valor secret (un altre text). Per tant, calen dos paràmetres d'entrada, també de tipus `String`. En el codi d'aquest mètode de ben segur caldrà manipular les cadenes de text.

Donat això, ara mateix, el codi del programa podria ser així. Fixeu-vos atentament que la traducció d'un enter aleatori a un caràcter aleatori es fa amb l'ajut d'una taula de traducció en forma de text amb tot l'alfabet (constant `abc`):

```java
import java.util.Scanner;
//Un programa per jugar al Mastermind.
public class MasterMind {
  //Constants
  public final static char TOT_CORRECTE = 'O';
  public final static char MALA_POSICIO = 'X';
  public final static char INCORRECTE = '-';
  public final static String abc = "abcdefghijklmnopqrstuvwxyz";
  //Mètodes associats al problema general
  //El mètode main no canvia respecte a la plantilla general
  public static void main (String[] args) {
    MasterMind programa = new MasterMind();
    programa.inici();
  }
  //Es pot declarar, però encara no correspon deduir el codi
  public void inici() {
    //El codi es farà en acabar el primer nivell de descomposició
    //...
  }
  //Mètodes associats al primer nivell de descomposició
  //Ja els pensareu en acabar el segon nivell de descomposició
  //...
  //Mètodes associats al segon nivell de descomposició
  //Paràm. entrada: cap
  //Paràm. sortida: una lletra (un caràcter)
  public char generarLletraAleatoria() {
    long nano = System.nanoTime();
    int index = (int)(nano % abc.length());
    return abc.charAt(index);
  }
  //Paràm. entrada: la resposta i el secret que cal comparar (text)
  //Paràm. sortida: la pista que cal mostrar (un text)
  public String generarPista(String s, String r) {
    String res = "";
    //Cal comparar cada caràcter de la solució i la resposta
    for (int i = 0; i < s.length(); i++) {
      //Obtenir els dos caràcters a comparar
      char charSecret = s.charAt(i);
      char charResposta = r.charAt(i);
      if (charSecret == charResposta) {
        res = res + TOT_CORRECTE;
      } else if (s.indexOf(charResposta) != -1) {
        //indexOf cerca si un caràcter concret existeix en un String
        res = res + MALA_POSICIO;
      } else {
        res = res + INCORRECTE;
      }
    }
    return res;
  }
}
```

2.4.3 Implementació del primer nivell de descomposició

Un cop finalitzat un nivell, es pot pujar al següent. D'acord a la descomposició del problema general, es pot procedir a establir els noms dels nous mètodes i quins haurien de ser els seus paràmetres. En aquest cas, però, per a alguns dels mètodes val la pena estudiar la seva implementació de manera molt més detallada, ja que aquest exemple ha estat cercat amb la intenció que apareguin alguns aspectes interessants quan interactuen mètodes amb paràmetres d'entrada i/o sortida.

D'entrada, aquí teniu una descripció general:

- `generarParaulaSecreta`. Genera la paraula secreta, composta per 5 lletres. Per tant, el resultat és un text (un valor de tipus `String`). Com que parteix de zero per fer-ho, no li calen paràmetres d'entrada.
- `preguntarResposta`. Com en exemples anteriors, haurà de ser un mètode que llegeix un text des del teclat i el retorna. Per tant, tampoc li calen paràmetres d'entrada i retornarà el text llegit.
- `resoldreResposta`. Resoldre una resposta vol dir generar la pista, establir si s'ha encertat, i mostrar un missatge en conseqüència. Quan es tracti amb més detall ja veureu quins haurien de ser els seus paràmetres.

generarParaulaSecreta

Aquest mètode es basa en l'ús repetit de la generació d'un caràcter individual per fer una paraula de 5 lletres. Per tant, invoca internament a `generarLletraAleatoria` i a partir de cada resultat crea el seu propi resultat. A part d'això, el fet que un mètode pugui invocar, a la seva vegada, un altre mètode, no té cap altra peculiaritat.

```
1  //Paràm. entrada: cap, tot s'obté de generarLletraAleatoria
2    //Paràm. sortida: una paraula de 5 lletres (un String)
3    public String generarParaulaSecreta() {
4      String res = "";
5      for (int i = 0; i < 5; i++) {
6        res = res + generarLletraAleatoria();
7      }
8      return res;
9    }
```

preguntarResposta

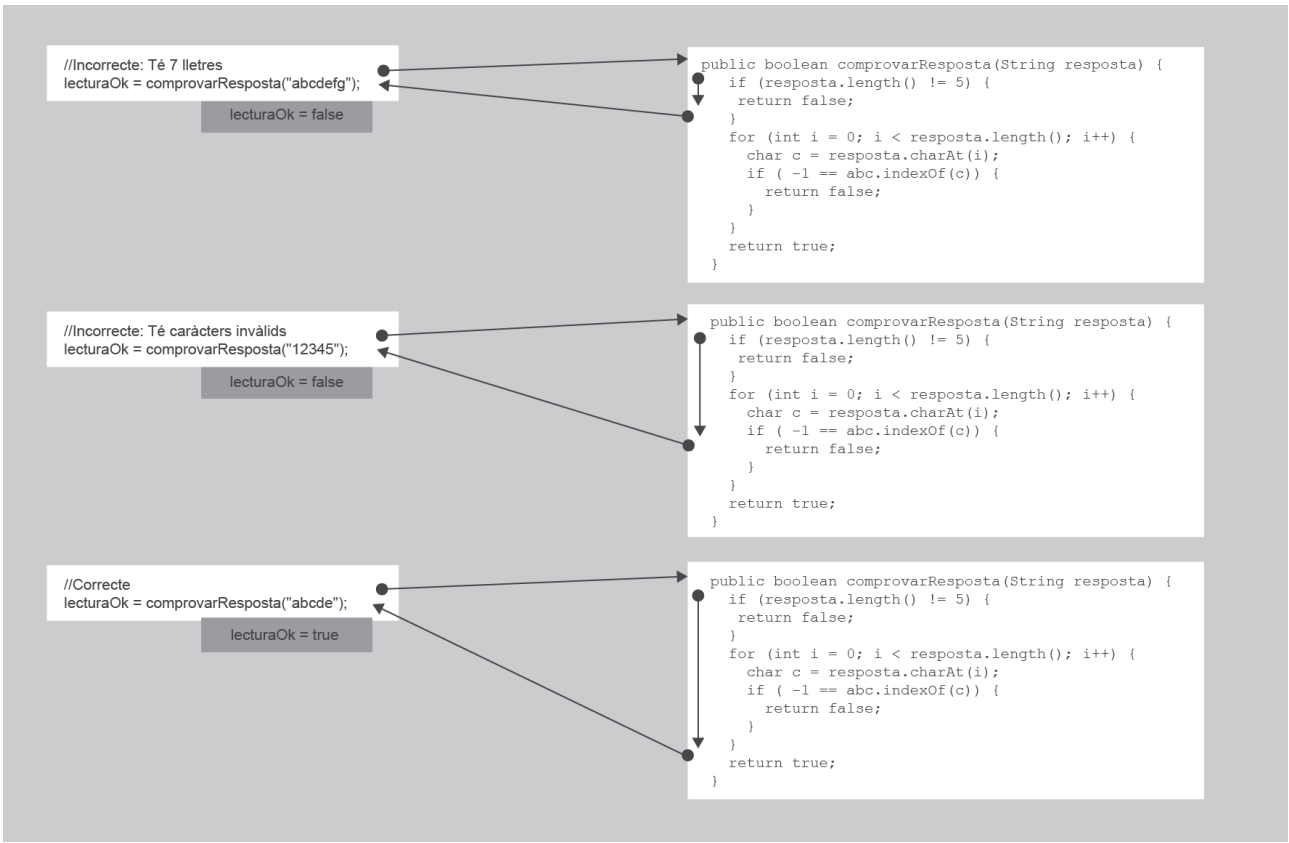
Escriure text demanant a l'usuari que introdueixi una paraula pel teclat i llegir-la en forma de cadena de text no és excessivament complicat, sobretot, atès que no cal fer comprovacions de tipus, com passa amb números (`hasNextInt`, etc.). Ara bé, si es vol ser estricte, caldria comprovar almenys que la resposta segueix el format esperat: té cinc lletres i està formada de caràcters de l'alfabet en minúscula (a...z). Ara bé, comprovar aquestes dues coses no es fa en una o dues línies de codi.

En casos com aquest, quan es vol fer una comprovació de certa complexitat sobre el format d'una lectura de dades escrites per l'usuari, és especialment útil plantejar-se-la com un nou subproblema. Separar tot el codi associat a la comprovació en un mètode a part simplifica el codi i millora la seva comprensió. En aquest cas, el nou mètode es podria dir `comprovarResposta`.

```
1  //Paràm. entrada: cap, tot s'obté del teclat
2    //Paràm. sortida: la paraula de resposta (un String)
3    public String preguntarResposta() {
4      Scanner lector = new Scanner(System.in);
5      boolean lecturaOk = false;
```

```
6     String res = null;
7     do {
8       System.out.print("Escriu " + LONG_SECRET + " lletres minúscules: ");
9       res = lector.next();
10      lector.nextLine();
11      lecturaOk = comprovarResposta(res);
12      if (!lecturaOk) {
13        System.out.println("Aquesta resposta no és vàlida!");
14      }
15    } while (!lecturaOk);
16    return res;
17  }
18  //Aquest mètode correspon al segon nivell de descomposició
19  //Paràm. entrada: text a comprovar
20  //Paràm. sortida: si és correcte o no (un booleà. true = correcte)
21  public boolean comprovarResposta(String resposta) {
22    if (resposta.length() != 5) {
23     //Ja sabem que no és correcte.
24     //Podem acabar l'execució del mètode immediatament.
25     return false;
26    }
27    for (int i = 0; i < resposta.length(); i++) {
28      char c = resposta.charAt(i);
29      if ( -1 == abc.indexOf(c)) {
30       //Ja sabem que no és correcte.
31       //Podem acabar l'execució del mètode immediatament.
32        return false;
33      }
34    }
35    //Si tot es compleix, segur que és correcte
36    return true;
37  }
```

Figura 2.4. Flux de control del mètode "comprovarResposta" davant diferents situacions



Aquesta mena de mètodes que serveixen per establir si un valor es considera correcte pràcticament sempre tenen el mateix format: l'entrada són els valors a

comprovar i la sortida, un booleà. Cal fer que aquesta sigui `true` si els valors es consideren correctes i `false` en cas contrari ("És el valor correcte? Cert/Fals"). Llavors, amb una estructura de selecció que compari el valor de retorn és fàcil actuar en conseqüència.

Per tant, tot just s'acaba de presentar una situació on, tot i que d'entrada s'ha considerat que no calia descompondre més, *a posteriori* s'ha establert que és millor fer-ho. Aquest fet no és gaire estrany en el procés de generació d'un programa.

La tècnica `comprovarResposta` també és habitual i us pot resultar molt útil per simplificar el codi dels vostres programes.

Si us fixeu, el codi de `comprovarResposta` té una peculiaritat. Hi ha més d'una sentència `return`. Això és possible, però per entendre què està fent, cal tenir present un aspecte essencial d'aquesta sentència: quan s'executa, el mètode acaba immediatament i el valor de retorn és igual a l'indicat dins la sentència `return`. La figura 2.4 esquematitza el funcionament del mètode d'acord a aquest comportament, segons diferents situacions.

resoldreResposta

Aquest mètode el primer que fa és invocar `generarPista` i mostrar el valor retornat. Ara bé, segons si aquest valor indica que si s'ha encertat (`OOOOO`) o no, actua en conseqüència. El que fa que valgui la pena estudiar-lo amb detall és veure quins paràmetres té.

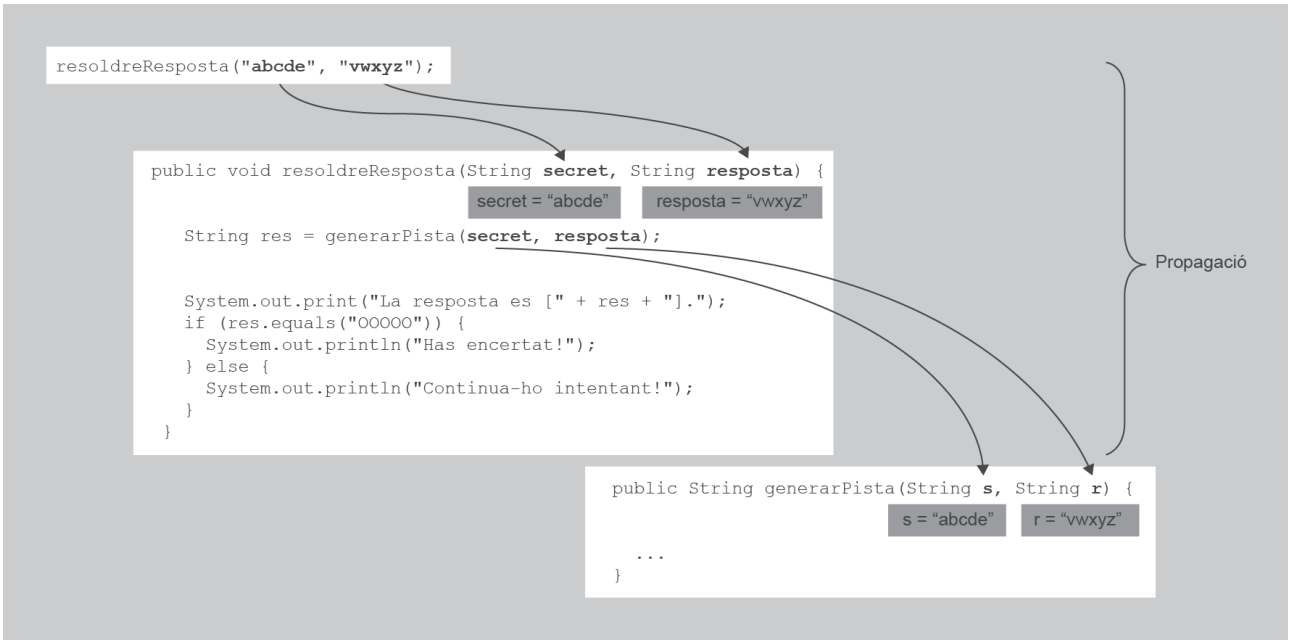
D'entrada només mostra coses per pantalla, no calcula res, per la qual cosa no cal cap valor de retorn. Ara bé, per poder invocar correctament `generarPista`, li cal disposar dels valors del text secret i la resposta de l'usuari. Aquests dos valors no els genera ell, sinó que són generats per altres mètodes. Per tant, per poder-ne disposar, li faran falta com paràmetres d'entrada.

```
1  //Paràm. entrada: la resposta i el valor secret (dos String)
2    //Paràm. sortida: cap (només mostra coses per pantalla)
3    public void resoldreResposta(String secret, String resposta) {
4      String res = generarPista(secret, resposta);
5      System.out.print("La resposta es [" + res + "].");
6      if (res.equals("OOOOO")) {
7        System.out.println("Has encertat!");
8      } else {
9        System.out.println("Continua intentant-ho!");
10     }
11   }
```

En aquest mètode és interessant observar que, internament, els dos paràmetres no són usats mai directament. Només serveixen per poder invocar correctament el mètode `generarPista`. Aquesta situació és la que s'anomena **propagació de paràmetres** entre mètodes ubicats en diferents nivells de descomposició. Els valors disposats en el mètode de més nivell es passen (propaguen) a mètodes de nivell inferior. Atès que un paràmetre d'entrada es considera com una variable qualsevol dins el codi d'un mètode, no hi ha cap problema a fer-ho.

La figura 2.5 esquematitza aquest procés. També és molt important veure com els valors es van copiant successivament a les variables definides pels paràmetres, independentment del fet que el seu nom vagi variant en cada cas.

Figura 2.5. Propagació dels valors dels paràmetres entre dos mètodes



2.4.4 Implementació del problema general

Un cop implementats tots els subproblemes, ha arribat el moment d'implementar el codi del problema general: el mètode `inici`. Primer de tot el que farà el programa és generar el valor secret, i després anar repetint la pregunta a l'usuari en mostrar la resposta. Això es pot fer usant una estructura de repetició controlada per un semàfor, que inicialment està a `false` i canvia a `true` quan l'usuari ha encertat. En una primera aproximació, el codi seria:

```
1  //Problema general: Versió 1
2    public void inici() {
3      String secret = generarParaulaSecreta();
4      //"encertat" diu si s'ha encertat i ja s'acaba el programa
5      boolean encertat = false;
6      while (!encertat) {
7        String resposta = preguntarResposta();
8        resoldreResposta(secret, resposta);
9      }
10   }
```

Ara bé, en aquest cas, el problema general també presenta una situació interessant. Sense variables globals, com es pot establir el valor de la variable `encertat` a `true`? Atès que és millor no usar variables globals, la resposta ha d'estar per força en el paràmetre d'entrada o sortida d'algun mètode. En un cas com aquest, on un mètode necessita un valor per poder dur a terme la seva tasca, la resposta exacta està en afegir un paràmetre de sortida a algun mètode.

Si observeu el codi, el mètode que estableix si la partida ha acabat (s'ha encertat o no) és `resoldreResposta`, ja que és qui compara la pista amb el valor associat a encertar la paraula secreta. D'entrada, aquest mètode no té cap valor de retorn. El que es pot fer és afegir un valor de retorn que us digui si cal acabar la partida o

no (un booleà). D'aquesta manera, el codi seria:

```
//Problema general: Versió 2
  public void inici() {
    String secret = generarParaulaSecreta();
    boolean encertat = false;
    while (!encertat) {
      String resposta = preguntarResposta();
      encertat = resoldreResposta(secret, resposta);
    }
  }
  //Aquest mètode correspon al primer nivell de descomposició
  //Versió 2
  //Paràm. entrada: la resposta i el valor secret (dos String)
  //Paràm. sortida: si s'ha encertat (un booleà)
  public boolean resoldreResposta(String secret, String resposta) {
    String res = generarPista(secret, resposta);
    boolean fi = false;
    System.out.print("La resposta és [" + res + "].");
    if (res.equals(ENCERTAT)) {
      System.out.println("Has encertat!");
      fi = true;
    } else {
      System.out.println("Continua intentant-ho!");
    }
    return fi;
  }
```

Aquest és un cas on cal establir paràmetres d'un mètode *a posteriori*, davant d'una necessitat imprevista d'un mètode de nivell superior. Tampoc es tracta d'una situació infreqüent.

> El procés de descomposició en nivell descendent no sempre és possible realitzar-lo linealment. Sovint cal recular i reconsiderar decisions preses anteriorment.

2.4.5 Implementació final

Per claredat, tot seguit es llista el resultat final. Per fer el codi més polit i llegible, alguns valors concrets s'han reemplaçat per constants (la longitud del text a descobrir i el text que indica que tot és correcte).

```
import java.util.Scanner;
//Un programa per jugar al Mastermind.
public class MasterMind {
  //Constants
  public final static char TOT_CORRECTE = 'O';
  public final static char MALA_POSICIO = 'X';
  public final static char INCORRECTE = '-';
  public final static String ENCERTAT = "OOOOO";
  public final int LONG_SECRET = 5;
  public final static String abc = "abcdefghijklmnopqrstuvwxyz";
 //Mètodes associats al problema general
  public static void main (String[] args) {
    MasterMind programa = new MasterMind();
    programa.inici();
  }
  public void inici() {
```

```
17      String secret = generarParaulaSecreta();
18      boolean encertat = false;
19      while (!encertat) {
20        String resposta = preguntarResposta();
21        encertat = resoldreResposta(secret, resposta);
22      }
23    }
24    //Mètodes associats al primer nivell de descomposició
25    //Paràm. entrada: cap, tot s'obté de generarLletraAleatoria
26    //Paràm. sortida: una paraula de 5 lletres (un String)
27    public String generarParaulaSecreta() {
28      String res = "";
29      for (int i = 0; i < LONG_SECRET; i++) {
30        res = res + generarLletraAleatoria();
31      }
32      return res;
33    }
34    //Paràm. entrada: cap, tot s'obté del teclat
35    //Paràm. sortida: la paraula de resposta (un String)
36    public String preguntarResposta() {
37      Scanner lector = new Scanner(System.in);
38      boolean lecturaOk = false;
39      String res = null;
40      do {
41        System.out.print("Escriu " + LONG_SECRET + " lletres minúscules: ");
42        res = lector.next();
43        lector.nextLine();
44        lecturaOk = comprovarResposta(res);
45        if (!lecturaOk) {
46          System.out.println("Aquesta resposta no és vàlida!");
47        }
48      } while (!lecturaOk);
49      return res;
50    }
51    //Paràm. entrada: la resposta i el valor secret (dos String)
52    //Paràm. sortida: si s'ha encertat (un booleà)
53    public boolean resoldreResposta(String secret, String resposta) {
54      String res = generarPista(secret, resposta);
55      boolean fi = false;
56      System.out.print("La resposta és [" + res + "].");
57      if (res.equals(ENCERTAT)) {
58        System.out.println("Has encertat!");
59        fi = true;
60      } else {
61        System.out.println("Continua intentant-ho!");
62      }
63      return fi;
64    }
65    //Mètodes associats al segon nivell de descomposició
66    //Paràm. entrada: cap
67    //Paràm. sortida: una lletra (un caràcter)
68    public char generarLletraAleatoria() {
69      long nano = System.nanoTime();
70      int index = (int)(nano % abc.length());
71      return abc.charAt(index);
72    }
73    //Paràm. entrada: text a comprovar
74    //Paràm. sortida: si és correcte o no (un booleà. true = correcte)
75    public boolean comprovarResposta(String resposta) {
76      if (resposta.length() != LONG_SECRET) {
77       //Ja sabem que no és correcte.
78       //Podem acabar l'execució del mètode immediatament.
79       return false;
80      }
81      for (int i = 0; i < resposta.length(); i++) {
82        char c = resposta.charAt(i);
83        if ( -1 == abc.indexOf(c)) {
84         //Ja sabem que no és correcte.
85         //Podem acabar l'execució del mètode immediatament.
86          return false;
```

```
87        }
88      }
89      //Si tot es compleix, segur que és correcte
90      return true;
91    }
92    //Paràm. entrada: la resposta i el secret que cal comparar (text)
93    //Paràm. sortida: la pista que cal mostrar (un text)
94    public String generarPista(String s, String r) {
95      String res = "";
96      for (int i = 0; i < s.length(); i++) {
97        char charSecret = s.charAt(i);
98        char charResposta = r.charAt(i);
99        if (charSecret == charResposta) {
100         res = res + TOT_CORRECTE;
101       } else if (s.indexOf(charResposta) != -1) {
102         res = res + MALA_POSICIO;
103       } else {
104         res = res + INCORRECTE;
105       }
106     }
107     return res;
108   }
109 }
```

2.5 Solucions als reptes proposats

Repte 1

```java
public class EscriureEstrelles {
  public static void main (String[] args) {
    EscriureEstrelles programa = new EscriureEstrelles();
    programa.inici();
  }
  public void inici() {
    estrellar(4);
    estrellar(10);
    estrellar(20);
  }
  //Té un paràmetre d'entrada, de tipus enter
  public void estrellar(int a) {
    for(int i = 0; i < a; i++) {
      System.out.print("*");
    }
    System.out.println();
  }
}
```

Repte 2

```java
//Modifiquem el valor d'un paràmetre. Afecta a la variable original?
public class ModificarParametreString {
  public static void main (String[] args) {
    ModificarParametreString programa = new ModificarParametreString();
    programa.inici();
  }
  public void inici() {
    String i = "Hola";
    System.out.println("Abans de cridar el mètode \"i\" val " + i);
    modificarParametre(i);
    System.out.println("Després de cridar el mètode \"i\" val " + i);
  }
  //Té un únic paràmetre d'entrada, de tipus enter
  public void modificarParametre(String a) {
    //Ara hi ha una variable "a" declarada.
    //El seu valor depèn de com s'ha invocat el mètode.
    a = "Adeu";
    System.out.println("Hem modificat el valor a " + a);
  }
}
```

Repte 3

```java
import java.util.Scanner;
public class LlegirReals {
  public static void main (String[] args) {
    LlegirReals programa = new LlegirReals();
    programa.inici();
  }
  public void inici() {
    System.out.println("Llegim un real per al teclat:");
    double a = llegirRealTeclat();
    System.out.println("El real ha estat " + a + ".");
    System.out.println("Llegim un altre real per al teclat:");
    a = llegirRealTeclat();
    System.out.println("L'altre real ha estat " + a + ".");
  }
  public double llegirRealTeclat() {
    Scanner lector = new Scanner(System.in);
    double realLlegit = 0;
    boolean llegit = false;
    while (!llegit) {
      llegit = lector.hasNextDouble();
      if (llegit) {
        realLlegit = lector.nextDouble();
      } else {
        System.out.println("Això no és un enter.");
        lector.next();
      }
    }
    lector.nextLine();
    return realLlegit;
  }
}
```

Repte 4

```java
public class MultipleMaxim {
  public static void main (String[] args) {
    MultipleMaxim programa = new MultipleMaxim();
    programa.inici();
  }
  public void inici() {
    int primerMinim = minim(3, 6);
    int segonMinim = minim(10, 15);
    int resMinim = minim(primerMinim, segonMinim);
    System.out.println("El mínim és " + resMinim);
  }
  //Paràm. entrada: els valors a tractar són dos enters
  //Paràm. sortida: el mínim entre els dos, un enter
  public int minim (int a, int b) {
    //"a" i "b" contenen els valors a tractar
    int min = b;
    if (a < b) {
      min = a;
    }
    //"min" conté el resultat
    return min;
  }
}
```

Repte 5

```java
public class NotaFinal {
  public static final double[] notes = {5.0, 7.5, 4.5, 3.75, 8.25};
  public static void main (String[] args) {
    NotaFinal programa = new NotaFinal();
    programa.inici();
  }
  public void inici() {
    double res = mitjana(notes);
    String notaFinal = notaAText(res);
    System.out.println("La nota final és " + notaFinal);
  }
  //Paràm. entrada: el valor a processar és un array de reals
  //Paràm. sortida: el resultat de fer el càlcul, un real (té decimals)
  public double mitjana(double[] array) {
    //"array" conté el valor a tractar
    double acumulador = 0;
    for (int i = 0; i < array.length; i++) {
      acumulador = acumulador + array[i];
    }
    //El resultat és (acumulador)/(nombre elements)
    //Es pot retornar el resultat d'una expressió
    return acumulador/array.length;
  }
  //Paràm. entrada: el valor a processar és un real (una nota)
  //Paràm. sortida: una cadena de text
  public String notaAText(double nota) {
    //"nota" conté el valor a tractar
    String text = null;
    if ((nota >= 9)&&(nota <= 10)) {
      text = "Excel·lent";
    } else if ((nota >= 6.5)&&(nota < 9)) {
      text = "Notable";
    } else if ((nota >= 5)&&(nota < 6.5)) {
      text = "Aprovat";
    } else  if ((nota >= 0)&&(nota < 5)) {
      text = "Suspès";
    } else {
      text = "Nota no vàlida";
    }
    //"text" conté el resultat
    return text;
  }
}
```

Repte 6

```java
import java.util.Scanner;
public class ComptaValors {
  public static void main (String[] args) {
    ComptaValors programa = new ComptaValors();
    programa.inici();
  }
  public void inici() {
    int[] valors = {1, 3, 7, 8, 5, 1, 3, 1, 5, 4};
    System.out.println("Introdueix un número enter:");
    int valor = llegirEnterTeclat();
    int res = comptaValors(valors, valor);
    System.out.println("El valor " + valor + " apareix " + res + " cops.");
  }
  public int llegirEnterTeclat() {
    Scanner lector = new Scanner(System.in);
    int enterLlegit = 0;
    boolean llegit = false;
    while (!llegit) {
      llegit = lector.hasNextInt();
      if (llegit) {
        enterLlegit = lector.nextInt();
      } else {
        System.out.println("Això no és un enter.");
        lector.next();
      }
    }
    lector.nextLine();
    return enterLlegit;
  }
  public int comptaValors(int[] array, int valor) {
    int acumulador = 0;
    for (int i = 0; i < array.length; i++) {
      if (array[i] == valor) {
       acumulador++;
      }
    }
    return acumulador;
  }
}
```