Programació modular

Joan Arnedo Moreno

Programació bàsica (ASX)
Programació (DAM)

Índex

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

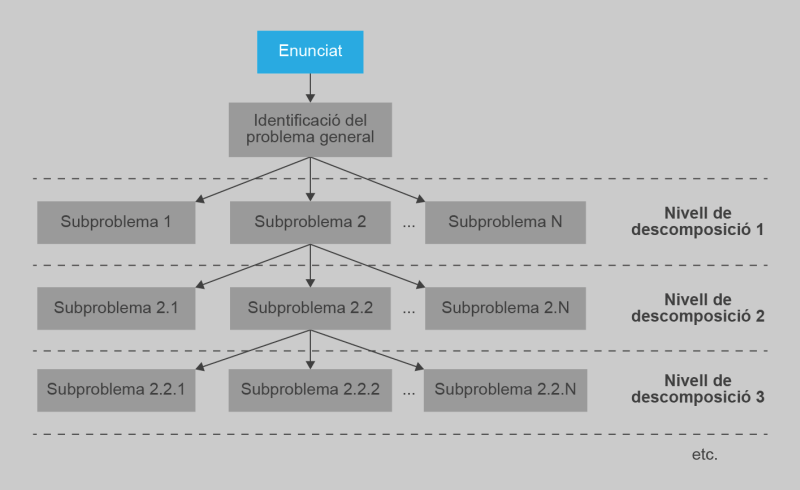
Hi ha dues estratègies bàsiques per resoldre la descomposició d'un problema: el disseny descendent i l'ascendent. Aquest apartat se centra en la primera, en ser la més utilitzada per norma general, i la més fàcil d'aplicar a l'hora de crear programes del nivell que abasten aquests materials.

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.

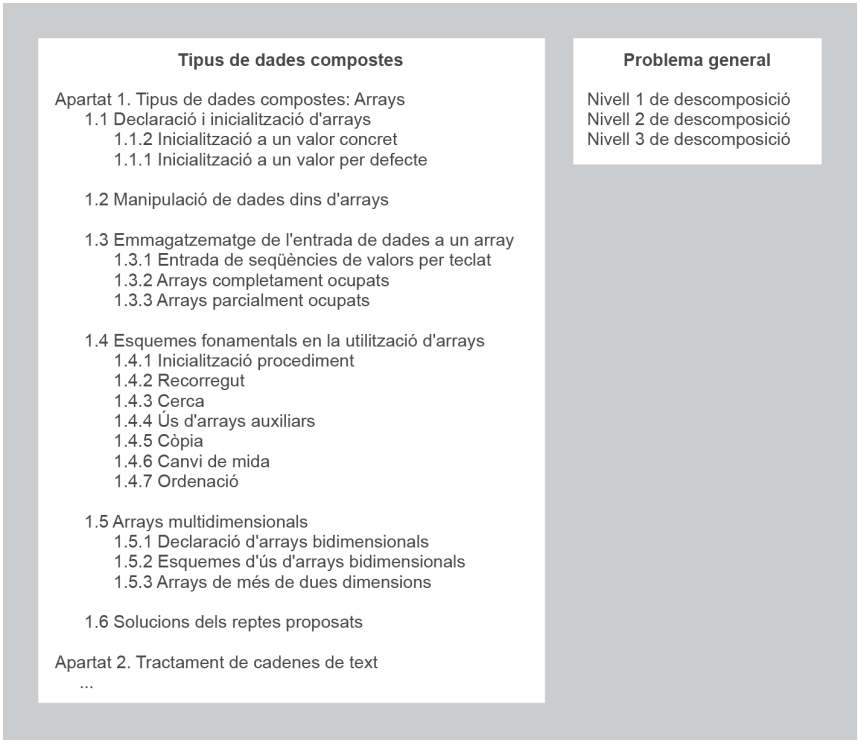
1.1.1 Exemples d'aplicació de disseny descendent

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.

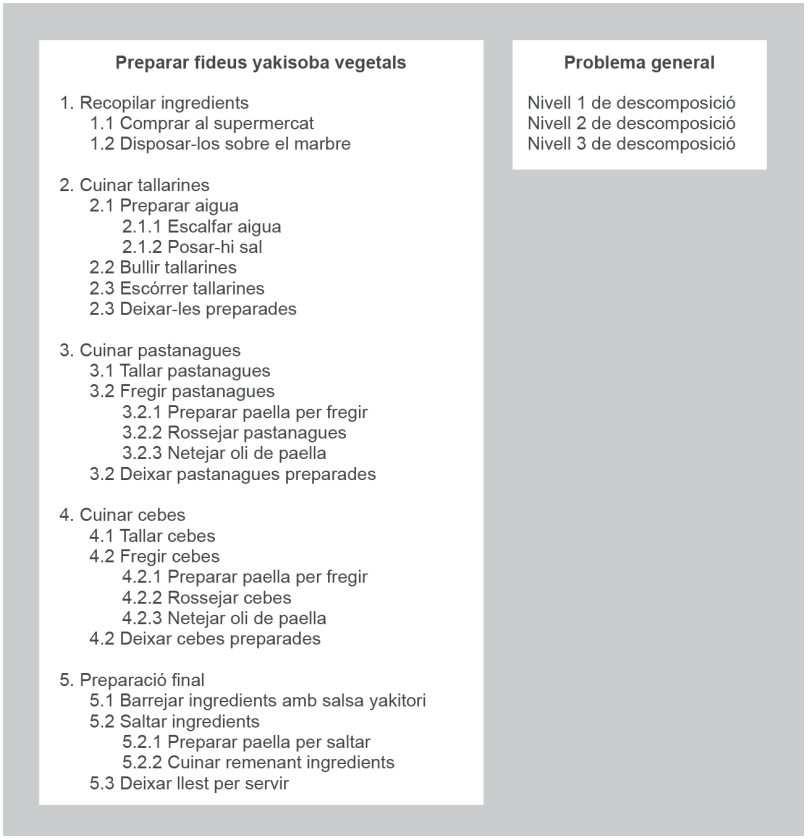
Una recepta de cuina

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

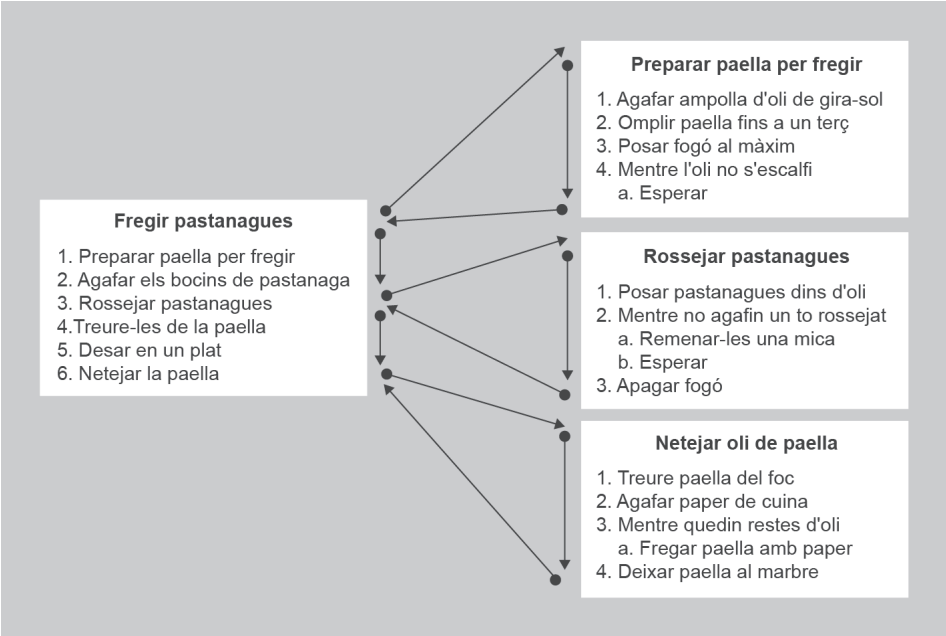
llavors s'aniria repetint per cada subproblema, normalment començant resolent els problemes més senzills (nivells inferiors), i anant a poc a poc resolent els més complexos (nivells superiors), fins a arribar al problema general, que és el de nivell més alt.

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

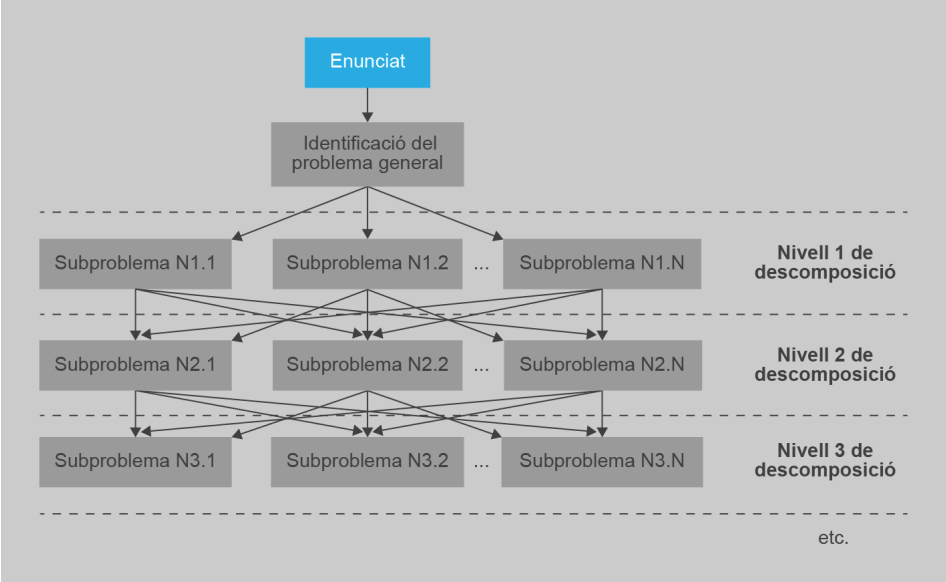
Ara bé, la descomposició mitjançant disseny descendent permet fer ús d'una característica molt útil quan s'usa per dissenyar algorismes. Es tracta de la possibilitat de cercar subproblemes idèntics, o si més no força semblants, i reaprofitar la seva solució en més d'un lloc dins del problema general. 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.

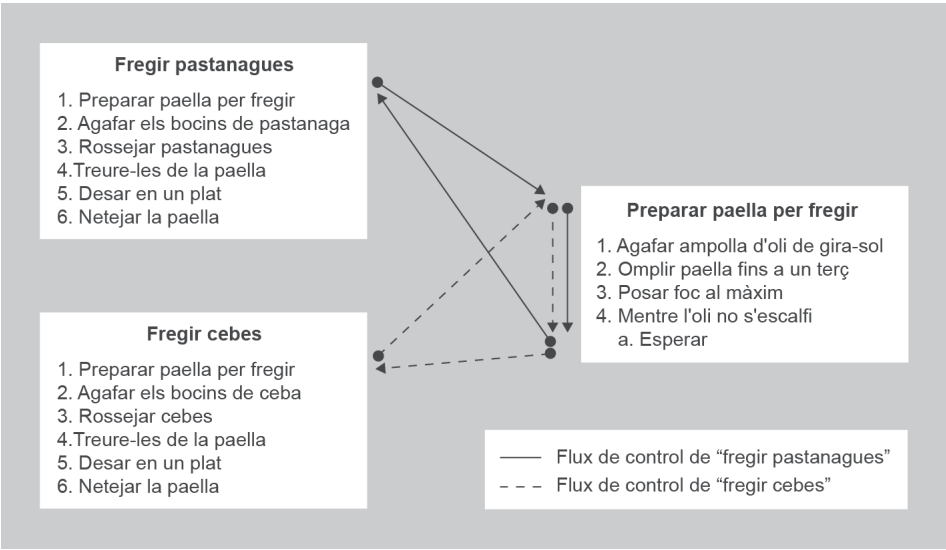
Un cop detectada aquesta característica del disseny descendent, és el moment de matisar la descripció de descomposició en nivells de la figura 1.1. En realitat, el procés és més aviat semblant al que descriu la figura 1.5, la qual indica un canvi de plantejament, ja que qualsevol subproblema d'un nivell 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.

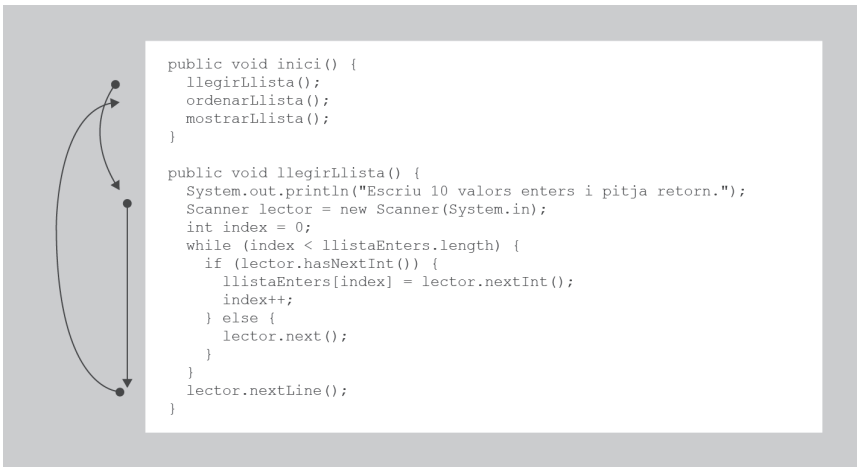
Si torneu a l'exemple, ara ja només us quedaria completar el codi associat al problema general (mètode **inici**). Per fer-ho, és possible invocar a **llegirLLista**, **ordenarLlista** i **mostrarLlista**. Donat aquest fet, el programa final ja seria el següent. Compileu-lo i executeu-lo per veure que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

```
//Mètodes associats al primer nivell de descomposició
  public void mostrarMenu() {
    System.out.println("\nBenvingut al registre de temperatures");
    System.out.println("————————————————————————————————————————");
    System.out.println("[RT] Registrar temperatures setmanals.");
    System.out.println("[MJ] Consultar temperatura mitjana.");
    System.out.println("[DF] Consultar diferència màxima.");
    System.out.println("[FI] Sortir.");
    System.out.print("Opció: ");
  }
  public void tractarOpcio() {
    Scanner lector = new Scanner(System.in);
    String opcio = lector.nextLine();
    if (opcio.equalsIgnoreCase("RT")) {
      registreTemperaturesSetmanals();
    } else if (opcio.equalsIgnoreCase("MJ")) {
      mostrarMitjana();
    } else if (opcio.equalsIgnoreCase("DF")) {
      mostrarDiferencia();
    } else if (opcio.equalsIgnoreCase("FI")) {
      finalitzarExecució();
    } else {
      System.out.println("Opció incorrecta!\n");
    }
  }
```

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!

```
public void inici() {
    while (!fi) {
      mostrarMenu();
      tractarOpcio();
    }
  }
```

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.");
    }
  }
```