Programació estructurada

Joan Arnedo Moreno

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

Índex

Introducció

Si bé quan es parla de les tasques del programador el primer que ve al cap és la generació del codi font, en realitat la seva feina va més enllà i engloba un procés més general que va des de saber definir exactament el problema que cal resoldre fins al desplegament de l'aplicació, passant per garantir que el programa és correcte. Per tant, abans de prosseguir amb els aspectes vinculats a la generació de codi font, val la pena donar una visió general del cicle de vida d'una aplicació. Dins d'aquest cicle de vida és especialment important el fet que, sense ni tan sols obrir les vostres eines de programació, heu de reflexionar i pensar com serà el vostre programa: esquematitzar quin és el seguit de passes que s'han de seguir perquè faci correctament la seva tasca. És a dir, definir un **algorisme**.

Per definir l'algorisme, cal tenir present que un programa es dedica principalment a transformar i manipular dades. Per tant, totes les instruccions vinculades a la seva gestió són de gran importància. En aquest aspecte, la tasca del programador és establir com cal utilitzar variables en l'avaluació d'expressions i l'emmagatzemament del resultat. Si un programa se cenyeix exclusivament a això, la seva estructura sempre tindrà forma de seqüència d'instruccions que es van executant de manera consecutiva, una darrere l'altra. Un cop totes i cadascuna de les instruccions s'han executat i s'arriba a la darrera de totes, el programa es dóna per finalitzat. Totes les instruccions sempre s'acaben executant, i cadascuna ho farà una sola vegada.

Ara bé, si us fixeu en els programes que useu habitualment, aquests estan plens de situacions en les quals això no es compleix. En un videojoc, quan s'inicia, sovint hi ha diferents opcions: començar la partida, recuperar una partida desada, entrar a la configuració, etc. A més a més, quan s'acaba la partida, no passa mai que s'acaba l'execució del programa i per tornar a jugar cal tornar-lo a executar. Normalment, us demana si voleu jugar de nou. Per tant, és clar que dins d'un programa hi ha moltes instruccions, però ni totes s'executen sempre, ni el camí que en segueix l'execució és únic cada cop que el posem en marxa. Aquest segueix una estructura més complexa, en què hi ha bifurcacions i bucles. I és que, precisament, una altra tasca també molt important del programador serà establir aquesta estructura: quin és el **flux de control**, o d'execució, del programa, cada cop que es posi en marxa. Per això, haurà de considerar sota quines condicions cal executar o no certes instruccions, i quan cal que algunes s'executin repetides vegades. En aquesta unitat veureu com es fa, mitjançant les sentències que permeten declarar **estructures de control**.

Com a pas previ a estudiar les estructures de control, l'apartat "Cicle de vida d'una aplicació" resumeix breument quines són les tasques del programador a l'hora de crear un programa i com podeu dissenyar els vostres algorismes.

A l'apartat "Estructures de selecció" s'expliquen quines sentències es poden usar

per crear bifurcacions dins del codi dels vostres programes, de manera que s'executin unes instruccions o unes altres de diferents d'acord amb certes condicions. Això permet que un programa actuï de manera diferent cada vegada que s'executa, normalment depenent de les dades que llegeixi del sistema d'entrada/sortida.

Tot seguit, a l'apartat "Estructures de repetició" s'explica com podeu dur a terme bucles d'instruccions dins del codi font, de manera que un mateix bloc s'executi un nombre consecutiu de vegades o fins que es compleixi alguna condició concreta. Això simplifica enormement algunes parts del codi o, si més no, el fa molt més curt, i permet poder tornar a dur a terme certes accions sense haver de tornar a executar el programa de nou des del principi.

Al llarg de la unitat, com a fil argumental, es partirà sempre de l'anàlisi d'un exemple en què es veu clarament de quina manera varia el comportament d'un programa en diferents execucions. Novament, després d'exposar temes importants, us proposem reptes per fer per tal d'entendre'ls millor.

Ara bé, heu de tenir en compte que, tot i que els exemples estaran centrats en la sintaxi específica del llenguatge Java, els conceptes explicats són genèrics de la immensa majoria de llenguatges de programació. Només us caldrà veure quina és la sintaxi exacta en cada cas, però l'estructura bàsica sol ser la mateixa. Per tant, igual que en la gestió de variables, heu de tenir present que aquests conceptes van més enllà del llenguatge Java.

Resultats d'aprenentatge

En finalitzar aquesta unitat, l'alumne/a:

1. Utilitza correctament tipus de dades simples i compostes emprant les estructures de control adients.

1. Algorismes

El desenvolupament d'una aplicació informàtica es pot dividir en un conjunt de fases que prenen com a punt de partida la detecció de la necessitat de fer un programa, i que finalitzen quan es comprova que aquest ja funciona correctament. Des d'un punt de vista simplista, un cop el programa està essent utilitzat, es podria considerar que la feina del desenvolupador ha acabat. Ara bé, moltes vegades es pot donar la circumstància que durant l'ús es detectin possibles canvis o millores que caldria fer. En aquest cas, caldrà recular a alguna de les fases prèvies i treballar-hi fins a disposar d'una nova versió del programa.

> En aquest darrer cas, apareix un cicle que fa que l'aplicació informàtica es trobi en contínua evolució. Per això s'acostuma a parlar del **cicle de vida** d'una aplicació informàtica.

En aquest apartat s'explicarà en què consisteix cadascuna de les fases d'aquest cicle de vida, enumerades tot seguit:

1. Definició del problema.
2. Disseny de l'algorisme.
3. Implementació del programa.
4. Verificació i proves.
5. Posada en marxa i manteniment.

1.1 Definició del problema

La definició del problema és el primer pas del procés que cal seguir per resoldre'l. Cal tenir molt clar què és el que s'ha de resoldre. I això implica una molt bona comunicació amb l'organització que encarrega la realització del programa i amb els usuaris finals, establint amb precisió i claredat els objectius que es volen assolir.

Normalment, el punt de partida s'estableix quan un client (ja sigueu vosaltres mateixos o una tercera persona), detecta que necessita resoldre diverses vegades un problema de certa complexitat i decideix que serà molt més fàcil si pot automatitzar les operacions mitjançant un programa. Arribats a aquesta conclusió, cal començar per la realització d'una anàlisi del problema, que de vegades pot dur bastant temps, ja que el problema pot ser difícil de comprendre o el client pot no tenir del tot clar què vol.

Un cop el problema està definit, cal dur a terme una **anàlisi funcional**. Aquesta no és estrictament responsabilitat del programador, raó per la qual no se'n farà èmfasi en aquest apartat, però això no implica que no n'hàgiu de ser coneixedors. A fi de comptes, els programadors sereu els encarregats de desenvolupar els programes resultants d'aquesta etapa.

En aquesta anàlisi, s'acostuma a fer el següent:

- Parlar amb tots els usuaris implicats de manera que aconseguim tenir tots els punts de vista possibles sobre el problema. Si vosaltres sou els usuaris la feina és més senzilla, evidentment.
- Donar totes les solucions possibles amb un estudi de viabilitat, considerant els costos econòmics (material i personal).
- Proposar al client, futur usuari de l'aplicació, una solució entre les possibles, i ajudar-lo a prendre la decisió més adequada.
- Trencar la solució adoptada en parts independents per dividir les tasques entre un equip de treball.

Com a resultat, es disposarà d'una idea clara de **què** cal fer i amb quins recursos. Només llavors és el moment de veure **com** s'ha de fer, en forma d'una aplicació informàtica.

1.2 Disseny de l'algorisme

Un cop establert quin és el problema que cal resoldre, és el moment d'aturar-se a reflexionar sobre quina estratègia caldrà seguir per resoldre'l mitjançant un programa i quina forma haurà de prendre el codi font. Per establir-ho, cal recordar clarament què pot fer un ordinador (quina mena d'ordres pot acceptar) i què hi ha dins d'un programa. Pel que fa a què pot fer un ordinador, bàsicament es tracta de processar dades. Quant a la composició d'un programa, es pot considerar que hi ha dos elements fonamentals que col·laboren per dur a terme la tasca encomanada. D'una banda, hi ha les dades que cal processar, ja sigui directament en forma de literals o introduïdes per l'usuari. D'una altra banda, el conjunt d'ordres que es dóna a l'ordinador, les instruccions, per tal que es produeixi aquest procés de transformació.

Què és un programa

Una definició clàssica és:
Programa = dades + algorisme.

> Dins d'un programa es representarà el que es coneix formalment com a **algorisme**: un mecanisme per resoldre un problema o una tasca concreta, descrit com una seqüència finita de passes.

Si bé tots els programes contenen algorismes, traduïts en un seguit d'instruccions, aquest concepte no està lligat exclusivament a la programació. Qualsevol descripció composta per un seguit de passes que cal seguir ordenadament per assolir una

fita és un algorisme. Fora del món dels ordinadors, un dels exemples més clars són les receptes de cuina. Per exemple, el conjunt següent de passes podria ser l'algorisme per fer un parell d'ous ferrats:

1. Agafar dos ous de la nevera.
   (a) Si no n'hi ha, anar-ne a comprar.
2. Agafar sal i oli de l'armari.
   (a) Si no n'hi ha, anar-ne a comprar.
3. Posar oli a la paella.
4. Posar la paella al foc.
5. Mentre l'oli no estigui calent, cal esperar.
6. Per cada ou cal fer el següent:
   (a) Trencar-lo i posar el contingut a la paella.
   (b) Posar-hi sal.
7. Mentre els ous no estiguin fregits, cal esperar.
8. Treure els ous de la paella i servir-los.

En qualsevol cas, el fet primordial és que, tant per establir què ha de fer una persona com un ordinador, abans cal pensar exactament com cal dividir la tasca en accions individuals i quin ordre han de seguir. Per tant, una de les tasques primordials del programador abans de ni tant sols començar a escriure codi font és dedicar un temps a reflexionar i a dissenyar un algorisme que serveixi per dur a terme la tasca donada per la definició del problema. Aquesta fase és molt important, ja que condicionarà totalment la fase d'implementació. Si l'algorisme és incorrecte, haureu perdut temps i esforç creant un codi font que no fa el que voleu.

1.2.1 Programació estructurada

Hi ha diferents aproximacions per dissenyar algorismes. La que aprendreu en aquest mòdul és l'anomenada **programació estructurada**. En aquesta, les passes d'un algorisme es divideixen en diferents blocs d'instruccions, cadascun triat únicament entre els tres tipus d'estructura següents:

- **Estructura lineal o seqüencial**: les instruccions s'executen en el mateix ordre en què s'han escrit. Aquest és el que heu vist fins ara. En l'exemple de l'ou ferrat, equivaldria a la transició entre el pas 5 i 6. Un sempre va darrere de l'altre, seqüencialment.

- **Estructura de selecció o condicional**: hi ha certes condicions que provoquen l'execució de blocs d'instruccions diferents depenent de si es compleixen o no aquestes condicions. En l'exemple de l'ou ferrat, aquest és el cas dels passos 1 o 2, en els quals, donada una condició, l'acció és diferent. Depenent de si hi ha ous a la nevera, se'n poden agafar o bé caldrà anar a comprar-ne.

- **Estructura de repetició o iteratiu**: el bloc d'instruccions s'executa un nombre finit de vegades, ja sigui un nombre concret o fins que es compleix una condició. En el cas de l'ou ferrat, el primer cas d'aquesta estructura es veu al pas 6, en què cal repetir les mateixes ordres un cert nombre de vegades (dues, una per cada ou). El segon supòsit s'esdevé a les passes 5 i 7, en què cal fer l'acció d'esperar mentre no es doni una condició concreta.

En un ordinador: obrir un fitxer de text

Quan executeu un editor de text i seleccioneu obrir un fitxer, de ben segur que l'ordinador ha de fer un conjunt de tasques una darrere de l'altra abans de poder editar el text. Primer us ha de preguntar on es troba el fitxer en qüestió. Després el busca i el carrega. I finalment processa les dades perquè siguin mostrades correctament a la pantalla. Cada pas pot ser més o menys complex, però és clar que han de seguir un ordre i no es pot avançar al següent fins haver finalitzat l'anterior. Es fan de manera seqüencial.

En un ordinador: les opcions d'un editor de text

Quan executeu un editor de text, de ben segur que veureu que hi ha un gran nombre d'accions diferents que podeu dur a terme: obrir un document, crear-ne un de nou, desar el document actual, etc. Cadascuna d'aquestes opcions es veu clarament diferenciada de la resta, ja que solen estar associades a elements gràfics independents, dins un menú o una barra d'eines. El que és clar és que el conjunt d'instruccions que cal executar en cada cas serà diferent, ja que cada opció fa una tasca diferent. Per tant, cal un mecanisme per establir quin bloc d'instruccions concret s'executa cada cop que seleccioneu cada opció.

En un ordinador: les opcions d'un editor de text

A l'hora d'imprimir un document en un editor de text, sovint podeu demanar quantes còpies es volen enviar a la impressora. Aquest nombre de còpies pot ser des d'una fins a un valor qualsevol, fins al límit dels fulls de paper i tinta de què disposeu. D'entrada, es pot establir que l'acció d'imprimir un únic document es basarà en un bloc d'instruccions concret. Cada cop que s'executen aquestes instruccions, l'ordinador imprimeix una còpia del document. Per tant, una manera senzilla de dur a terme moltes còpies d'un document seria simplement repetir tants cops com còpies vulguem el conjunt d'instruccions que s'usaria per imprimir-ne només una.

Aquests tres tipus d'estructura, anomenades a escala general **estructures de control**, permeten establir quines instruccions del vostre programa cal executar en cada moment i especificar l'ordre en què s'executen les instruccions i quins camins segueixen en el procés d'execució. És a dir, establir quin és el **flux de control** del programa.

De moment, només heu tingut contacte amb l'estructura seqüencial, a la qual corresponen tots els exemples vistos fins ara. En els apartats següents veureu com cal usar les estructures de selecció i repetició.

1.2.2 Representació d'algorismes

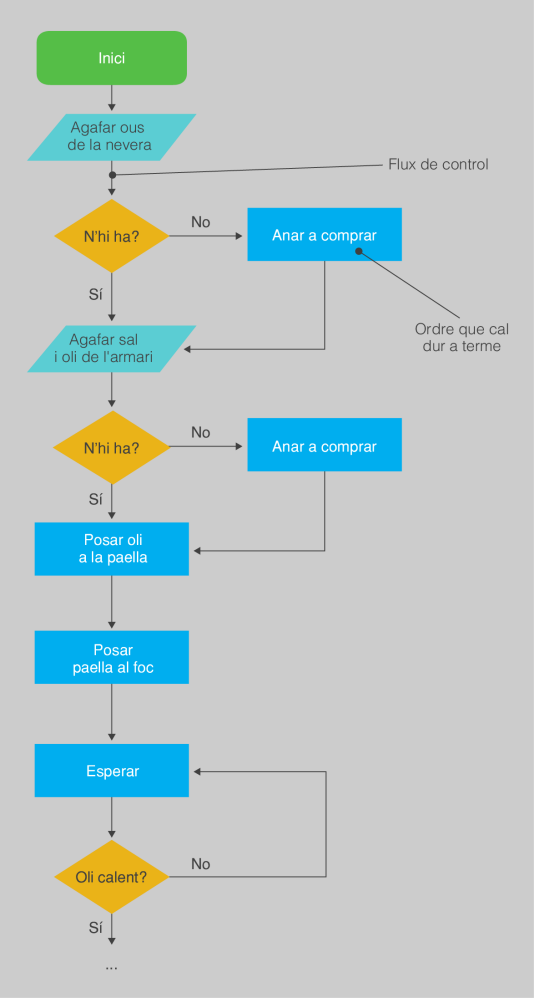
Pseudocodi

El pseudocodi és un llenguatge informal d'alt nivell que usa les convencions i l'estructura d'un llenguatge de programació, però que està orientat a ser entès pels humans.

Un algoritme pot ser representat mitjançant molts tipus de notacions. Una manera molt simple és amb una llista de frases expressades amb les vostres paraules (llenguatge natural), tal com s'ha vist en l'exemple de l'ou ferrat. Aquesta és una aproximació suficient per fer-se una idea de les tasques que cal fer. Ara bé, també és un sistema que pot ser ambigu i resultar confús per a algorismes complexos. Hi ha altres mecanismes de notació més formal, especialment usats quan l'algorisme per descriure formarà part d'un programa d'ordinador. Entre aquests hi ha el pseudocodi, els diagrames de flux de control o, directament, els codis font dels diferents llenguatges de programació.

A partir d'ara, per esquematitzar el funcionament de certes estructures de control s'usaran diagrames de control de flux, mentre que per representar algorismes complets, s'usarà directament codi font en llenguatge Java.

FIGURA 1.1. Aspecte d'un diagrama de flux de control



Un **diagrama de flux de control** consisteix en una subdivisió de passes seqüencials, d'acord amb les sentències i estructures de control d'un programa, que mostra els diferents camins que pot seguir un programa a l'hora d'executar les seves instruccions. Cada passa s'associa a una figura geomètrica específica.

L'usarem exclusivament per aclarir-ne el funcionament. No s'explicarà amb detall tot el conjunt de símbols que es poden usar. Tampoc no se seguirà el format fins al darrer detall. És suficient que disposeu d'uns esquemes senzills que serveixin per donar-vos una idea clara del flux de control del programa per a cada estructura usada, representada de manera visual. A títol merament il·lustratiu, perquè tingueu un primer contacte de l'aspecte que tenen, la figura 1.1 mostra el diagrama de flux de control de part de l'algorisme per fer ous ferrats. Observeu com les figures geomètriques especifiquen les instruccions que es van executant i com les fletxes estableixen l'ordre d'execució de les instruccions: el flux de control. En el cas de la descripció de l'algorisme d'un programa, el contingut de cada figura, l'ordre que cal seguir, correspondrà a les instruccions del programa.

1.3 Implementació del programa

Una vegada heu definit l'algorisme, cal convertir-lo a codi font en un fitxer, d'acord amb un llenguatge de programació concret, per tal d'obtenir un fitxer executable. Segons la notació emprada per representar l'algorisme dissenyat, això serà més fàcil o més complicat. Evidentment, si l'algorisme s'ha representat mitjançant la sintaxi d'un llenguatge concret, convertir-lo a codi font és molt fàcil, ja que és un senzill exercici d'enganxar i copiar.

Durant aquesta etapa, és important que tot el procés estigui ben documentat, per tal que qualsevol altre desenvolupador, fins i tot el programador mateix quan ja ha passat un cert temps, pugui retocar l'aplicació si cal. Penseu que en un programa informàtic s'automatitza una forma d'actuació, la qual cosa implica reflectir un cert coneixement. Però el coneixement no és únic i, per tant, entendre el que un altre desenvolupador o un mateix va decidir en el passat és molt difícil. La documentació és l'única manera de garantir aquest coneixement. Per tant, el mínim que es pot demanar quan s'implementa un programa és que es comenti el codi font de manera que sigui més entenedor. En aquest aspecte, s'espera que com a mínim s'expliqui amb comentaris què fa el programa i els blocs d'instruccions que es considerin de comprensió difícil.

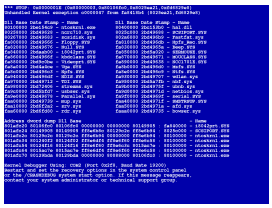
1.4 Verificació i proves

Abans que un programa es pugui considerar finalitzat i es posi a disposició del client, caldrà detectar possibles errades i comprovar que actua com s'espera. Hi

ha moltes estratègies per provar la correcció d'un programa, algunes de molt complexes. Per a aquest mòdul, n'hi ha prou amb una aproximació relativament simple però més que efectiva: usar *jocs de proves*.

> Un **joc de proves** és un conjunt de situacions o entrades de dades que permeten provar l'actuació del programa. Aquest conjunt hauria de ser complet, en el sentit que ha d'abastar totes les possibilitats reals.

Primer caldrà dissenyar i preparar el joc de proves per executar el programa posteriorment en les diferents situacions preparades. D'aquesta manera se'n pot comprovar el bon funcionament en tot moment. En cas que sempre funcionin, podeu estar més o menys segurs de la fiabilitat del programa. Ara bé, la realitat és que sempre es produeixen situacions no previstes que s'acabaran detectant quan el programa ja estigui en marxa i essent utilitzat pel client. De ben segur que algun cop us heu trobat amb programes comercials, com alguns sistemes operatius, que tot i estar a la venda i considerats finalitzats, es comportaven anòmalament en condicions normals (es pengen, reinicien l'ordinador, etc.). En programes complexos, es fa difícil detectar fins al darrer error possible.



Un comportament erroni clàssic: la pantalla blava del sistema operatiu Windows. Imatge de Wikimedia Commons

Normalment, quan un programa no actua correctament, el problema pot estar en el següent:

- El disseny de l'algorisme.
- La codificació, que és semànticament incorrecta (però amb sintaxi correcta).

Disposeu d'un annex on us expliquem el funcionament del depurador inclòs en l'IDE que usareu al llarg del curs.

A vegades l'error es troba de manera senzilla amb un simple seguiment de l'algorisme dissenyat, observant si les variables es modifiquen correctament i prenen els valors esperats donades unes dades inicials concretes. Però la majoria de les vegades l'error costa de trobar perquè es parteix de la premissa que l'algorisme és correcte, cosa que no necessàriament ha de ser certa. En aquest cas, un mecanisme inavaluable per trobar l'errada és dur a terme la depuració del programa.

Un programa depurador (*debugger*, en anglès) és un programa que permet:

El terme original *debug* vol dir literalment 'eliminar cuques'.

- Executar un programa instrucció per instrucció i veure què passa després de l'execució de cadascuna
- Obtenir el valor de les dades abans i després d'executar una instrucció
- Modificar el valor de les dades durant l'execució
- Interrompre o aturar l'execució del programa en qualsevol punt

La majoria d'entorns de programació actuals disposen d'eines de depuració. Si no és el cas, el programador no tindrà més remei que dur a terme la depuració manualment. Aquest mètode consisteix a intercalar temporalment enmig del codi del programa instruccions que mostrin per pantalla els valors de les variables,

Col·loquialment, aquestes instruccions per mostrar valors de variables s'anomenen "xivatos".

per tal de fer el seguiment de l'execució i localitzar la causa del problema. Per fer-ho, es poden usar les instruccions que aporti el llenguatge per mostrar dades a la pantalla, com `System.out.println(...)` en Java. Un cop el programa funciona correctament, caldrà treure el codi, o si més no convertir aquestes instruccions en comentaris.

1.5 Posada en marxa i manteniment

Una vegada es té el programa final, suposadament lliure d'errors, cal instal·lar-lo seguint les instruccions de l'entorn de treball i posar-lo en funcionament. Això és el que formalment s'anomena la **fase d'explotació**.

Sembla que arribeu al final del procés, però encara ens queda una acció: formar l'usuari. Cal tenir paciència. Penseu que vosaltres heu estat molt de temps pensant en el problema i treballant en la solució. Per a vosaltres és la solució normal, la teniu assumida, però per a l'usuari no és necessàriament la solució desitjada o esperada. Per tant, caldrà tenir una bona dosi de paciència per explicar el funcionament de l'aplicació. Aquesta formació ha d'estar sempre acompanyada de la guia o manual de l'usuari. No ha de ser un llibre de lectura difícil; haurien de ser els apunts que li permetessin moure's ràpidament per l'aplicació, tractant totes les possibilitats d'una manera ràpida i clara.

Arribats a aquest punt ja us disposeu a descansar, però podeu tenir la mala sort que amb l'ús de l'aplicació s'arribi a la conclusió que el programa no fa exactament el que se n'espera o es detecten comportaments erronis que han eludit la fase de proves. Des d'una perspectiva més positiva, també pot ser que en aquesta fase el client estigui molt satisfet i detecti que estaria bé que, ara que ja té una aplicació funcional, us torni a contractar per afegir-hi noves funcionalitats. En aquest cas, caldrà retrocedir fins a la fase corresponent i fer les modificacions pertinents. Comença una nova iteració en el cicle de vida de l'aplicació.

2. Estructures de selecció

Entre els diferents tipus d'estructures de control que permeten establir el flux de control d'un programa, les més fàcils d'entendre són aquelles que creen bifurcacions o camins alternatius, de manera que, segons les circumstàncies, s'executi un conjunt d'instruccions o un altre. D'aquesta manera, donades diferents execucions d'un mateix codi font, part de les instruccions que s'executen poden ser diferents per a cada cas.

> Les **estructures de selecció** permeten prendre decisions sobre quin conjunt d'instruccions cal executar en un punt del programa. O sigui, seleccionar quin codi s'executa en un moment determinat entre camins alternatius.

Tota estructura de selecció es basa en l'avaluació d'una expressió que ha de donar un resultat booleà: `true` (cert) o `false` (fals). Aquesta expressió s'anomena la **condició lògica** de l'estructura.

El conjunt d'instruccions que s'executarà dependrà del resultat de la condició lògica, i actuarà com una mena d'interruptor que marca el flux que cal seguir dins del programa. Normalment, aquesta condició lògica es basa en part, o en la seva totalitat, en valors emmagatzemats en variables amb un valor que pot ser diferent per a diferents execucions del programa. En cas contrari, no té sentit usar una estructura de selecció, ja que mentre s'està escrivint el programa ja es pot predir quin serà el resultat de l'expressió. Per tant, com que sempre serà el mateix, sempre s'executaran les mateixes instruccions sense que hi hagi cap bifurcació possible.

Hi ha diferents models de fluxos alternatius a l'hora d'executar instruccions, tot i que per a tots s'usa la mateixa família de sentències i una estructura similar en el codi font. Per tant, tots els aspectes destacats per a un dels models s'apliquen també a tots els altres. També cal dir que, si bé els exemples i la sintaxi descrita en aquest mòdul se centren en el llenguatge Java, la majoria de les estructures de selecció descrites són compartides amb els altres llenguatges de programació, per la qual cosa el concepte general és aplicable més enllà del Java. Només caldrà que cerqueu a la documentació la sintaxi específica per al llenguatge triat.

2.1 Una desviació temporal del camí: selecció simple

El cas més simple dins de les estructures de selecció és aquell en què hi ha un conjunt o bloc d'instruccions que només voleu que s'executin sota unes circumstàncies concretes. En cas contrari, aquest bloc és ignorat i, des del punt

La selecció simple, una desviació temporal en el camí d'instruccions. Imatge de Wikimedia Commons

de vista de l'execució del programa, és com si no existís. Un exemple seria el programa d'una botiga virtual que aplica un descompte al preu final d'acord amb un cert criteri (per exemple, si la compra total és com a mínim de 100 €). En aquest cas, hi ha un conjunt d'instruccions, les que apliquen el descompte, que només s'executen quan es compleix la condició. En cas contrari, s'ignoren i el preu final és el mateix que l'original.

L'estructura de **selecció simple** permet controlar el fet que s'executi un conjunt d'instruccions si i només si es compleix la condició lògica (és a dir, el resultat d'avaluar la condició lògica és igual a `true`). En cas contrari, no s'executen.

2.1.1 Sintaxi i comportament

Per dur a terme aquest tipus de control sobre les instruccions del programa, cal usar una sentència `if` (*si*...). En el cas del Java, la sintaxi és la següent:

```
instruccions del programa
if (expressió booleana) {
  Instruccions per executar si l'expressió avalua a true (cert) – Bloc if
}
resta d'instruccions del programa
```

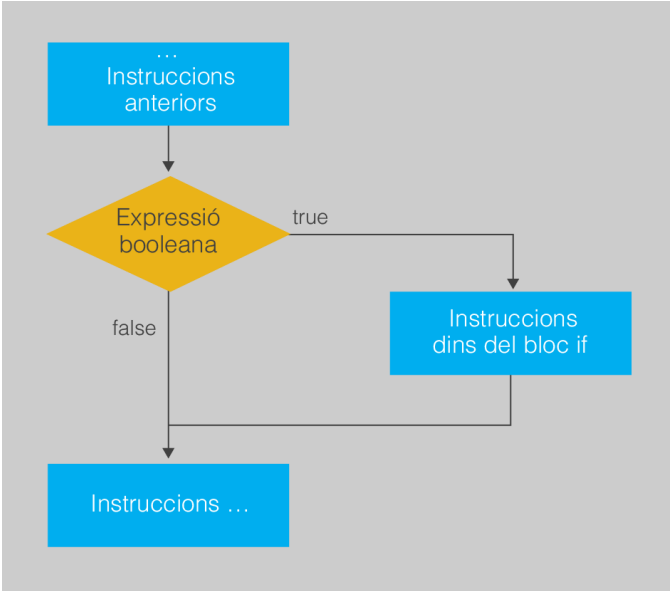
if

El nom de la sentència `if` bàsicament diu: "Si es compleix certa condició, fes això...".

Si entre els parèntesis es posa una expressió que no avalua un resultat de tipus booleà, hi haurà un error de compilació.

La figura 2.1 mostra un diagrama del flux de control d'aquesta sentència, que estableix els diferents blocs d'instruccions que s'executen en cada cas, depenent del resultat d'avaluar l'expressió booleana.

Figura 2.1. Diagrama de flux de control per a una selecció simple



Un diagrama de **flux de control** consisteix en una subdivisió de passes seqüencials, d'acord amb les sentències i estructures de control d'un programa, que mostra els diferents camins que pot seguir el programa a l'hora d'executar les instruccions. Cada passa s'associa amb una figura geomètrica específica.

A partir d'ara, per esquematitzar el funcionament de les estructures de control s'usarà aquesta mena de diagrames. Només l'usarem per aclarir-ne el funcionament, de manera que no s'explicarà amb detall tot el conjunt de símbols que es poden usar. A l'hora d'usar-los, tampoc no se seguirà el format fins al darrer detall. És suficient que disposeu d'uns esquemes senzills que serveixin per donar-vos una idea clara del flux de control del programa per a cada estructura usada de manera visual.

Una possible bifurcació en el camí d'un flux de control s'indica amb un rombe que conté una expressió per avaluar.

2.1.2 Exemple: calcular un descompte

Es vol fer un programa que apliqui un descompte a un preu depenent del seu valor. Per veure clarament que hi ha diferents camins dins de les instruccions, primer de tot establirem quines són les tasques que ha de fer el programa i en quin ordre.

El programa hauria de fer:

1. Decidir quin és el valor mínim per optar al descompte i quant es descomptarà.
2. Demanar que s'introdueixi el preu inicial, en euros, pel teclat.
3. Llegir-lo.
4. Veure si el preu introduït és igual o major que el valor mínim per optar al descompte.
   (a) Si és així, s'aplica el descompte sobre el preu inicial.
5. Mostrar el preu final.

Recordeu que pot ser que Java llegeixi els enters introduïts per teclat amb els decimals indicats amb coma, i no punt, segons la configuració local.

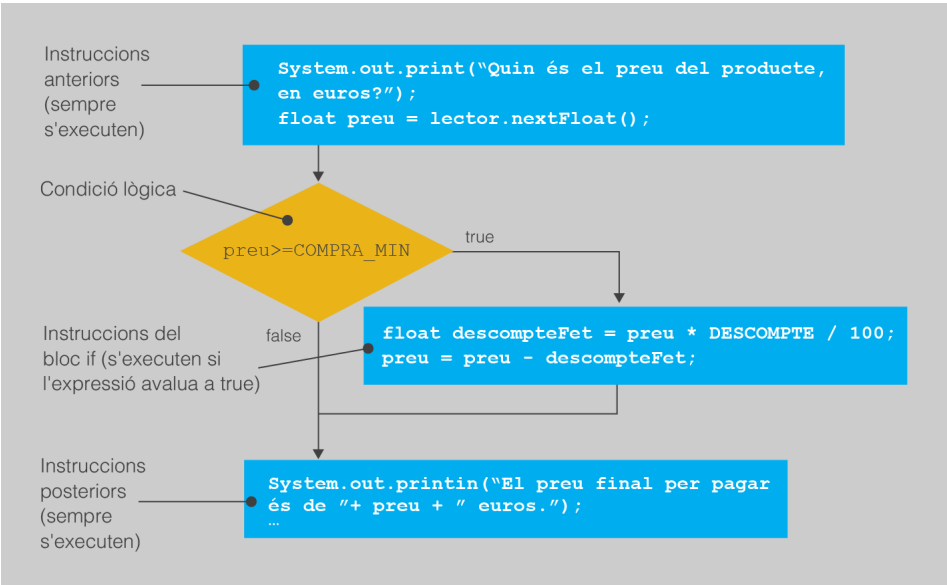
En el pas 4 es pot observar que cal prendre una decisió d'acord amb una condició i que alguna de les tasques només es fa si aquesta es compleix (el pas I). Per tant, queda establert que cal una estructura de selecció simple. Partint d'aquí, un possible codi font que correspondria a aquest esquema seria el següent. Observeu quines instruccions es corresponen a cadascuna de les passes descrites.

Tot seguit es mostra el codi font que du a terme aquestes tasques. Compileu-lo, executeu-lo i observeu com el resultat final mostrat per pantalla és diferent segons el valor que introduïu pel teclat.

```java
import java.util.Scanner;
//Un programa que calcula descomptes.
public class Descompte {
  //PAS 1
  //Es fa un descompte del 8%.
  public static final float DESCOMPTE = 8;
  //Es fa descompte per compres d'un mínim de 100 euros.
  public static final float COMPRA_MIN = 100;
  public static void main (String[] args) {
    Scanner lector = new Scanner(System.in);
    //PAS 2 i 3
    System.out.print("Quin és el preu del producte, en euros? ");
    float preu = lector.nextFloat();
    lector.nextLine();
    //PAS 4
    //Estructura de selecció simple.
    //Si l'expressió avalua true, executa el bloc dins l'if.
    //En cas contrari, ignora'l.
    if (preu >= COMPRA_MIN) {
      //PAS I
      float descompteFet = preu * DESCOMPTE / 100;
      preu = preu - descompteFet;
    }
    //PAS 5
    System.out.println("El preu final per pagar és de " + preu + " euros.");
  }
}
```

A la figura 2.2 es representa el diagrama de flux de control d'aquest programa en concret. A la figura es visualitza a quina part del diagrama correspon cada part del codi.

FIGURA 2.2. Diagrama de flux de control per al programa de càlcul d'un descompte



Repte 1: modifiqueu el programa perquè, en lloc de fer un descompte del 8% si la compra és de 100 € o més, apliqui una penalització de 2 € si el preu és inferior a 30 €.

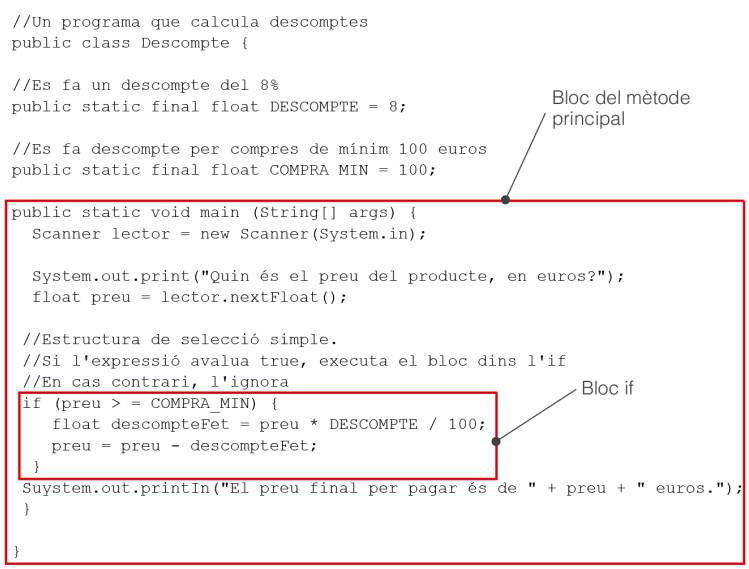
2.1.3 Aspectes importants de la selecció simple

La introducció d'estructures de selecció en un programa complica la sintaxi del codi font, per la qual cosa cal ser molt acurat a l'hora d'escriure una sentència `if`. Si no ho feu, o bé el compilador donarà error, o bé el programa no es comportarà correctament, i haureu de repassar el codi font per veure on és l'error. Cal dir que el segon cas acostuma a ser molt més empipador que el primer. Per tant, val la pena fer un repàs dels aspectes més rellevants de la selecció simple. Aquestes qüestions afecten en una mesura o altra totes les estructures de selecció.

- L'expressió booleana que denota la condició lògica pot ser tan complexa com es vulgui, però ha d'estar sempre entre parèntesis.
- Les instruccions que cal executar si la condició és certa estan englobades entre dues claus (`{`, `}`). Aquest conjunt es considera un bloc d'instruccions associat a la sentència `if` (*bloc `if`*).
- La línia on hi ha les claus o la condició no acaba mai en punt i coma (`;`), al contrari que altres instruccions.
- Tot i que no és imprescindible, és un bon costum que les instruccions del bloc estiguin sagnades.

Vist això, val la pena remarcar que amb la introducció d'estructures de selecció simple també apareix per primer cop codi font amb diferents blocs d'instruccions: el del mètode principal i els associats a la sentència `if`. La principal característica d'aquest fet és que la relació entre blocs és jeràrquica: tots els nous blocs d'instruccions són subblocs del mètode principal. La figura 2.3 mostra els diferents blocs existents en l'exemple.

FIGURA 2.3. Blocs d'instruccions en el programa d'exemple



Com anireu veient, aquesta circumstància es repetirà en totes les estructures de control d'un programa. Per això heu de tenir molt clar on comença i acaba cada bloc, i quins blocs són subblocs d'un altre.

També val la pena fer èmfasi que, quan s'usin estructures que delimiten blocs d'instruccions diferents, identificats per estar escrits entre claus ({, }), és important sagnar cada línia. D'aquesta manera, es facilita la llegibilitat del codi font i la identificació de cada bloc d'instruccions. Si us hi fixeu, veureu que, de fet, això ja s'havia aplicat fins ara a les instruccions dins del mètode principal (també delimitades per claus), respecte a la declaració d'inici de la classe. Totes les instruccions del mètode principal estan sagnades respecte al marge de la declaració del mètode principal.

A mode de resum, la taula 2.1 mostra una petita llista d'errors típics que es poden cometre en usar una sentència `if`.

TAULA 2.1. Errors típics en usar una sentència "if"

Missatge d'error en compilar	Error comès	Exemple
'(' expected	Expressió no envoltada de parèntesis	`if preu >= COMPRA_MIN) {`
';' expected	Falta `;` en alguna instrucció del bloc `if`	`preu = preu - descompteFet`
Sempre executa bloc `if`	S'ha posat `;` a la sentència `if`	`if (preu >= COMPRA_MIN); {`
Només fa condicionalment la primera instrucció del bloc `if`, la resta les fa sempre	Falten les claus que han d'envoltar el bloc, `{...}`	`if (preu >= COMPRA_MIN)`

Vinculat al darrer error, cal esmentar que quan el bloc `if` només té una única instrucció, l'ús de claus és opcional. De totes maneres, és molt recomanable usar-les sempre, independentment d'aquest fet. Això facilita la identificació del bloc de codi associat a la sentència `if` quan s'està llegint el codi font.

```
1  if (preu >= COMPRA_MIN)
2    preu = preu – (preu * DESCOMPTE / 100);
3
4  és equivalent a
5
6  if (preu >= COMPRA_MIN) {
7    preu = preu – (preu * DESCOMPTE / 100);
8  }
```

Ara bé, alerta!

```
1  if (preu >= COMPRA_MIN)
2    float descompteFet = preu * DESCOMPTE / 100;
3    preu = preu – descompteFet;
4
5  és equivalent a (fixeu–vos on estan ubicades les claus)
6
7  if (preu >= COMPRA_MIN) {
8    float descompteFet = preu * DESCOMPTE / 100;
9  }
10 preu = preu – descompteFet;
```

2.2 Dos camins alternatius: la sentència ”if/else”

La selecció doble, una bifurcació en el camí. Imatge de Wikimedia Commons

Suposeu que ara voleu fer un programa en què s'ha d'intentar endevinar un nombre entre 1 i 10. En aquest cas, i a diferència de l'anterior, ara hi ha dos escenaris excloents: o s'ha endevinat el nombre, o no s'ha endevinat. Segons de quin cas es tracti, la resposta del programa ha de ser diferent. Per tant, a més a més de codi comú per a qualsevol cas i d'un bloc que especifiqui les accions per dur a terme si es compleix la condició, ara en cal un altre que indiqui què cal fer només en cas contrari. Per poder fer això tenim l'estructura de selecció doble, la sentència `if/else` (si... si no...).

> L'estructura de **selecció doble** permet controlar el fet que s'executi un conjunt d'instruccions, només si es compleix la condició lògica, i que se n'executi un altre, només si no es compleix la condició lògica/ Imatge de Wikimedia Commons

És important recordar que els dos blocs de codi són excloents. Mai no pot passar que tots dos s'acabin executant.

2.2.1 Sintaxi i comportament

Per dur a terme aquest tipus de control sobre les instruccions del programa, cal usar una sentència `if/else` (*si... si no...*). En el llenguatge Java, la sintaxi és la següent:

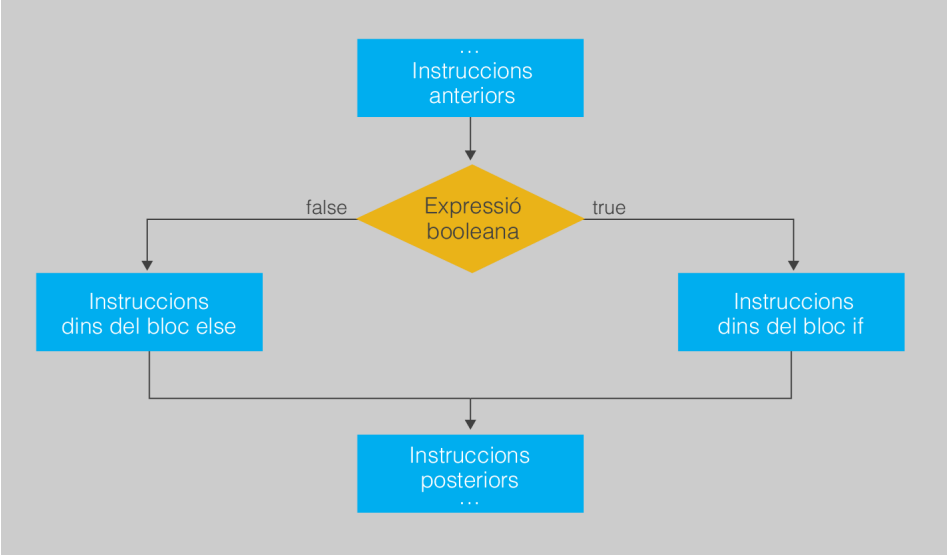
if/else

La sentència `if/else` bàsicament diu: “Si es compleix certa condició, fes això... i si no, això altre...”.

```
1 if (expressió booleana) {
2   Instruccions per executar si l'expressió l'avalua a true (cert) – Bloc if
3 } else {
4   Instruccions alternatives per executar si l'expressió l'avalua a false (fals)
        – Bloc else
5 }
```

El bloc `if` té exactament les mateixes característiques que quan s'usa en una estructura de selecció simple. Aquestes es compleixen també per al cas del bloc `else`, amb la particularitat que no té cap expressió assignada. Simplement, quan la condició lògica de la sentència `if` no es compleix s'executen les instruccions del bloc `else`. La figura 2.4 en mostra el diagrama del flux de control a escala genèrica.

FIGURA 2.4. Diagrama de flux de control d'una selecció doble



Si per algun motiu no es posa cap instrucció dins del bloc `else` i es deixa un espai buit entre les dues claus, la selecció doble es comporta igual que la selecció simple. En aquest cas, és millor usar només la sentència `if` en lloc de `if/else`.

2.2.2 Exemple: endevina el nombre secret

Un possible codi d'un programa en què s'intenta endevinar un nombre serveix per mostrar amb més claredat com es pot usar una sentència `if/else`. Novament, abans de veure'l val la pena reflexionar sobre quines són les tasques que hauria de fer el programa i en quin ordre.

Bàsicament, el programa ha de fer:

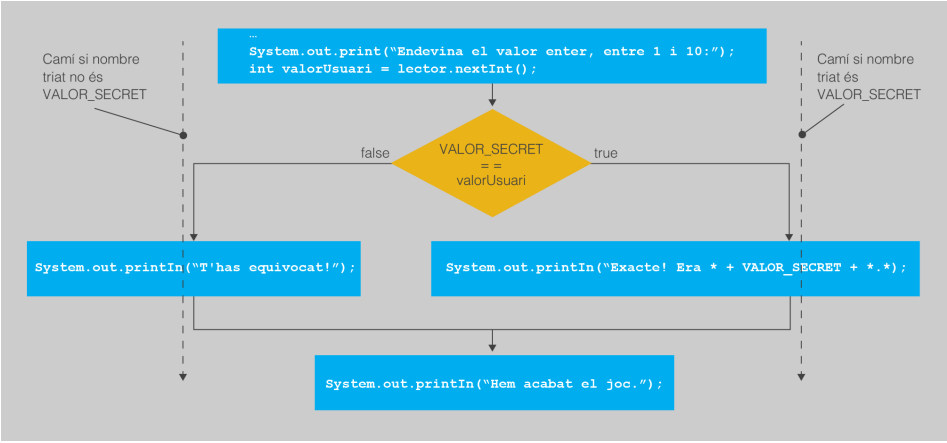
1. Decidir quin serà el nombre per endevinar.
2. Demanar que s'introdueixi un nombre pel teclat.
3. Llegir-lo.
4. Veure si el nombre introduït és igual al valor secret:
   (a) Si és igual que el nombre pensat, informa que s'ha encertat.
   (b) Si no, informa que s'ha fallat.

Queda clar que en el pas 4 cal prendre una decisió d'acord amb una condició. El programa haurà d'emprendre dues vies d'acció diferents segons si la condició es compleix o no. Per tant, cal una estructura de selecció doble. Partint d'aquí, un possible codi font que correspondria a aquest esquema seria el següent. De nou, observeu quines instruccions es corresponen a cadascun dels passos descrits. Compileu-lo i executeu-lo per veure com funciona.

```java
import java.util.Scanner;
//Un programa en què cal endevinar un nombre.
public class Endevina {
  //PAS 1
  //El nombre per endevinar serà el 4.
  public static final int VALOR_SECRET = 4;
  public static void main (String[] args) {
    Scanner lector = new Scanner(System.in);
    //PAS 2 i 3
    System.out.println("Comencem el joc.");
    System.out.print("Endevina el valor enter, entre 0 i 10: ");
    int valorUsuari = lector.nextInt();
    lector.nextLine();
    //PAS 4
    //Estructura de selecció doble.
    //O s'endevina o es falla.
    if (VALOR_SECRET == valorUsuari) {
      //PAS I
      //Si l'expressió avalua true, executa el bloc dins l'if.
      System.out.println("Exacte! Era " + VALOR_SECRET + ".");
    } else {
      //PAS II
      //Si l'expressió avalua false, executa el bloc dins l'else.
      System.out.println("T'has equivocat!");
    }
    System.out.println("Hem acabat el joc.");
  }
}
```

Tot seguit, la figura 2.5 mostra un esquema de quin és el flux exacte d'execució quan es posa en marxa el programa i quin paper té cada part del codi d'acord amb la definició de la sintaxi.

Figura 2.5. Esquema del flux de control d'un programa per endevinar un nombre



Repte 2: modifiqueu el programa perquè, en lloc d'un únic valor secret, n'hi hagi dos. Per guanyar, només cal encertar-ne un dels dos. La condició lògica que us caldrà ja no es pot resoldre amb una expressió composta per una única comparació. Serà més complexa.

2.3 Diversos camins: la sentència "if/else if/else"

La selecció múltiple: múltiples camins alternatius. Imatge de Wikimedia Commons

Finalment, a l'hora d'establir el flux de control d'un programa, també hi ha la possibilitat que hi hagi un nombre arbitrari de camins alternatius, no solament dos. Per exemple, imagineu un programa que, a partir de la nota numèrica d'un examen, ha d'establir quina és la qualificació de l'alumne. Per això caldrà veure dins de quin rang es troba el nombre. En qualsevol cas, els resultats possibles són més de dos.

> L'estructura de **selecció múltiple** permet controlar el fet que en complir-se un cas entre un conjunt finit de casos s'executi el conjunt d'instruccions corresponent.

2.3.1 Sintaxi i comportament

Aquest tipus de control sobre les instruccions del programa té associada la sentència `if/else if/else` (si és el cas, en aquest altre cas, si no és cap cas). Bàsicament, es tracta de la mateixa estructura que la sentència `if/else`, però amb un nombre arbitrari de blocs `if` (després del primer bloc, anomenats `else if`). En el cas del Java, la sintaxi és la següent:

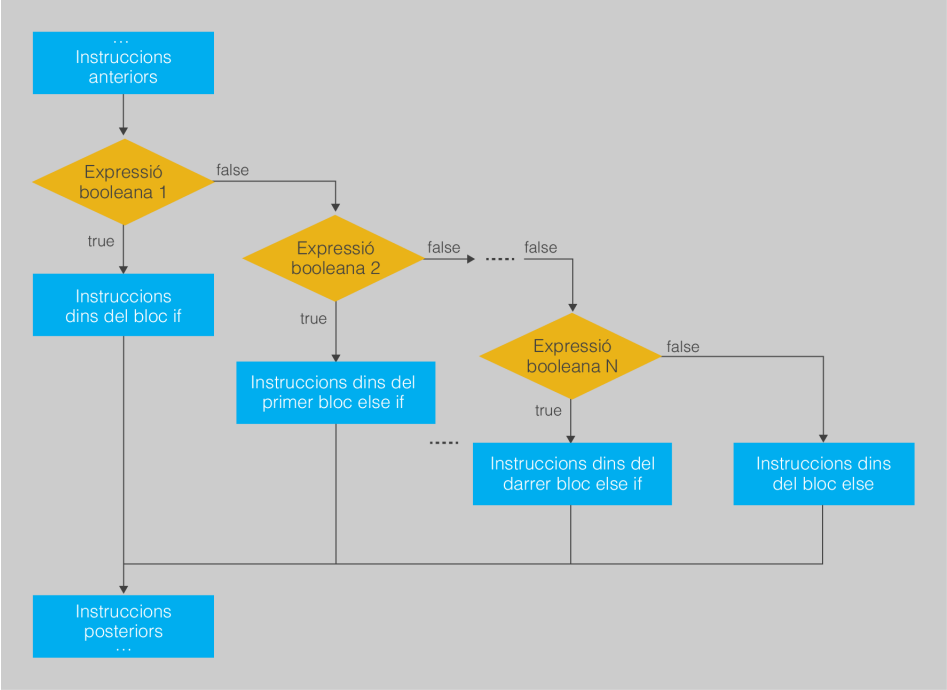
```
instruccions del programa
if (expressió booleana 1) {
  Instruccions per executar si l'expressió 1 l'avalua a true (cert) – Bloc if
} else if (expressió booleana 2) {
  Instruccions per executar si l'expressió 2 l'avalua a true (cert) – Bloc else
       if
} else if (expressió booleana 3) {
  Instruccions per executar si l'expressió 3 l'avalua a true (cert) – Bloc else
       if

...es repeteix tant cops com calgui...

} else if (expressió booleana N) {
  Instruccions per executar si l'expressió N avalua a true (cert) – Bloc else
      if
} else {
  Instruccions alternatives si totes les expressions 1...N avaluen a false (
      fals) – Bloc else
}
resta d'instruccions del programa
```

ifelse ifelse

El nom de la sentència `if/else if/else` significa bàsicament: "Si es compleix certa condició, fes això... Si no, mira si es compleix això altre... I si no, això altre...".

La figura 2.6 mostra un esquema del flux d'execució, d'acord amb el resultat d'anar avaluant cadascuna de les condicions lògiques. En aquest cas és una mica més complicat, ja que, com es pot veure, hi ha més d'una condició lògica dins de la sentència. Cada condició es va avaluant ordenadament, des de la primera fins a la darrera, i s'executarà el codi del primer cas en què l'expressió avaluï com a cert. Un cop fet això, ja no es torna a avaluar cap altra condició restant i s'ignora la resta de blocs. Si en aquest procés es dóna el cas que cap de les expressions avalua a cert, s'executaran les instruccions del bloc `else`.

FIGURA 2.6. Diagrama de flux de control d'una selecció múltiple



El punt important d'aquesta sentència és que només s'executarà un únic bloc de tots els possibles. Fins i tot en el cas que més d'una de les expressions booleanes pugui avaluar a cert, només s'executarà el bloc associat a la primera d'aquestes expressions dins de l'ordre establert en la sentència.

També és destacable el fet que el bloc `else` és opcional. Si no volem, no cal posar-lo. En aquest cas, si no es compleix cap de les condicions, no s'executa cap instrucció entre les incloses dins de la sentència.

2.3.2 Exemple: transformar avaluació numèrica a text

La manera d'usar la sentència `if/else if/else` es pot veure millor mitjançant el codi de l'exemple proposat anteriorment. Un cop més, abans de veure el codi es repassaran els passos que ha de fer el programa per dur a terme el seu objectiu i l'ordre que cal seguir.

1. Demanar que s'introdueixi la nota pel teclat.
2. Llegir-la.
3. Mostrar un text o un altre segons el rang de valors dins del qual es troba la nota:
   (a) Si és major o igual que 9 i menor o igual que 10, la nota és d'"Excel·lent".
   (b) Si és major i igual que 6,5 però estrictament menor que 9, la nota és "Notable".

(c) Si és major i igual que 5 però estrictament menor que 6,5, la nota és "Suficient".

(d) Si no és cap dels casos anteriors, la nota és de "Suspès".

Aquest cop és el pas 3 el que presenta diferents possibilitats segons si es compleixen certes condicions. En aquest cas, però, hi ha més de d'un camí (n'hi ha quatre). A més a més, cal decidir si es compleix una condició diferent per decidir quin és el camí que cal seguir entre totes les opcions. El darrer es du a terme simplement quan no es compleix cap dels anteriors. Per tant, es tracta d'una selecció múltiple.

No oblideu que en una estructura de selecció la condició lògica sempre va entre parèntesis.

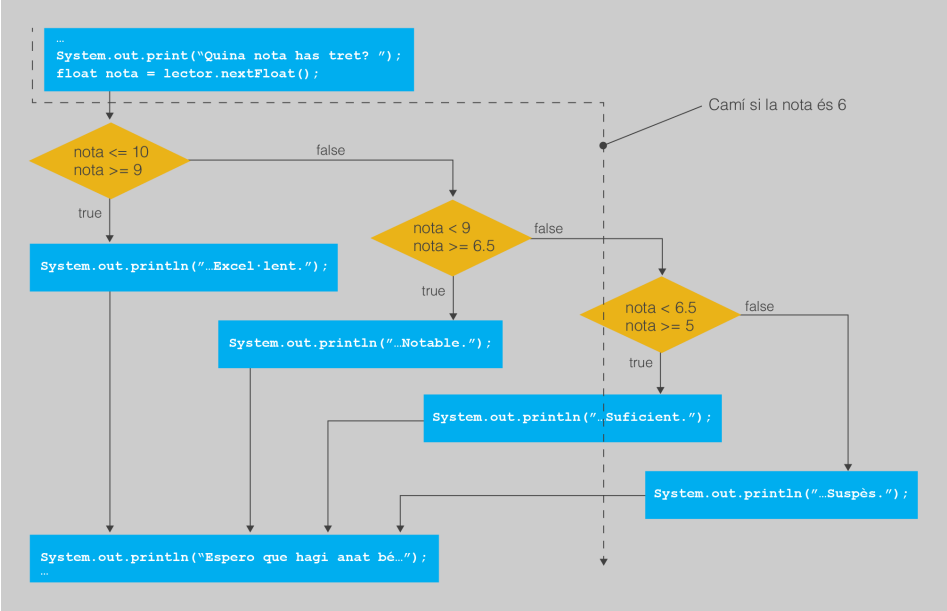
Un fet destacat en aquesta descripció de les tasques que ha de dur a terme el programa, com a mínim respecte als exemples anteriors, és que per a cada pas possible s'han de complir dues condicions alhora. El valor de la nota ha d'estar per sota de cert valor i per sobre d'un altre. L'expressió de tipus booleà que cal avaluar per veure quin camí cal seguir ha de comprovar totes dues coses: que es compleix una cosa **i** l'altra. O sigui, cal una expressió lògica basada en la conjunció de dues comparacions. Un cop més, observeu quines instruccions es corresponen a cadascun dels passos descrits. Fixeu-vos en particular en com són les condicions lògiques per a cada cas.

Compileu i executeu el programa següent per veure com la nota mostrada depèn del valor introduït:

```
1  import java.util.Scanner;
2  //Un programa que indica la nota en text a partir de la numèrica.
3  public class Avaluacio {
4    public static void main (String[] args) {
5      Scanner lector = new Scanner(System.in);
6      //PAS 1 i 2
7      System.out.print("Quina nota has tret? ");
8      float nota = lector.nextFloat();
9      lector.nextLine();
10     //PAS 3
11     //Estructura de selecció múltiple.
12     //S'entra al bloc on la condició lògica avaluï a true.
13     //Si cap no ho fa, s'entra al bloc else.
14     System.out.print("La teva nota final és ");
15     if ((nota >= 9)&&(nota <= 10)) {
16       //PAS I
17       System.out.println("Excel·lent.");
18     } else if ((nota >= 6.5)&&(nota < 9)) {
19       //PAS II
20       System.out.println("Notable.");
21     } else if ((nota >= 5)&&(nota < 6.5)) {
22       //PAS III
23       System.out.println("Aprovat.");
24     } else  {
25       //PAS IV
26       System.out.println("Suspès.");
27     }
28     System.out.println("Espero que hagi anat bé...");
29   }
30 }
```

El diagrama de flux per a aquest programa en concret es mostra tot seguit, a la figura 2.7. En aquest cas, el codi seguirà un entre cinc camins possibles, tots disjunts.

Figura 2.7. Esquema del flux de control d'un programa d'assignar la nota



Una altra possibilitat

El programa que tot just s'ha exposat analitza totes les possibilitats a l'hora d'avaluar les notes de manera molt estricta. Totes les condicions possibles són absolutament disjuntes. Mai no es pot donar el cas que dues avaluïn cert, ja que la nota introduïda pertanyerà a un i només a un dels rangs establerts. Ara bé, la selecció múltiple accepta que les condicions no ho siguin. En cas que se'n compleixi més d'una, el bloc d'instruccions que s'executarà serà el de la primera condició que ha avaluat com a `true`, per ordre d'escriptura en el codi.

Un cop es té una mica d'experiència programant, es pot jugar amb aquest fet, per plantejar aquest programa d'una altra manera:

1. Demanar que s'introdueixi la nota pel teclat.
2. Llegir-la.
3. Mostrar un text o un altre segons dins de quin rang de valors es troba la nota:
   (a) Si és major o igual que 9, la nota és "Excel·lent".
   (b) Si és major o igual que 6,5, la nota és "Notable".
   (c) Si és major o igual que 5, la nota és "Suficient".
   (d) Si no és cap dels casos anteriors, la nota és "Suspès".

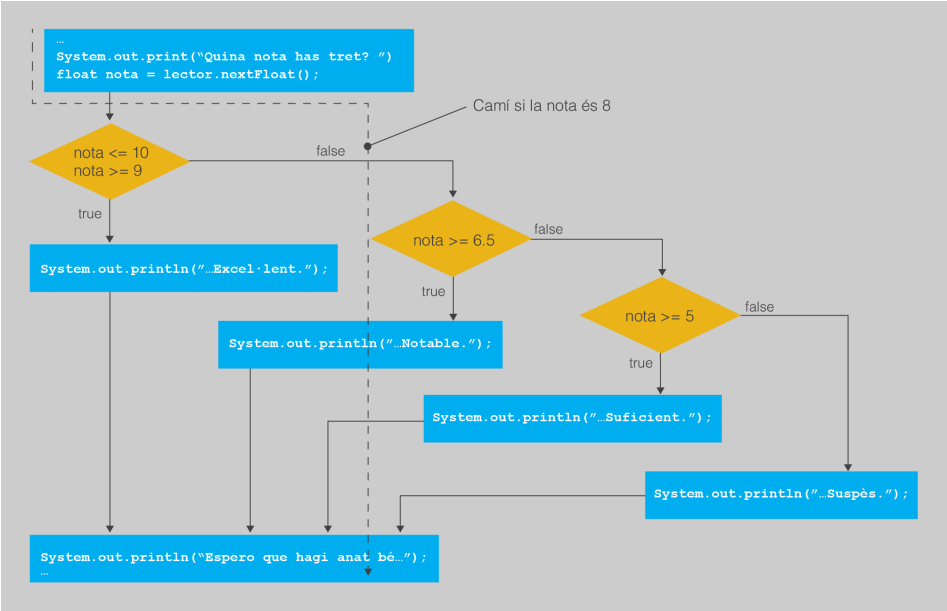
En aquest plantejament, l'ordre d'avaluació de les condicions és molt més important. Per a una nota de 9,5 es compleixen totes i cadascuna de les condicions. Però només es durà a terme l'acció relativa a la primera de les enumerades (en aquest cas, la `I`).

El codi font associat seria el següent, pràcticament idèntic a l'anterior. Compileu-lo i executeu-lo per comprovar que el comportament és el mateix.

```java
import java.util.Scanner;
//Un programa que indica la nota en text a partir de la numèrica.
public class AvaluacioSimplificat {
  public static void main (String[] args) {
    Scanner lector = new Scanner(System.in);
    //PAS 1 i 2
    System.out.print("Quina nota has tret? ");
    float nota = lector.nextFloat();
    lector.nextLine();
    //PAS 3
    //Estructura de selecció múltiple.
    //S'entra al bloc on la condició lògica avaluï a true.
    //Les condicions s'avaluen per ordre d'aparició.
    //Si cap no ho fa, s'entra al bloc else.
    System.out.print("La teva nota final és ");
    if ((nota >= 9)&&(nota <= 10)) {
      //PAS I
      System.out.println("Excel·lent.");
    } else if (nota >= 6.5) {
      //PAS II
      System.out.println("Notable.");
    } else if (nota >= 5) {
      //PAS III
      System.out.println("Aprovat.");
    } else  {
      //PAS IV
      System.out.println("Suspès.");
    }
    System.out.println("Espero que hagi anat bé...");
  }
}
```

Observeu atentament què passa si la nota és 8. En aquest cas, avaluen com a certes la segona (`nota >= 6.5`) i la tercera condició (`nota >= 5`), però el bloc d'instruccions que s'executarà és el de la segona. Això es veu clarament en el diagrama de la figura 2.8 si aneu seguint ordenadament el flux de control establert. De fet, si us hi fixeu, el diagrama en si és com el de la figura 2.7, i només varien les condicions lògiques.

FIGURA 2.8. Esquema del flux de control del programa alternatiu d'assignar la nota



2.4 Combinació d'estructures de selecció

Les sentències que defineixen estructures de selecció són instruccions com qualssevol altres dins d'un programa, si bé amb una sintaxi una mica més complicada. Per tant, res no impedeix que tornin a aparèixer dins de blocs d'instruccions d'altres estructures de selecció. Això permet crear una disposició de bifurcacions dins el flux de control per tal de dotar al programa d'un comportament complex, per comprovar si es compleixen diferents condicions d'acord amb cada situació.

2.4.1 Exemple: descompte màxim i control d'errors

Per veure un exemple en què resulta útil la combinació d'estructures de selecció, imagineu que voleu fer un parell de modificacions a l'exemple del càlcul d'un descompte. D'una banda, que hi hagi un valor màxim de descompte, de manera que si per algun motiu correspon fer un descompte per sobre d'aquest valor, només s'apliqui el màxim establert. D'altra banda, estaria bé corregir un comportament una mica estrany que ara mateix té el programa: que és capaç d'acceptar preus negatius. En aquest cas, com que és evident que l'usuari s'ha equivocat, estaria bé avisar-lo amb algun missatge.

Ara el programa hauria de fer el següent:

1. Decidir quin és el valor mínim per optar al descompte, quant es descomptarà i el valor màxim possible.
2. Demanar que s'introdueixi el preu inicial, en euros, pel teclat.
3. Llegir-lo.
4. Comprovar que el preu és correcte i no és negatiu:
   (a) Si es compleix, veure si el preu introduït és igual o superior al valor mínim per optar al descompte:
      i. Si és així, calcular el descompte.
      ii. Comprovar si el descompte supera el màxim permissible:
         A. Si és així, el descompte es redueix al màxim permissible.
      iii. Aplicar el descompte sobre el preu inicial.
   (b) Mostrar el preu final.
   (c) Si el preu era negatiu, mostrar un missatge d'error.

Així, doncs, apareixen dues noves condicions. Abans de fer res cal veure si el preu és positiu i, immediatament després de calcular el descompte, cal veure si es compleix la nova condició de si el valor és superior al màxim o no. La particularitat és que ara les diferents condicions del programa només es comproven si les anteriors es van complint, una darrere de l'altra.

El codi següent fa ús d'una combinació d'estructures de selecció per dur a terme aquesta modificació. Compileu-lo i executeu-lo. Mitjançant comentaris s'associa cada instrucció a cadascun dels passos descrits.

```
import java.util.Scanner;
//Un programa que calcula descomptes.
public class DescompteControlErrors {
  //PAS 1
  //Es fa un descompte del 8%.
  public static final float DESCOMPTE = 8;
  //Es fa descompte per compres de 100 euros o més.
  public static final float COMPRA_MIN = 100;
  //Valor del descompte màxim: 10 euros.
  public static final float DESC_MAXIM = 10;
  public static void main (String[] args) {
    Scanner lector = new Scanner(System.in);
    //PASSOS 2 i 3
    System.out.print("Quin és el preu del producte, en euros? ");
    float preu = lector.nextFloat();
    lector.nextLine();
    //PAS 4. El preu és positiu?
    if (preu > 0) {
      //PAS I
      if (preu >= COMPRA_MIN) {
        //PAS a
        float descompteFet = preu * DESCOMPTE / 100;
        //PAS b. Si el descompte supera el màxim, cal reduir-lo.
        if (descompteFet > DESC_MAXIM) {
           //PAS alfa. Reduir descompte
           descompteFet = DESC_MAXIM;
        }
        //PAS c
        preu = preu - descompteFet;
      }
      //PAS II. Es mostra el preu final.
      System.out.println("El preu final per pagar és de " + preu + " euros.");
    } else {
      //PAS III. El preu era negatiu. Cal avisar l'usuari.
      System.out.println("Preu incorrecte. És negatiu.");
    }
  }
}
```

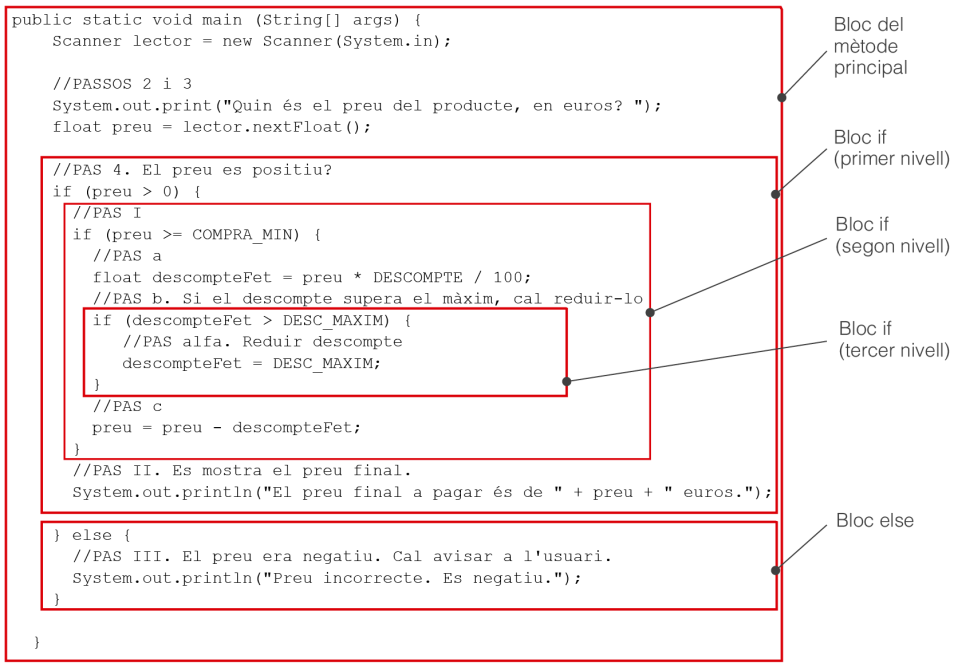
Recordeu que en Java cada bloc s'identifica per estar entre claus: {...}.

La figura 2.9 mostra un esquema de com queden distribuïts els blocs d'instruccions dins del programa i de com s'estructuren jeràrquicament en nivells de profunditat. Com es pot apreciar, quan es combinen estructures de control és especialment important sagnar el codi font per fer-lo intel·ligible i usar sempre claus per poder distingir clarament on comença i on acaba cada bloc diferent.

Un dels aspectes més importants d'aquest exemple és que el programa ha de ser capaç de comprovar si les dades que ha introduït l'usuari compleixen certes condicions i així avisar-lo de possibles errors. Moltes vegades, per aconseguir-ho cal combinar diferents estructures de selecció, de manera que només se segueix pel camí que processa les dades si es pot garantir que són correctes. De fet, tant en l'exemple d'endevinar un nombre com en el de transformar les notes també valdria la pena fer aquesta comprovació, i veure si el valor introduït està realment entre els valors permissibles (0 i 10).

Les estructures de selecció són fonamentals per poder comprovar si les dades que introdueix l'usuari són correctes abans de processar-les.

FIGURA 2.9. Blocs d'instruccions en una combinació d'estructures de selecció



Repte 3: modifiqueu els exemples d'endevinar (`Endevina`) un nombre i transformar una nota numèrica en textual (`AvaluacioSimplificat`) perquè comprovin que el valor que ha introduït l'usuari es troba dins del rang de valors correcte (entre 1 i 10). En algun cas, potser no cal combinar diverses estructures de selecció...

2.4.2 Múltiples blocs d'instruccions i àmbit de variables

En el moment que el codi font d'un programa s'organitza en diferents blocs d'instruccions, uns dins dels altres, a causa de l'aparició d'estructures de control, cal anar amb molt de compte de respectar l'àmbit de la declaració d'una variable. La millor manera de veure què pot passar si no es té cura d'això és seguint un exemple concret.

Recordeu que quan es diu "usar una variable" es vol dir en realitat "usar l'identificador associat a una variable".

Suposeu que ara voleu modificar el programa anterior perquè, a més del preu final, també mostri exactament quants euros s'han descomptat sobre el preu inicial. Com es disposa de la variable `descompteFet` ja declarada, on s'emmagatzema precisament el valor exacte del descompte, només hauria de ser qüestió de consultar-ne el valor i mostrar-lo per pantalla, tal com ja es fa amb el preu. Per tant, d'entrada, una possible modificació en el codi original seria la que es mostra tot seguit. Integreu-la en el programa original i proveu si funciona o no.

```
...
    if (preu > 0) {
      //PAS I
      if (preu >= COMPRA_MIN) {
        float descompteFet = preu * DESCOMPTE / 100;
        if (descompteFet > DESC_MAXIM) {
           descompteFet = DESC_MAXIM;
        }
        preu = preu - descompteFet;
      }
      System.out.println("El preu final per pagar és de " + preu + " euros.");
      //Modificació. Ara també mostrem el descompte... o no?
      System.out.println("S'ha fet un descompte de " + descompteFet + " euros
          .");
    } else {
      System.out.println("Preu incorrecte. És negatiu.");
    }
    ...
```

Malauradament, si intenteu compilar el programa modificat amb aquest nou codi, hi haurà un error en la nova línia on s'intenta mostrar el valor del descompte. Aquest us informa que l'identificador `descompteFet` no ha estat declarat (*cannot find symbol*) i, per tant, no pot ser usat. Però tot just unes línies abans, a la primera instrucció del bloc `if`, hi ha la declaració. Com és possible? La resposta és que, ara que ja hi ha més d'un bloc d'instruccions en el codi font del programa, no se n'ha respectat l'àmbit. Recordeu que la declaració d'una variable només té validesa des que es declara fins al delimitador de final del bloc on s'ha declarat (en aquest cas, la clau `}`).

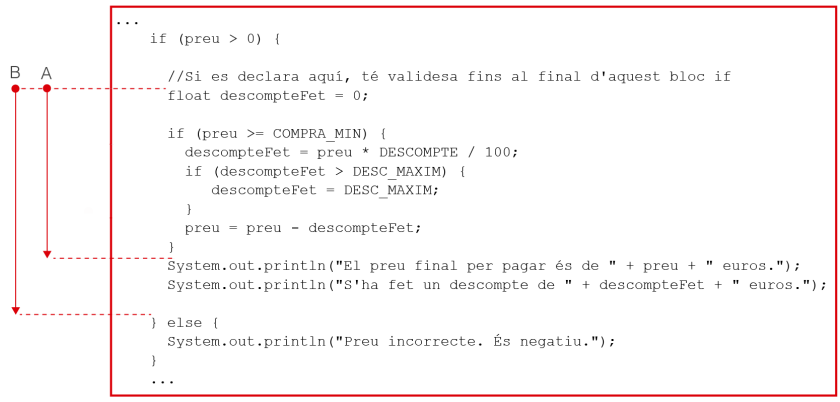
Aquesta condició es compleix per a la variable `preu`, ja que ha estat declarada al bloc de codi del programa principal, i per tant pot ser usada fins a la clau que tanca el mètode principal (és a dir, a tot el programa). Ara bé, la variable `descompteFet` ha estat declarada en el bloc `if` i, per tant, deixa de tenir validesa en acabar el bloc. Com que es consulta més enllà d'aquest bloc, és com si no s'hagués declarat mai.

Per solucionar aquest problema, la declaració s'ha de dur amb anterioritat a intentar mostrar-la per pantalla i, alhora, en el mateix bloc d'instruccions, no dins del bloc `if`. Això es podria fer com es mostra a continuació. Incorporeu el canvi al codi original i proveu si ara funciona.

```
...
    if (preu > 0) {
      //Si es declara aquí, té validesa fins al final d'aquest bloc if.
      float descompteFet = 0;
      if (preu >= COMPRA_MIN) {
        descompteFet = preu * DESCOMPTE / 100;
        if (descompteFet > DESC_MAXIM) {
           descompteFet = DESC_MAXIM;
        }
        preu = preu - descompteFet;
      }
      System.out.println("El preu final per pagar és de " + preu + " euros.");
      System.out.println("S'ha fet un descompte de " + descompteFet + " euros
          .");

    } else {
      System.out.println("Preu incorrecte. És negatiu.");
    }
    ...
```

La figura 2.10 mostra quin era l'àmbit inicial de la variable `descompteFet` en el programa original (A) i quin és el que resulta de la modificació de la línia on es declara (B) per tal de solucionar l'error.

FIGURA 2.10. Àmbits de la variable "descompteFet"

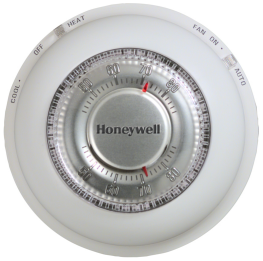


Una bona pregunta que us podeu fer és quin sentit té llavors declarar variables dins de blocs de codi que no siguin el mètode principal. La resposta és que, a escala de funcionament del programa, no hi ha cap diferència. Es podrien declarar totes a l'inici del programa principal, per garantir que mai no us trobeu fora del seu àmbit, i tot funcionaria igual. Ara bé, declarar variables tot just en el moment que s'usen té sentit amb vista a la llegibilitat del programa. Si declareu una variable prop del codi on se'n fa ús, és més senzill per a algú altre que llegeixi el codi font establir-ne el valor sense haver de repassar tot el codi des del principi. Això és especialment útil en programes llargs.

2.5 El commutador: la sentència "switch"

Hi ha una estructura de selecció una mica especial, motiu pel qual s'ha deixat per al final. El que la fa especial és que no es basa a avaluar una condició lògica composta per una expressió booleana, sinó que estableix el flux de control a partir de l'avaluació d'una expressió de tipus enter o caràcter (però mai real).

La sentència `switch` enumera, un per un, un conjunt de valors discrets que es volen tractar, i assigna les instruccions que cal executar si l'expressió avalua en cada valor diferent. Finalment, especifica què cal fer si l'expressió no ha avaluat en cap dels valors enumerats. És com un commutador o una palanca de canvis, en què s'assigna un codi per a cada valor possible que cal tractar. Aquest comportament seria equivalent a una estructura de selecció múltiple en què, implícitament, totes les condicions són comparar si una expressió és igual a cert valor. Cada branca controlaria un valor diferent. El cas final és equivalent a l'`else`.



Una sentència "switch" és un selector entre diferents valors.

De fet, en la majoria de casos, aquesta sentència no aporta res des del punt de vista del flux de control que no es pugui fer amb una selecció múltiple. Però és molt útil amb vista a millorar la llegibilitat del codi o facilitar la generació del codi del programa. Ara bé, sí que hi ha un petit detall en què aquesta

sentència aporta alguna cosa que la resta d'estructures de selecció no poden fer directament: executar de manera consecutiva més d'un bloc de codi relatiu a diferents condicions.

2.5.1 Sintaxi i comportament

La sintaxi de la sentència `switch` s'ajusta al format descrit tot seguit. Novament, es basa en una paraula clau seguida d'una condició entre parèntesis i un bloc de codi entre claus. Ara bé, la diferència respecte de les sentències vistes fins ara és que els conjunts d'instruccions assignats a cada cas independent no es distingeixen entre si per claus. Aquests es van enumerant com un conjunt d'apartats etiquetats per la paraula clau `case`, seguida del valor que es vol tractar i de dos punts. De manera opcional, es pot posar una instrucció especial que serveix de delimitador de final d'apartat, anomenada `break`. Al final de tots els apartats se'n posa un d'especial, anomenat `default`, que no ha d'acabar mai en `break`.

```
switch(expressió de tipus enter) {
  case valor1:
    instruccions si l'expressió avalua a valor1
    (opcionalment) break;
  case valor2:
    instruccions si l'expressió avalua a valor2
    (opcionalment) break;
  ...
  case valorN:
    instruccions si l'expressió avalua a valorN
    (opcionalment) break;
  default:
    instruccions si l'expressió avalua a algun valor que no és valor1...valorN
}
```

switch

El nom de la sentència `switch` vol dir bàsicament: "commuta entre aquestes accions d'acord amb un valor seleccionat".

En executar la sentència `switch` s'avalua l'expressió entre parèntesis i immediatament s'executa el conjunt d'instruccions assignat a aquell valor entre els diferents apartats `case`. Si no n'hi ha cap amb aquest valor entre els disponibles, llavors s'executa l'apartat etiquetat com a `default` (per defecte).

Cal dir que els valors numèrics no han de seguir cap ordre en concret per definir cada apartat. Si bé pot ser més polit enumerar-los per ordre creixent, això no té cap efecte sobre el comportament del programa.

La particularitat especial d'aquesta sentència la tenim en l'ús de la instrucció `break`. Aquesta indica què cal fer un cop executades les instruccions de l'apartat. Si apareix la instrucció, el programa se salta la resta d'apartats i continua amb la instrucció posterior a la sentència `switch` (després de la clau que tanca el bloc). En aquest aspecte, si tots els apartats acaben en `break`, el funcionament de `switch` és equivalent a fer:

```
if (expressió == valor 2) {
  instruccions si l'expressió avalua a valor1
} else if (expressió == valor2) {
  instruccions si l'expressió avalua a valor2
...
} else if (expressió == valorN) {
```

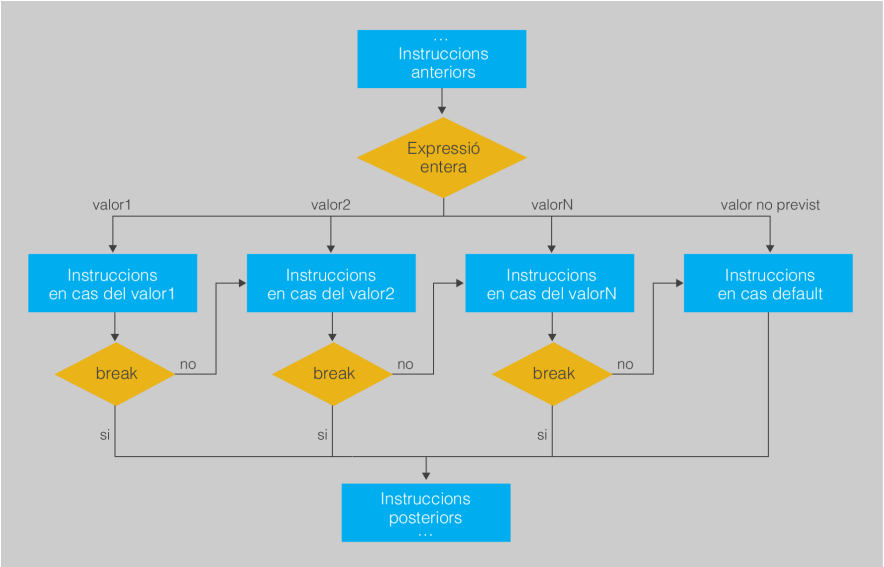
```
  instruccions si l'expressió avalua a valorN
} else {
  instruccions si l'expressió avalua a algun valor que no és valor1...valorN
}
```

Ara bé, si algun dels apartats no acaba en `break`, en executar la darrera instrucció, en lloc de saltar la resta d'apartats, el que es fa és seguir executant les instruccions de l'apartat que ve immediatament després. Així anirà fent, apartat per apartat, fins trobar-ne algun que acabi en `break`. La figura 2.11 mostra aquest comportament tan particular.

break

La paraula clau `break` és com dir "trenca, surt fora d'aquesta sentència *switch*".

FIGURA 2.11. Diagrama de flux de control d'una sentència "switch"



2.5.2 Exemple simple: diferents opcions en un menú

La millor manera d'entendre com es comporta la sentència `switch` és veient un parell d'exemples. Per començar, es mostrarà el cas més senzill, en què tots els apartats tenen una instrucció `break` al final i, per tant, és com tenir una selecció múltiple en què cada cas correspon directament a una condició clarament diferenciada amb el seu propi bloc d'instruccions independent dels altres.

Un ús molt típic de la sentència `switch` és per generar de manera senzilla menús en què l'usuari pot triar entre diferents opcions. Segons l'opció triada, s'executen directament les instruccions associades a cadascuna. Suposeu un programa en què, a partir dos nombres enters, es demana quina operació es vol fer entre aquests: sumar, restar, multiplicar o dividir.

El programa hauria de fer el següent:

1. Preguntar els dos nombres enters.
2. Llegir-los.
3. Mostrar un menú amb les quatre opcions possibles, enumerades d'1 a 4.
4. Llegir l'opció triada.

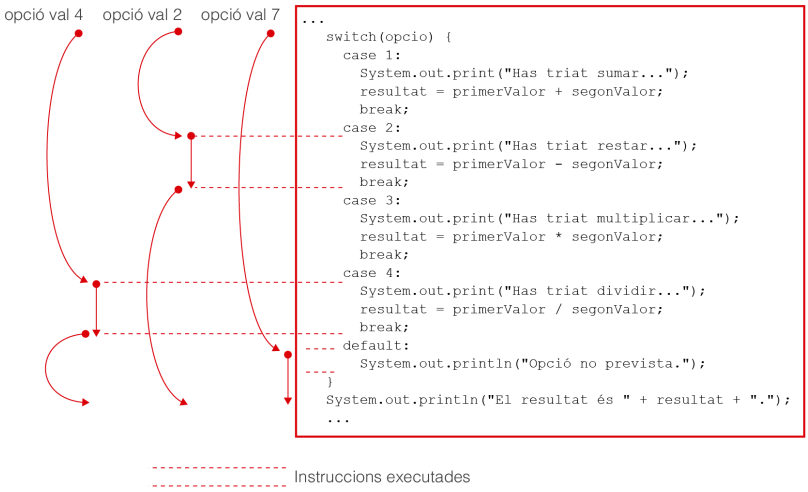
5. Segons l'opció:
   (a) Si és 1 se sumen els dos enters.
   (b) Si és 2 es resten.
   (c) Si és 3 es multipliquen.
   (d) Si és 4 es divideixen.
   (e) Si és cap altra cal informar que l'opció és incorrecta.
6. Finalment es mostra el resultat per pantalla.

El codi per fer això mitjançant la sentència `switch` podria ser el següent. Compileu-lo, executeu-lo i observeu-ne el comportament.

```
1  import java.util.Scanner;
2  //Un programa que dóna diferents opcions de càlcul, usant la sentència switch.
3  public class Opcions {
4    public static void main (String[] args) {
5      Scanner lector = new Scanner(System.in);
6      //PAS 1 i 2
7      System.out.print("Introdueix un valor enter: ");
8      int primerValor = lector.nextInt();
9      lector.nextLine();
10     System.out.print("Introdueix-ne un altre: ");
11     int segonValor = lector.nextInt();
12     lector.nextLine();
13     //PAS 3 i 4
14     System.out.println("Quina operació vols fer? ");
15     System.out.println("[1] Sumar");
16     System.out.println("[2] Restar");
17     System.out.println("[3] Multiplicar");
18     System.out.println("[4] Dividir");
19     System.out.print("Selecciona l'opció [1-4]: ");
20     int opcio = lector.nextInt();
21     lector.nextLine();
22     int resultat = 0;
23     //PAS 5
24     switch(opcio) {
25       //PAS I
26       case 1:
27         System.out.print("Has triat sumar...");
28         resultat = primerValor + segonValor;
29         break;
30       //PAS II
31       case 2:
32         System.out.print("Has triat restar...");
33         resultat = primerValor - segonValor;
34         break;
35       //PAS III
36       case 3:
37         System.out.print("Has triat multiplicar...");
38         resultat = primerValor * segonValor;
39         break;
40       //PAS IV
41       case 4:
42         System.out.print("Has triat dividir...");
43         resultat = primerValor / segonValor;
44         break;
45       //PAS V
46       default:
47         System.out.println("Opció no prevista.");
48     }
49     //PAS 6
50     System.out.println("El resultat és " + resultat + ".");
51   }
52 }
```

La figura 2.12 mostra un esquema de les instruccions executades segons alguns valors d'exemple. En lloc d'usar un diagrama de flux de control, s'usa directament el codi com a referència per fer la relació més aclaridora. Com es pot veure, l'opció `default` és especialment útil per controlar valors d'entrada no vàlids.

FIGURA 2.12. Exemple de flux de control del programa d'opcions de càlcul



2.5.3 Exemple amb propagació d'instruccions: els dies d'un mes

És el moment de veure què succeeix si falten instruccions `break` en alguns apartats. El primer que podeu provar és treure els `break` de l'exemple anterior. Veureu què passa. Cada cop que trieu una opció s'executarà aquella opció i totes les posteriors (ho podreu veure, ja que s'anirà mostrant per pantalla el missatge per a cada operació). Per tant, l'únic resultat vàlid serà el darrer i sempre estareu dividint.

Un altre exemple diferent bastant típic és el de calcular quants dies té un mes. A grans trets, el programa faria el següent:

1. Preguntar el número de mes (un valor de tipus enter).
2. Llegir-lo.
3. Segons el valor del mes:
   (a) Si és 2 cal dir que el nombre de dies és 28 o 29.
   (b) Si és 4, 6, 9 o 11 cal dir que el nombre de dies és 30.
   (c) Si és 1, 3, 5, 7, 8, 10 o 12 cal dir que el nombre de dies és 31.
   (d) Si és cap altre, cal dir que que s'ha introduït un número de mes incorrecte.

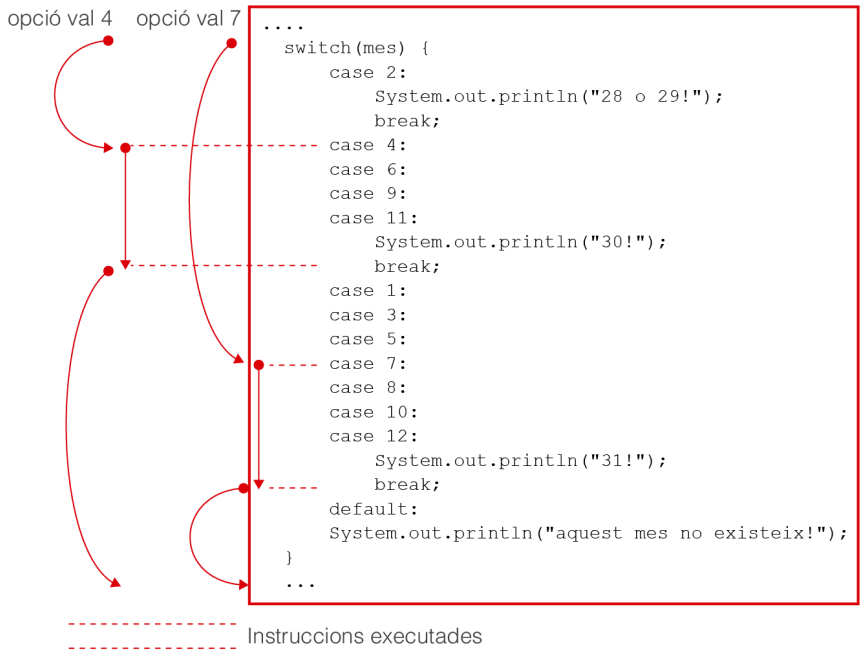
Ara bé, ara es jugarà amb la propagació d'instruccions quan manca el `break` per establir dins del codi conjunts de valors que donen el mateix resultat. En triar un valor, s'executen les instruccions del cas associat a aquell valor i, mentre no trobi

un `break`, continua executant la resta de casos consecutivament, l'un darrere de l'altre. Si un cas està buit, no fa res i salta directament al següent. Com es pot veure a l'exemple, no cal que els valors per a cada cas estiguin ordenats d'acord amb cap ordre numèric concret. Un cop més, el cas `default` serveix per controlar valors fora de rang. Comproveu que el programa fa el que se n'espera en el vostre entorn de desenvolupament.

```java
import java.util.Scanner;

//A veure quants dies té un mes...
public class DiesDelMes {

  public static void main (String[] args) {
    Scanner lector = new Scanner(System.in);

    //PAS 1 i 2
    System.out.print("Quin número de mes vols analitzar [1–12]? ");
    int mes = lector.nextInt();
    lector.nextLine();

    //PAS 3
    System.out.print("Els dies d'aquest mes deuen ser...");
    switch(mes) {
        //PAS I
        case 2:
            System.out.println("28 o 29!");
            break;
        //PAS II
        case 4:
        case 6:
        case 9:
        case 11:
            System.out.println("30!");
            break;
        //PAS III
        case 1:
        case 3:
        case 5:
        case 7:
        case 8:
        case 10:
        case 12:
            System.out.println("31!");
            break;
        default:
        System.out.println("Aquest mes no existeix!");
    }

  }

}
```

La figura 2.13 mostra un parell d'exemples de quin és el flux de control de la sentència `switch` per a certs valors. Tingueu en compte que el programa comença pel valor que coincideix amb el resultat d'avaluar l'expressió i que, un cop allà, va avançant de manera seqüencial per tots els casos, hi hagi el que hi hagi, fins a trobar una sentència `break`.

Figura 2.13. Exemple de flux de control per al programa dels dies del més. Una sentència "switch" amb casos sense "break"



Finalment, cal dir que, com ja s'ha esmentat, no hi ha res que faci la sentència `switch` que no pugui fer una selecció múltiple. L'elecció de `switch` se sol fer exclusivament per motius de claredat del codi. Compareu el codi anterior amb el següent, equivalent. Tot seguit, comproveu que fa el mateix.

```
1  import java.util.Scanner;
2
3  //A veure quants dies té un mes...
4  public class DiesDelMesIf {
5
6    public static void main (String[] args) {
7      Scanner lector = new Scanner(System.in);
8
9      System.out.print("Quin número de mes vols analitzar [1-12]? ");
10     int mes = lector.nextInt();
11     lector.nextLine();
12
13     System.out.print("Els dies d'aquest mes deuen ser...");
14     if (mes == 2) {
15       System.out.println("28 o 29!");
16     } else if ((mes == 4)||(mes == 6)||(mes == 9)||(mes == 11)) {
17       System.out.println("30!");
18     } else if ((mes == 1)||(mes == 3)||(mes == 5)||(mes == 7)||(mes == 8)||(mes
           == 10)||(mes == 12)) {
19       System.out.println("31!");
20     } else {
21       System.out.println("Aquest mes no existeix!");
22     }
23   }
24
25 }
```

Decidir quin és més pràctic és decisió vostra. De totes maneres, no sol ser bona idea tenir expressions tan llargues que se surtin del camp de visió en l'editor emprat.

2.6 Control d'errors en l'entrada bàsica mitjançant estructures de selecció

Les estructures de selecció són especialment útils com a mecanisme per controlar si les dades que ha introduït un usuari són correctes abans d'usar-les dins del programa. Un cop ja sabeu com funcionen, és un bon moment per ampliar les explicacions sobre l'entrada bàsica de dades per teclat, de manera que sigui possible controlar si el valor que s'ha introduït encaixa amb el tipus de dada esperat per a cada instrucció de lectura.

Si es parteix que l'extensió per llegir dades de teclat s'ha inicialitzat anteriorment com sempre:

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

Abans de llegir qualsevol dada introduïda pel teclat, és possible usar les instruccions de la taula 2.2 per establir si la dada que tot just s'ha escrit pel teclat és d'un tipus concret o no. Quan s'usen, l'usuari pot introduir dades pel teclat (el programa s'atura) i s'avalua si el que ha escrit pertany a un tipus de dada o un altre. Aquestes instruccions es consideren expressions booleanes que avaluen com a `true` si el tipus concorda i com a `false` si no és el cas.

Un cop usada aquesta instrucció, només es podran llegir les dades correctament usant la instrucció `lector.next...` corresponent al mateix tipus si ha avaluat com a `true`. Si s'intenta després que hagi avaluat `false`, l'error d'execució està garantit.

Taula 2.2. Instruccions per a la lectura de dades pel teclat

Instrucció	Tipus de dada controlada
`lector.hasNextByte()`	`byte`
`lector.hasNextShort()`	`short`
`lector.hasNextInt()`	`int`
`lector.hasNextLong()`	`long`
`lector.hasNextFloat()`	`float`
`lector.hasNextDouble()`	`double`
`lector.hasNextBoolean()`	`boolean`

La instrucció associada a una cadena de text no existeix, ja que qualsevol dada escrita pel teclat sempre es pot interpretar com a text. Per tant, la lectura mai no pot donar error.

Per exemple, el programa pensat per endevinar un nombre es podria fer tal com mostra el codi següent. Pareu atenció al comportament de les combinacions d'estructures de selecció. Fixeu-vos que hi ha tres camins disjunts possibles fruit d'aquest fet: si la dada escrita no és un enter, si ho és però no s'encerta el nombre, i finalment, si ho és i sí que s'encerta. Compileu-lo i executeu-lo per

veure que, efectivament, ara es pot detectar que les dades introduïdes no són del tipus correcte.

```
import java.util.Scanner;
//Un programa en què cal endevinar un nombre.
public class EndevinaControlErrorsEntrada {
  //PAS 1
  //El nombre per endevinar serà el 4.
  public static final int VALOR_SECRET = 4;
  public static void main (String[] args) {
    Scanner lector = new Scanner(System.in);
    System.out.println("Comencem el joc.");
    System.out.print("Endevina el valor enter, entre 0 i 10: ");
    boolean tipusCorrecte = lector.hasNextInt();
    if (tipusCorrecte) {
      //S'ha escrit un enter correctament. Ja es pot llegir.
      int valorUsuari = lector.nextInt();
      lector.nextLine();
      if (VALOR_SECRET == valorUsuari) {
        System.out.println("Exacte! Era " + VALOR_SECRET + ".");
      } else {
        System.out.println("T'has equivocat!");
      }
      System.out.println("Hem acabat el joc.");
    } else {
      //No s'ha escrit un enter.
      System.out.println("El valor introduït no és un enter.");
    }
  }
}
```

Repte 4: apliqueu el mateix tipus de control sobre les dades de l'entrada a l'exemple del càlcul del descompte.

2.7 Solucions dels reptes proposats

Repte 1:

```
import java.util.Scanner;
public class Penalitzacio {
  public static final float PENALITZACIó = 2;
  //Es fa descompte per compres de 100 euros o més.
  public static final float COMPRA_MIN = 30;
  public static void main (String[] args) {
    Scanner lector = new Scanner(System.in);
    System.out.print("Quin és el preu del producte en euros? ");
    float preu = lector.nextFloat();
    lector.nextLine();
    if (preu < COMPRA_MIN) {
      //PAS I
      preu = preu + PENALITZACIó;
    }
    System.out.println("El preu final per pagar és de " + preu + " euros.");
  }
}
```

Repte 2:

```
import java.util.Scanner;
public class DosSecrets  {
  //PAS 1
  //El nombres per endevinar seran el 4 i el 7.
  public static final int VALOR_SECRET_UN = 4;
  public static final int ALTRE_VALOR_DOS = 7;
  public static void main (String[] args) {
    Scanner lector = new Scanner(System.in);
    //PAS 2 i 3
    System.out.println("Comencem el joc.");
    System.out.print("Endevina el valor enter, entre 0 i 10: ");
    int valorUsuari = lector.nextInt();
    lector.nextLine();
    //PAS 4
    //Estructura de selecció doble.
    //O s'endevina o es falla.
    if ((VALOR_SECRET_UN == valorUsuari)||(ALTRE_VALOR_DOS == valorUsuari)) {
      //PAS I
      //Si l'expressió avalua true, executa el bloc dins l'if.
      System.out.println("Exacte! Era " + valorUsuari + ".");
    } else {
      //PAS II
      //Si l'expressió avalua false, executa el bloc dins l'else.
      System.out.println("T'has equivocat!");
    }
    System.out.println("Hem acabat el joc.");
  }
}
```

Repte 3:

```java
import java.util.Scanner;
public class EndevinaControlValors {
  public static final int VALOR_SECRET = 4;
  public static void main(String[] args) {
    Scanner lector = new Scanner(System.in);
    System.out.println("Comencem el joc.");
    System.out.print("Endevina el valor enter, entre 0 i 10: ");
    int valorUsuari = lector.nextInt();
    lector.nextLine();
    if ((valorUsuari < 0) || (valorUsuari > 10)){
      System.out.println("Error: el valor ha d'estar entre 0 i 10.");
    }else{
      if (VALOR_SECRET == valorUsuari) {
        System.out.println("Exacte! Era " + VALOR_SECRET + ".");
      }else{
        System.out.println("T'has equivocat!");
      }
      System.out.println("Hem acabat el joc.");
    }
  }
}
```

```java
import java.util.Scanner;
public class AvaluacioControlValors {
  public static void main(String[] args) {
    Scanner lector = new Scanner(System.in);
    System.out.print("Quina nota has tret? ");
    float nota = lector.nextFloat();
    lector.nextLine();
    if ((nota >= 9) && (nota <= 10)) {
      System.out.println("La teva nota final és Excel·lent.");
    } else if ((nota >= 6.5) && (nota < 9)) {
      System.out.println("La teva nota final és Notable.");
    } else if ((nota >= 5) && (nota < 6.5)) {
      System.out.println("La teva nota final és Aprovat.");
    } else if ((nota >= 0) && (nota < 5)) {
      System.out.println("La teva nota final és Suspès.");
    } else {
      System.out.println("El valor escrit no està entre 0 i 10!");
    }
  }
}
```

Repte 4:

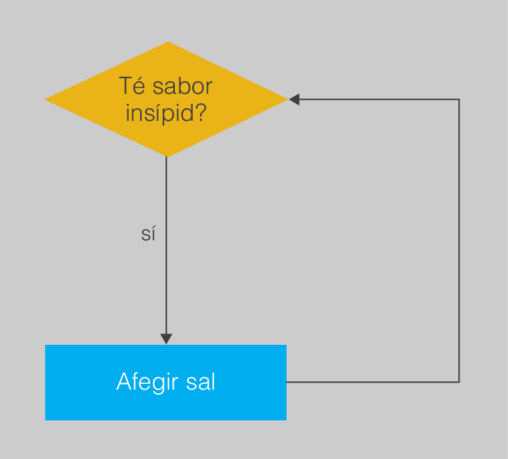
```java
import java.util.Scanner;
public class DescompteControlErrorsEntrada {
  //Es fa un descompte del 8%.
  public static final float DESCOMPTE = 8;
  //Es fa descompte per compres de 100 euros o més.
  public static final float COMPRA_MIN = 100;
  //Valor del descompte màxim: 10 euros.
  public static final float DESC_MAXIM = 10;
  public static void main (String[] args) {
    Scanner lector = new Scanner(System.in);
    System.out.print("Quin és el preu del producte en euros? ");
    boolean tipusCorrecte = lector.hasNextFloat();
    //El tipus és correcte?
    if (tipusCorrecte) {
      float preu = lector.nextFloat();
      lector.nextLine();
      if (preu > 0) {
        if (preu >= COMPRA_MIN) {
          float descompteFet = preu * DESCOMPTE / 100;
          if (descompteFet > DESC_MAXIM) {
           descompteFet = DESC_MAXIM;
          }
          preu = preu - descompteFet;
        }
        System.out.println("El preu final per pagar és de " + preu + " euros.")
            ;
      } else {
        System.out.println("Preu incorrecte. És negatiu.");
      }
    } else {
      //No s'ha escrit un enter.
      System.out.println("El valor introduït no és un enter.");
    }
  }
}
```

3. Estructures de repetició

Una estructura de repetició permet executar les mateixes instruccions diverses vegades. Imatge de Wikimedia Commons

Fins al moment, tots els programes que heu estudiat, ja fossin més complexos o més senzills, tenien una característica en comú. El seu flux de control avançava inexorablement cap a la instrucció final del mètode principal sense possibilitat de recular. Sigui quin sigui el camí alternatiu per on s'hi ha arribat, un cop una instrucció ha estat executada, ja no es tornarà a executar mai més. Aquest fet contrasta amb el comportament que podeu observar en moltes activitats que fem diàriament, com per exemple, quan posem sal al menjar que cuinem. Podem repetir diverses vegades el procés de posar sal i tastar-ho fins que el gust sigui el que volem (i llavors ja deixem de posar sal i de tastar). Aquest procés es representa com un diagrama de flux de control a la figura 3.1.

Figura 3.1. Una acció es repeteix mentre es compleixi una condició



Això també passa en programes del vostre ordinador, en què es poden repetir accions sense problemes: obrir i desar diferents fitxers, connectar-se a diferents pàgines web, etc. Aquí és on entra en joc el tipus següent d'estructura de control, també molt important dins el codi d'un programa.

> Les **estructures de repetició** o iteratives permeten repetir una mateixa seqüència d'instruccions diverses vegades, mentre es compleixi una certa condició.

En el seu aspecte general, tenen molt de semblant a una estructura de selecció. Hi ha una sentència especial que cal escriure al codi font, unida a una condició lògica i un bloc de codi (en aquest cas, sempre en serà només un). Però en aquest cas, mentre la condició lògica sigui certa, tota la seqüència d'instruccions es va executant repetidament. En el moment en què es deixa de complir la condició, es deixa d'executar el bloc de codi i ja se segueix amb la instrucció que hi ha després de la sentència de l'estructura de repetició.

Anomenem **bucle** o cicle el conjunt d'instruccions que s'ha de repetir un cert nombre de vegades, i anomenem **iteració** cada execució individual del bucle.

Com passa amb les estructures de selecció, hi ha diferents tipus de sentències, cadascuna amb les seves particularitats. Normalment, la diferència principal està vinculada al moment en què s'avalua la condició per veure si cal tornar a repetir el bloc d'instruccions o no. Al llarg d'aquest apartat, les anireu veient amb detall.

3.1 Control de les estructures repetitives

Com passava amb les estructures de selecció, les estructures de repetició no tenen sentit si no és que la condició lògica depèn d'alguna variable que pugui veure modificat el seu valor per a diferents execucions. En cas contrari, la condició sempre valdrà el mateix per a qualsevol execució possible i usar una estructura de control no serà gaire útil. En aquest cas, si la condició sempre és `false`, mai no s'executa el bucle, per la qual cosa és codi inútil. Però, per a les estructures de repetició, si la condició sempre és `true` el problema és molt més greu. Com que absolutament sempre que s'avalua si cal fer una nova iteració, la condició es compleix, el bucle no es deixa mai de repetir. El programa mai no acabarà!

Un **bucle infinit** és una seqüència d'instruccions dins d'un programa que itera indefinidament, normalment perquè s'espera que s'assoleixi una condició que mai no s'arriba a produir.

Bucle infinit

Un bucle infinit és un error semàntic de programació. Si n'hi ha, el programa compilarà perfectament de tota manera.

Quan això succeeix, el programa no es pot aturar de cap altra manera que no sigui tancant-lo directament des del sistema operatiu (per exemple, tancant la finestra associada o amb alguna seqüència especial d'escapament del teclat) o usant algun mecanisme de control de l'execució del vostre programa que ofereixi l'IDE usat.

Tenint en compte el perill que un programa s'acabi executant indefinidament, forçosament dins de tot bucle hi ha d'haver instruccions que manipulin variables el valor de les quals permeti controlar la repetició o el final del bucle. Aquestes variables s'anomenen **variables de control**.

Garantir l'assignació correcta de valors de les variables de control d'una estructura repetitiva és extremadament important. Quan genereu un programa, és imprescindible que el codi permeti que, en algun moment, la variable canviï de valor, de manera que la condició lògica es deixi de complir. Si això no és així, tindreu un bucle infinit.

Normalment, les variables de control dins d'un bucle es poden englobar dins d'algun d'aquests tipus de comportament:

- **Comptador:** una variable de tipus enter que va augmentant o disminuint, indicant de manera clara el nombre d'iteracions que caldrà fer.

- **Acumulador:** una variable en què es van acumulant directament els càlculs que es volen fer, de manera que en assolir cert valor es considera que ja no cal fer més iteracions. Si bé s'assemblen als comptadors, no són ben bé el mateix.

- **Semàfor:** una variable que serveix com a interruptor explícit de si cal seguir fent iteracions. Quan ja no en volem fer més, el codi simplement s'encarrega d'assignar-li el valor específic que servirà perquè la condició avaluï `false`.

Els semàfors també s'anomenen popularment *flags* (banderoles d'avís).

Evidentment, una condició lògica pot prendre la forma d'una expressió molt complexa, però per començar n'hi ha prou de conèixer aquests tres models. De vegades, les diferències poden ser subtils, i per això tampoc no cal amoïnar-se gaire si no es té clar a quin tipus pertany exactament la variable de control. Igualment, tenir presents aquests tres papers bàsics pot ser d'ajut amb vista a enfocar la programació d'una estructura de selecció.

3.2 Repetir si es compleix una condició: la sentència "while"

L'estructura de repetició per antonomàsia és la codificada mitjançant la sentència `while`. Aquesta existeix d'una manera o una altra en la majoria de llenguatges de programació. La seva particularitat és que pràcticament qualsevol codi basat en una estructura de repetició es pot representar usant aquesta sentència. Amb aquesta ja en tindríeu prou per tractar gairebé qualsevol cas possible. Això succeeix amb contraposició de les estructures de selecció, en què cada tipus de sentència ofereix diferents possibilitats.

La sentència **while** permet repetir l'execució del bucle mentre es verifiqui la condició lògica. Aquesta condició es verifica al principi de cada iteració. Si la primera vegada, tot just quan s'executa la sentència per primer cop, ja no es compleix, no s'executa cap iteració.

3.2.1 Sintaxi i estructura

Per dur a terme aquest tipus de control sobre les iteracions d'un bucle, la sintaxi d'aquesta sentència en el llenguatge Java és la següent:

```
1 while (expressió booleana) {
2   Instruccions per executar dins del bucle
3 }
```

while

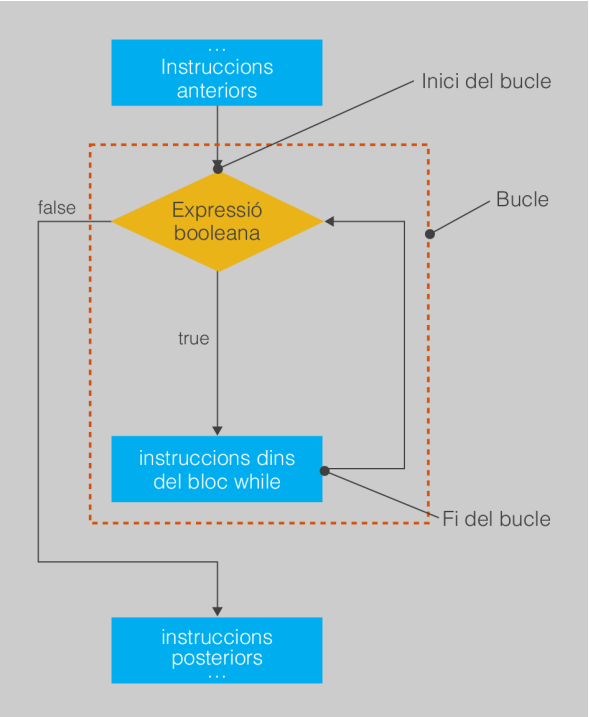
El nom de la sentència `while` bàsicament significa: "Mentre això es compleixi, fes això altre...".

Com podeu veure, el seu format és molt semblant a la sentència `if`, simplement canviant la paraula clau per `while`. Com ja passava amb les diferents sentències

dins les estructures de selecció, si entre els parèntesis es posa una expressió que no avalua un resultat de tipus booleà, hi haurà un error de compilació.

La figura 3.2 mostra un diagrama del flux de control d'aquesta sentència, i estableix l'ordre sota el qual s'avalua la condició lògica representada per l'expressió booleana amb vista a establir si cal executar o no una iteració del bucle.

Figura 3.2. Diagrama de flux de control d'una sentència "while"



La millor manera de veure com funciona i constatar-ne la utilitat és mitjançant alguns exemples concrets.

3.2.2 Exemple: estalviar-vos d'escriure el mateix molts cops

La utilitat més directa d'una estructura de repetició és que, en cas que vulgueu executar exactament les mateixes instruccions moltes vegades, no les hàgiu d'escriure una pila de cops. Per exemple, suposeu que voleu escriure una línia horitzontal a la pantalla exactament amb quaranta caràcters '-'. Evidentment, res us impedeix pitjar exactament 40 vegades la tecla, ni una més ni una menys, i ja està. Però, també ho faríeu si cal escriure'n cent, mil o fins i tot més? Fer-ho amb una estructura de repetició és més còmode.

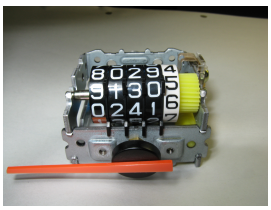
Per assolir aquesta fita, cal que el bucle iteri un cert nombre de vegades. Per fer-ho, us caldrà una variable de control de tipus comptador, que serveixi per comptar quantes iteracions hi ha fetes. O sigui, cada cop que es produeix una iteració, sumar-hi 1. Un cop aquesta variable supera el valor que volem, ja no cal fer més iteracions. Per tant, la condició lògica ha de controlar si el valor de la variable és inferior al nombre d'iteracions que cal fer.

Un possible codi que il·lustra això seria el següent. Compileu-lo i executeu-lo per veure què fa:

```
//Un programa que escriu una línia amb 100 caràcters '-'.
public class Linia {
  public static void main (String[] args) {
    //Inicialitzem un comptador
    int i = 0;
    //Ja hem fet això 100 cops?
    while (i < 100) {
      System.out.print('-');
      //Ho hem fet un cop, sumem 1 al comptador
      i = i + 1;
    }
    //Forcem un salt de línia
    System.out.println();
  }
}
```

Hi ha un parell de qüestions del codi anterior que val la pena comentar. D'una banda, la nomenclatura de la variable de control de tipus comptador. Evidentment, es pot usar l'identificador que es vulgui, però és comú usar lletres individuals (`i`, `j`, `k`...) per poder identificar ràpidament quina variable dins un bucle és un comptador i facilitar-ne la comprensió. D'altra banda, a l'hora d'usar comptadors, se sol començar des del valor 0 en lloc d'1.

Per veure amb més detall què està passant dins un bucle, en lloc d'un esquema basat en el diagrama de flux de control usarem una taula. Cada fila correspondrà a una iteració i a cada columna s'especificarà el valor de les diferents variables dins del bucle, de manera que es pot veure com evolucionen a cada iteració. Normalment, es fa èmfasi en el valor a l'inici del bucle, tot just abans d'avaluar la condició lògica, i al final del bucle, tot just després de la darrera instrucció, per veure quines han estat les modificacions en executar les instruccions contingudes. Evidentment, els valors de les variables a l'inici d'una iteració han de ser exactament els mateixos que tot just a l'inici de la següent.



Un comptador controla el nombre d'iteracions que cal fer. Imatge de Wikimedia Commons

La taula 3.1 mostra un resum de com canvia el valor de la variable de control i de quin és el resultat d'avaluar la condició lògica a cada iteració per a l'exemple actual.

Col·loquialment, per dir que ja no cal fer més iteracions se sol dir que "se surt del bucle".

Taula 3.1. Evolució del bucle per escriure 100 caràcters "-"

Iteració	Inici del bucle		Fi del bucle
	'i' val	Condició val	'i' val
1	0	(0 < 100), `true`	1
2	1	(1 < 100), `true`	2
3	2	(2 < 100), `true`	3
...			
99	98	(98 < 100), `true`	99
100	99	(99 < 100), `true`	100
101	100	(100 < 100), `false`	Ja hem sortit del bucle

Repte 1: en molts casos, el nombre de repeticions no serà fix, sinó que podrà dependre d'una altra variable. Modifiqueu l'exemple perquè primer pregunti a l'usuari quants caràcters '-' vol escriure per pantalla, i llavors els escrigui. Quan proveu el programa, no introduïu un nombre gaire alt!

3.2.3 Exemple: aprofitar un comptador

La utilitat de les estructures de selecció va molt més enllà de ser una manera senzilla d'estalviar-se escriure el mateix codi moltes vegades. Amb aquestes es poden executar instruccions de maneres que no es poden replicar simplement fent moltes vegades "copiar" i "enganxar" dins del codi font. I és que, a l'hora de la veritat, un comptador sol tenir un paper més important que simplement comptar el nombre d'iteracions que s'han fet. El valor que emmagatzema també es pot usar dins del bucle per dur a terme altres propòsits.

Per exemple suposeu que voleu imprimir per pantalla tots els valors de la taula de multiplicar d'un nombre enter qualsevol, des de l'1 fins al 10. Sense aturar-nos gaire a pensar, aquest programa hauria de fer:

- 1. Demanar que s'introdueixi un nombre pel teclat.
- 2. Llegir-lo.
- 3. Mostrar el resultat de multiplicar el nombre per 1.
- 4. Mostrar el resultat de multiplicar el nombre per 2.
- ...
- 12. Mostrar el resultat de multiplicar el nombre per 10.

D'aquesta descripció es desprèn que del pas 3 al 12 s'estan repetint ordres, que segurament es poden reemplaçar per una estructura de repetició. Concretament hi ha 10 passes entre la 3 i la 12, i a cadascuna el que es fa és més o menys el mateix: multiplicar el valor introduït per un valor que cada vegada s'incrementa en 1. És a dir, cal un comptador. Però aquest comptador ara, a més a més de garantir que es fan 10 iteracions, també s'usa per fer càlculs.

D'acord amb això, un possible codi podria ser el següent. Proveu que funciona.

```
1  import java.util.Scanner;
2  //Un programa que mostra la taula de multiplicar d'un nombre.
3  public class TaulaMultiplicar {
4    public static void main (String[] args) {
5      //S'inicialitza la biblioteca.
6      Scanner lector = new Scanner(System.in);
7      //Pregunta el nombre.
8      System.out.print("Quina taula de multiplicar vols? ");
9      int taula = lector.nextInt();
10     lector.nextLine();
11     //El comptador servirà per fer càlculs.
12     int i = 1;
```

```
13      while (i <= 10) {
14        int resultat = taula*i;
15        System.out.println(taula + " * " + i + " = " + resultat);
16        i = i + 1;
17      }
18      System.out.println("Aquesta ha estat la taula del " + taula);
19    }
20  }
```

Fixeu-vos que, tot i que normalment en els comptadors es comença a comptar pel 0, en aquest cas cal començar per l'1, ja que dins dels càlculs que cal fer, el primer pas requereix que la multiplicació sigui per 1. La taula 3.2 mostra l'evolució de les variables a cada iteració, suposant que es faci la taula de multiplicar del 5.

Taula 3.2. Evolució del bucle per escriure la taula de multiplicar del 5

Iteració	**Inici del bucle**		**Fi del bucle**	
	'i' val	**Condició val**	**'resultat' val**	**'i' val**
1	1	(1 $\leq$ 10), `true`	5*1, 5	2
2	2	(2 $\leq$ 10), `true`	5*2, 10	3
3	3	(3 $\leq$ 10), `true`	5*3, 15	4
...				
9	9	(9 $\leq$ 10), `true`	5*19, 45	10
10	10	(10 $\leq$ 10), `true`	5*10, 50	11
11	11	(11 $\leq$ 10), `false`	Ja hem sortit del bucle	

Repte 2: un comptador tant pot començar a comptar des de 0 i anar pujant, com des del final i anar disminuint com un compte enrere. Modifiqueu aquest programa perquè la taula de multiplicar comenci mostrant el valor per a 10 i vagi baixant fins a l'1.

3.2.4 Exemple: no sempre se suma u

A l'hora d'usar variables de control amb la funció de comptador, és molt típic que l'increment sigui d'u en u. Això és lògic, ja que és la manera més simple de comptar quantes iteracions s'han fet. Per fer 10 iteracions, si bé és el mateix sumar d'u en u i comparar amb 10 o sumar de dos en dos i comparar amb 20, és clar que el primer cas és molt més entenedor. Ara bé, quan els comptadors també tenen dins del bucle un paper per fer certes operacions, s'ha de tenir present que es poden incrementar o modificar en qualsevol valor.

Per exemple suposeu un programa que permeti sumar tots els valors enters múltiples de 3 dins d'un interval entre 0 i un valor qualsevol. Una primera aproximació podria ser:

1. Preguntar el límit de l'interval.
2. Llegir-lo.

3. Anar mirant un per un tots els valors dins l'interval, de 0 al límit estipulat.
   (a) Si és múltiple de tres, es va acumulant en una variable en què es desa el resultat final.
   (b) Si no ho és, s'ignora.
4. Un cop recorregut tot l'interval (superat el valor límit), es mostra el resultat acumulat.

Per veure si un nombre és múltiple de 3 cal comprovar si el seu mòdul (%) 3 és igual a 0.

El codi font que faria això podria ser el següent. Aquest codi té la particularitat –i per això val la pena estudiar-lo– que combina estructures de repetició amb estructures de selecció. Ara bé, el format de l'estructura de repetició no és gaire diferent dels exemples anteriors: un comptador que va de 0 a un valor concret per controlar quants cops itera el bucle. Per fer-lo més entenedor quan l'executeu, cada cop que es troba un múltiple de 3 el mostra per pantalla.

```
1  import java.util.Scanner;
2  //Anem a sumar un seguit de múltiples de tres.
3  public class SumarMultiplesTres {
4    public static void main (String[] args) {
5      Scanner lector = new Scanner(System.in);
6      //PAS 1 i 2
7      System.out.print("Fins a quin valor vols sumar múltiples de 3? ");
8      int limit = lector.nextInt();
9      lector.nextLine();
10     int resultat = 0;
11     //PAS 3: Anar mirant els valors un per un. Es comença pel 0.
12     int i = 0;
13     while (i <= limit) {
14       //PAS a: És múltiple de tres?
15       if ( (i%3) == 0) {
16         System.out.println("Afegim " + i +"...");
17         resultat = resultat + i;
18       }
19       //PAS 3: Seguir anar mirant els valors un per un.
20       i = i + 1;
21     }
22     System.out.println("El resultat final és " + resultat + ".");
23   }
24 }
```

Ara bé, hi ha una manera de simplificar aquest programa. Seria molt més senzill si, en lloc d'anar provant un per un tots els valors dins del rang, el comptador sempre tingui únicament valors múltiples de 3. En aquest cas, només caldria anar sumant els valors sense necessitar cap estructura de selecció. Partint d'aquest fet, quins valors hauria d'anar prenent el comptador? Doncs 0, 3, 6, 9, 12, 15, etc. Si pensem una mica en aquesta seqüència de nombres, es veu que hi ha prou que el comptador s'incrementi de tres en tres.

Per tant, el codi del bucle es podria reemplaçar pel següent:

```
1  ...
2      while (i <= limit) {
3        System.out.println("Afegim " + i +"...");
4        resultat = resultat + i;
5        //Incrementem de tres en tres.
6        i = i + 3;
7      }
8      ...
```

Una variable de control amb el paper de comptador es pot modificar de qualsevol manera que es consideri escaient. Ara bé, sempre cal garantir que a cada iteració us apropeu a la condició lògica `false`, i eviteu així un possible bucle infinit.

La taula 3.3 mostra l'evolució de les variables del programa a cada iteració del bucle, suposant que es vol fer càlcul fins al nombre 20. La prova que es fan 7 iteracions és que, si executeu el programa, es pot veure com s'executa exactament 7 vegades la instrucció `System.out.println...`.

Taula 3.3. Evolució del bucle per escriure la suma dels valors enters múltiples de 3

Iteració	Inici del bucle		Fi del bucle	
	'i' val	Condició val	'resultat' val	'i' val
1	0	$(0 \leq 20)$, `true`	0+0, 0	3
2	3	$(3 \leq 20)$, `true`	0+3, 3	6
3	6	$(6 \leq 20)$, `true`	3+6, 9	9
4	9	$(9 \leq 20)$, `true`	9+9, 18	12
5	12	$(12 \leq 20)$, `true`	18+12, 30	15
6	15	$(15 \leq 20)$, `true`	30+15, 45	18
7	18	$(18 \leq 20)$, `true`	45+18, 63	21
8	11	$(21 \leq 20)$, `false`	Ja hem sortit del bucle	

Abans de donar per tancat aquest exemple, hi ha un aspecte força significatiu que val la pena tenir en compte quan s'usen estructures de repetició. En aquest cas concret, què passa si quan us pregunten el valor fins on es vol sumar escriviu 0 o un nombre negatiu? Bé, per contestar aquesta pregunta cal recordar que la condició s'avalua tot just abans de cada iteració, inclosa la primera vegada que s'arriba a la sentència `while`. En aquest cas, tan sols començar, la condició lògica no es compleix, per la qual cosa mai no s'arriba a executar cap iteració.

Recordeu, doncs, que en usar una sentència `while` si la primera vegada que s'avalua la condició lògica aquesta avalua a `false` el codi inclòs dins del bucle no s'arribarà a executar mai.

3.2.5 Exemple: acumular càlculs

Els exemples que heu vist fins ara tenien en comú l'ús d'una variable de control de tipus comptador, a partir de la qual es pot veure de manera més o menys clara quantes iteracions farà el bucle. En tots els casos, la particularitat és que aquest comptador serveix com a valor auxiliar, que fins i tot pot ajudar a fer els càlculs que volem, però no és pròpiament el valor resultant. En l'exemple d'imprimir diversos caràcters '-' el resultat és el fet d'imprimir per pantalla. En el de la taula de multiplicar, el resultat que volem és el resultat de la multiplicació. Finalment, en l'exemple de sumar múltiples de tres, el resultat és la variable mateixa `resultat`.

Ara bé, en una estructura de repetició, l'evolució del mateix resultat que s'està calculant pot ser el senyal de sortida per deixar de fer iteracions. Aquest seria el cas d'usar variables amb el paper d'acumulador. Un exemple d'aquest comportament, en què una variable es va modificant de manera que es deixa d'iterar quan aquesta conté el resultat final, és el càlcul de l'operació mòdul amb enters (%).

Un acumulador reflecteix les modificacions en el resultat que apropen el final del bucle. Imatge de Paul Carvill

El mòdul calcula el residu de dividir un enter (el dividend) per un altre (el divisor). Una estratègia simple per calcular-lo és anar restant el divisor al dividend fins que ja no es pot fer més, ja que donaria negatiu. En aquest cas, el valor del dividend es va modificant directament fins a trobar la solució.

Si s'estructura pas per pas, seria:

1. Es pregunta quin és el dividend.
2. Es llegeix.
3. Es pregunta quin és el divisor.
4. Es llegeix.
5. Si el dividend és més petit que el divisor, ja hem acabat. El divisor ja és resultat del mòdul.
6. En cas contrari, al dividend se li resta el divisor.
7. Es torna a comprovar el pas 5.

Tot i que s'ha usat la paraula "si" en el pas 5, observant atentament les passes 5 a 7 es veu que en realitat denoten un bucle, ja que es van repetint fins que es produeix una condició concreta: que el valor del divisor és menor que el del dividend. Per tant, la condició per seguir repetint les instruccions és que el dividend sigui igual o més gran que el divisor.

El codi font del programa que du a terme aquestes passes seria el següent. Proveu que funciona al vostre entorn de treball. Per fer l'execució més aclaridora, a cada iteració es mostra per pantalla com es va modificant el dividend.

```
1  import java.util.Scanner;
2  //Un programa que llegeix un enter i el mostra per pantalla.
3  public class Modul {
4    public static void main (String[] args) {
5      Scanner lector = new Scanner(System.in);
6      //PAS 1 i 2
7      System.out.print("Quin és el dividend? ");
8      int dividend = lector.nextInt();
9      lector.nextLine();
10     //PAS 2 i 3
11     System.out.print("Quin és el divisor? ");
12     int divisor = lector.nextInt();
13     lector.nextLine();
14     //PAS 5
15     while (dividend >= divisor) {
16       //PAS 6
17       dividend = dividend - divisor;
18       System.out.println("Bucle: per ara el dividend val " + dividend + ".");
19       //PAS 7: Simplement equival a dir que fem la volta al bucle per avaluar
             la condició
```

```
20      }
21      System.out.println("El resultat final és " + dividend + ".");
22    }
23  }
```

La taula 3.4 mostra l'evolució del valor del dividend per a cada iteració per al càlcul de 17%4 (que ha de donar 1). El resultat final és el valor acumulat en la variable `dividend` quan se surt del bucle.

Taula 3.4. Evolució del bucle per calcular el mòdul usant un acumulador

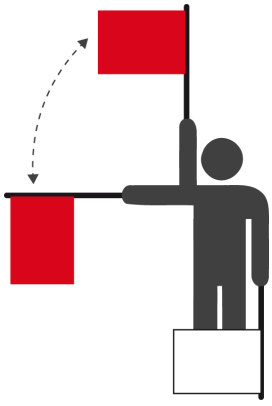
Iteració	Inici del bucle		Fi del bucle
	'dividend' val	Condició val	'dividend' val
1	17	(17 > 4), `true`	17 – 4 = 13
2	13	(13 > 4), `true`	13 – 4 = 9
3	9	(9 > 4), `true`	9 – 4 = 5
4	5	(5 > 4), `true`	5 – 4 = 1
5	1	(1 > 20), `false`	Ja hem sortit del bucle

Repte 3: l'ús de comptadors i acumuladors no és excloent, sinó que pot ser complementari. Penseu com es podria modificar el programa per calcular el resultat del mòdul i la divisió entera alhora. Recordeu que la divisió entera simplement seria comptar quantes vegades s'ha pogut restar el divisor.

3.2.6 Exemple: semàfors

El darrer cas que queda per estudiar és l'us d'un semàfor per indicar de manera explícita si ja no cal fer iteracions. Aquesta estratègia d'ús de variables de control es basa en situacions en què decidir si es vol continuar fent iteracions d'un bucle no es pot predir o calcular d'acord amb un valor que va augmentant o disminuint. Simplement, cal anar repetint fins que es compleixi una condició molt concreta. Llavors, del que es disposa és d'una variable de control, normalment de tipus booleà, sobre la qual es fa una assignació explícita que provocarà que la condició lògica avaluï directament a `false` i se surti del bucle.

Un exemple d'aquest cas és un programa en què cal endevinar un valor secret. Mitjançant una estructura de selecció és possible establir si s'ha encertat o no, però el que no té sentit és que cada cop que es vulgui provar d'endevinar-lo, s'hagi d'executar el programa de nou. El més normal és que es pregunti a l'usuari fins que l'encerti. En aquest cas, però, no hi ha un valor que a poc a poc, gradualment, va variant fins a poder establir que cal deixar d'iterar. Es passa de cop de continuar preguntant al fet que ja no calgui, segons la condició de si s'ha encertat o no. Aquesta es pot donar en qualsevol iteració, però és impossible estimar quan, ja que depèn totalment del valor introduït per l'usuari.



Un semàfor o banderola avisa: "Objectiu complert! No cal iterar més."

Si descrivim el què cal fer pas per pas, seria:

1. Decidir quin serà el nombre per endevinar.

2. Demanar que s'introdueixi un nombre pel teclat.
3. Llegir-lo.
4. Si el nombre no és el valor secret:
   (a) Cal avisar que s'ha fallat.
   (b) Tornar al pas 2.
5. En cas contrari, ja hem acabat. Mostrar una felicitació.

Les passes 2 a 4 denoten que hi ha una estructura de repetició, la condició de la qual és que cal iterar mentre el valor secret no s'encerti.

El codi corresponent a aquest procés és el que es mostra tot seguit. Proveu que, efectivament, continua preguntant nombres fins que s'encerta el valor secret. En aquest cas, la variable de control que fa de semàfor és `haEncertat`. Fixeu-vos que, evidentment, cal que el valor inicial sigui el que permetrà que, com a mínim, hi hagi una iteració del bucle (la condició avaluï a `true`). En cas contrari, mai no s'entraria al bucle.

```
1  import java.util.Scanner;
2  //Un programa en què cal endevinar un nombre.
3  public class Endevina {
4    //PAS 1
5    public static final int VALOR_SECRET = 4;
6    public static void main (String[] args) {
7      Scanner lector = new Scanner(System.in);
8      System.out.println("Comencem el joc.");
9      System.out.println("Endevina el valor enter, entre 0 i 10.");
10     boolean haEncertat = false;
11     while (!haEncertat) {
12       System.out.print("Quin valor creus que és? ");
13       int valorUsuari = lector.nextInt();
14       lector.nextLine();
15       if (VALOR_SECRET != valorUsuari) {
16         System.out.print("Has fallat! Torna a intentar-ho...");
17       } else {
18         haEncertat = true;
19       }
20     }
21     System.out.println("Enhorabona. Per fi l'has encertat!");
22   }
23 }
```

La taula 3.5 mostra l'evolució de les iteracions del bucle quan algú prova els nombres 3, 6, 9 i 4, en aquest ordre.

TAULA 3.5. Evolució del bucle per endevinar un valor secret

Iteració	Inici del bucle		Fi del bucle	
	'haEncertat' val	Condició val	'valorUsuari' val	'haEncertat' val
1	`false`	(!`false`), `true`	3	`false`
2	`false`	(!`false`), `true`	6	`false`
3	`false`	(!`false`), `true`	9	`false`
4	`false`	(!`false`), `true`	4	`true`
5	`true`	(!`true`), `false`	Ja hem sortit del bucle	

3.2.7 Exemple: semàfors i comptadors alhora

La categorització de les variables de control en tres models diferents no significa que una estructura de repetició sempre hagi de dependre d'una única variable. Recordeu que la condició lògica es representa com una expressió booleana i, per tant, pot ser tan complexa com es vulgui. En alguns casos, hi pot haver més d'una condició sota les quals es considera que no cal fer més iteracions d'un bucle.

Torneu a fer una ullada al programa per endevinar un valor secret. Ara ja té una mica més de sentit, però encara hi ha un aspecte que potser el fa una mica estrany: el programa no s'acabarà fins que endevineu el valor secret. Continuarà preguntant una vegada i una altra fins que encerteu. Si no encerteu mai, no s'acabarà mai. Potser seria més raonable posar un límit al nombre d'intents, de manera que si es falla més d'un cert nombre de vegades es considera que heu perdut el joc i s'acaba el programa. En aquest cas, la condició lògica ha de preveure **dues** possibilitats: si s'ha encertat el valor secret o si s'ha esgotat el nombre d'intents. És a dir, combinar un semàfor i un comptador.

Mai no hi haurà més iteracions que el valor estipulat en el comptador d'intents, d'acord amb el seu increment.

Descrit pas per pas, podria ser:

1. Decidir quin serà el nombre per endevinar i el màxim d'intents.
2. Demanar que s'introdueixi un nombre pel teclat.
3. Llegir-lo.
4. Descomptar un intent.
5. Si el nombre no és el valor secret, i si no s'han esgotat els intents:
   (a) Cal avisar que s'ha fallat.
   (b) Tornar al pas 2.
6. En cas contrari, ja hem acabat.
7. Si el darrer nombre dit és el valor secret, cal dir que s'ha guanyat.
8. Si el darrer nombre dit no és el valor secret, cal dir que s'ha perdut.

La característica més important d'aquest problema, en què hi ha més d'una possibilitat sota la qual el bucle deixa d'iterar, és, un cop se'n surt, detectar el motiu exacte pel qual se n'ha sortit. Abans, sortir del bucle sempre implicava que s'havia encertat el valor secret, però ara ja no. Aquest és el paper que tenen les passes 7 i 8. En aquest cas, una manera de veure si les iteracions han acabat perquè s'ha endevinat el valor secret o perquè s'han esgotat els intents és mirant el darrer valor que s'ha entrat.

El programa que duria a terme aquest joc seria el següent. Proveu de posar-lo en marxa.

```java
import java.util.Scanner;
//Un programa en què cal endevinar un nombre.
public class EndevinaSemafor {
  //PAS 1
  public static final int VALOR_SECRET = 4;
  public static final int MAX_INTENTS = 3;
  public static void main (String[] args) {
    Scanner lector = new Scanner(System.in);
    System.out.println("Comencem el joc.");
    System.out.println("Endevina el valor enter, entre 0 i 10, en tres intents
        .");
    boolean haEncertat = false;
    int intents = MAX_INTENTS;
    //Per garantir l'àmbit correcte de la variable, ara cal declarar-la.
    int valorUsuari = 0;
    while ((!haEncertat)&&(intents > 0)) {
      //PAS 2 i 3
      System.out.print("Quin valor creus que és? ");
      valorUsuari = lector.nextInt();
      lector.nextLine();
      //PAS 4
      intents = intents - 1;
      //PAS 5 i 6
      if (VALOR_SECRET != valorUsuari) {
        System.out.print("Has fallat! Torna a intentar-ho.");
      } else {
        haEncertat = true;
      }
    }
    if (valorUsuari == VALOR_SECRET) {
      //PAS 7: S'ha sortit del bucle perquè el valor era correcte.
      System.out.println("Enhorabona. Has encertat!");
    } else {
      //PAS 8: S'ha sortit del bucle perquè s'han esgotat els intents.
      System.out.println("Intents esgotats. Has perdut!");
    }
  }
}
```

Abans de seguir, val la pena subratllar que, en aquest programa, el valor llegit des del teclat s'usa fora del bucle, en la sentència `if/else`. Per tant, per garantir que l'àmbit de la variable `valorUsuari` és el correcte, s'ha de declarar amb anterioritat i dins del mateix bloc. Si es declara dins del codi del bucle, hi haurà un error.

La taula 3.6 mostra un exemple d'evolució del bucle per a un cas en què s'intenten els valors 2 i 4, de manera que s'encerta el valor abans d'esgotar els intents. A la darrera fila se subratlla la part de la condició que és determinant perquè avaluï a fals. Noteu que, en sortir del bucle, `valorUsuari` val 4, raó per la qual l'estructura de selecció considerarà que s'ha guanyat.

TAULA 3.6. Evolució del bucle per endevinar un valor secret, amb intents limitats, guanyant

	'haEncertat' val	'intents' val	Condició val	'valorUsuari' val	'haEncertat' val	'intents' val
1	`false`	3	(!`false`)&&(3>0), `true`	2	`false`	2
2	`false`	2	(!`false`)&&(2>0), `true`	4	`true`	1
3	`true`	1	(!`true`)&&(1>0), `false`		Ja hem sortit del bucle	

També és interessant veure què passa si es fallen tots els intents, per exemple introduint 2, 5 i 7. Això es veu a la taula 3.7. En aquest cas, en sortir del bucle, `valorUsuari` val 7, de manera que l'estructura de selecció considerarà que s'ha perdut.

Taula 3.7. Evolució del bucle per endevinar un valor secret, amb intents limitats, perdent

Iteració	Inici del bucle			Fi del bucle		
	'haEncertat' val	'intents' val	Condició val	'valorUsuari' val	'haEncertat' val	'intents' val
1	`false`	3	(!`false`)&&(3>0), `true`	2	`false`	2
2	`false`	2	(!`false`)&&(2>0), `true`	5	`false`	1
3	`false`	1	(!`false`)&&(1>0), `true`	7	`false`	0
4	`false`	0	(!`false`)&&(0>0), `false`		Ja hem sortit del bucle	

Finalment, per la complexitat de l'exemple, també val la pena estudiar amb detall un cas especial: quan s'encerta en el darrer intent, per exemple, en introduir successivament els valors 2, 5 i 4. Aquest cas és especial, ja que la condició lògica avalua a `false` com a resultat del fet que les dues expressions individuals també avaluen a `false`, tant la que comprova si s'han esgotat el nombre d'intents com la que comprova si ha encertat. Ara bé, com que el darrer valor que s'ha intentat és el correcte, l'estructura de selecció considera que es guanya el joc. Això es pot veure més detalladament a la taula 3.8.

Taula 3.8. Evolució del bucle per endevinar un valor secret, amb intents limitats, guanyant //in extremis//

Iteració	Inici del bucle			Fi del bucle		
	'haEncertat' val	'intents' val	Condició val	'valorUsuari' val	'haEncertat' val	'intents' val
1	`false`	3	(!`false`)&&(3>0), `true`	2	`false`	2
2	`false`	2	(!`false`)&&(2>0), `true`	5	`false`	1
3	`false`	1	(!`false`)&&(1>0), `true`	4	`true`	0
4	`true`	0	(!`true`)&&(0>0), `false`		Ja hem sortit del bucle	

3.3 Repetir almenys un cop: la sentència "do/while"

Tot i que la sentència `while` és més que suficient per dur a terme pràcticament qualsevol estructura de repetició, n'hi ha d'altres que s'adapten millor a certs casos molt concrets. La seva aportació consisteix exclusivament a facilitar la lectura del codi o permetre dur a terme algunes tasques de manera automàtica.

La sentència **do/while** permet repetir l'execució del bucle mentre es verifiqui la condició lògica. A diferència de la sentència `while`, la condició es verifica al final de cada iteració. Per tant, independentment de com avaluï la condició, com a mínim sempre es durà a terme la primera iteració.

Al contrari que la sentència `while`, la sentència `do/while` no és tan comuna en els llenguatges de programació.

3.3.1 Sintaxi i estructura

Per dur a terme aquest tipus de control sobre les iteracions d'un bucle, la sintaxi d'aquesta sentència en llenguatge Java és la següent:

```
1 do {
2   Instruccions per executar dins del bucle
3 } while (expressió booleana);
```

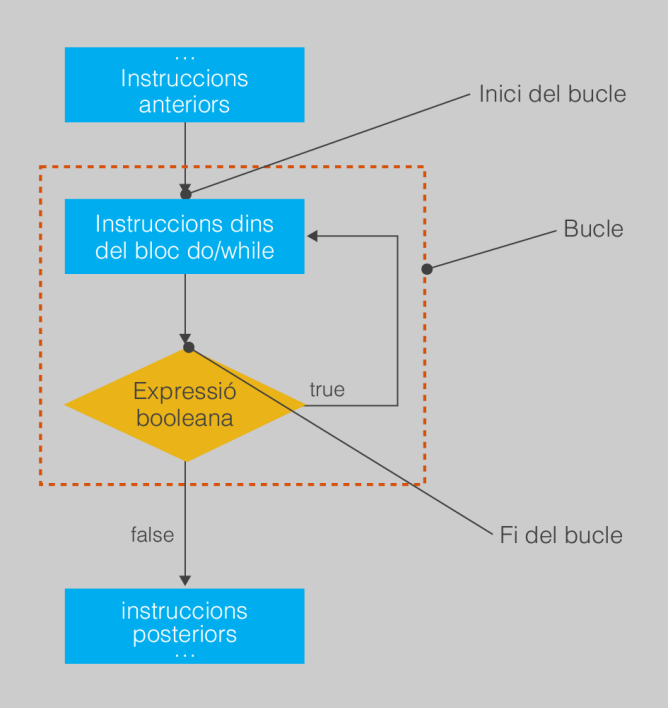
Com podeu veure, és molt semblant a la sentència `while`, però invertint el lloc on apareix la condició lògica. Per poder distingir clarament on comença el bucle s'usa la paraula clau `do`. Les instruccions del bucle continuen encerclades entre claus, `{...}`.

do/while

El nom de la sentència `do/while` vol dir: "Fes això, i continua fent-ho mentre es compleixi aquesta condició".

La figura 3.3 mostra un diagrama del flux de control d'aquesta sentència i estableix també l'ordre sota el qual s'avalua l'expressió que representa la condició lògica, en aquest cas al final, amb vista a decidir si cal executar o no una nova iteració del bucle.

Figura 3.3. Diagrama de flux de control d'una sentència "do-while"



3.3.2 Exemple: control d'entrada per teclat

En un exemple anterior ja s'ha vist la utilitat de les estructures de repetició a l'hora de preguntar dades a l'usuari. Ara bé, un àmbit en què encara és més útil i més freqüent usar-la és el de controlar si la dada es correspon amb algun dels valors esperats abans de donar-la per bona i usar-la. Amb el que heu vist fins al moment, si l'usuari s'equivoca i escriu alguna dada errònia (per exemple, fora del rang esperat), sou capaços de detectar-ho i mitjançant una estructura de selecció avisar l'usuari, però llavors el programa acaba. Això no té gaire sentit; seria molt més còmode simplement tornar-li a preguntar repetidament fins que introdueixi un valor que s'adeqüi als valors esperats.

Tot i que amb un ús assenyat de la sentència `while` és pot dur a terme aquesta tasca, reemplaçar-la per la sentència `do/while` pot ser especialment còmode. El motiu principal és que en aquests casos, abans d'avaluar si la dada és correcta o no, primer cal haver-la llegit. Per tant, és més intuïtiu avaluar la condició lògica al final i no al principi del bucle, un cop realment ja es disposa de la dada.

Normalment, en usar aquesta sentència, la condició lògica avalua directament el valor que es vol controlar, per veure si compleix els requisits establerts. La dada llegida mateixa fa les funcions de semàfor.

L'esquema general per fer això seria:

1. Demanar la dada a l'usuari.
2. Llegir-la.
3. Mirar si la dada és correcta. Si no ho és, tornar al punt 1.
4. Si és correcta, ja hi podem operar.

Les passes 1 a 3 identifiquen que hi ha un bucle que té com a condició lògica que les dades escrites no són correctes. Ara bé, l'avaluació d'aquest fet es fa al final, no al principi de la seqüència d'instruccions. A més a més, es compleix que, fins i tot en el millor cas, si l'usuari no s'equivoca, les passes 1 a 3 sempre es duran a terme almenys un cop.

Per exemple, proveu el programa següent, que garanteix que qualsevol valor introduït estarà entre 0 i 10 (com és el cas del que s'espera en l'exemple anterior d'endevinar un nombre).

```
1  import java.util.Scanner;
2  //Anem a llegir un enter entre 0 i 10.
3  public class ValorFitat {
4    public static void main (String[] args) {
5      Scanner lector = new Scanner(System.in);
6      int valorUsuari = 0;
7      do {
8        //PAS 1 i 2: Com a mínim, segur que es pregunta un cop.
9        System.out.print("Introdueix un valor enter entre 0 i 10: ");
10       valorUsuari = lector.nextInt();
```

```
11        lector.nextLine();
12        //PAS 3: Només té sentit avaluar si 'valor' és vàlid un cop s'ha llegit.
13      } while ((valorUsuari < 0)||(valorUsuari > 10));
14      //PAS 4: Tot correcte
15      System.out.println("Dada correcta. Has escrit " + valorUsuari);
16    }
17  }
```

La taula 3.9 mostra un exemple d'evolució del bucle amb l'entrada dels nombres –4, 15 i 4. En aquest cas, la columna d'avaluació de la condició es troba al final del bucle. A l'avaluació, se subratlla la part de l'expressió que és determinant perquè avaluï a `true` i es provoqui una nova iteració.

TAULA 3.9. Evolució del bucle per garantir un valor entre 0 i 10 amb una sentència "do/while"

Iteració	Inici del bucle	Fi del bucle	
	'valorUsuari' val	'valorUsuari' val	Condició val
1	0	-4	(–4<0)||(–4>10), `true`
2	-4	15	(15<0)||(15>10), `true`
3	15	4	(4<0)||(4>10), `false`
4	4	Ja hem sortit del bucle	

En realitat, en aquest cas, un cop la condició ha avaluat a `false` a la tercera iteració ja es continua directament amb la instrucció següent del programa. No hi ha cap nou salt a l'inici del bucle, al contrari del que passa amb la sentència `while`. L'avaluació de la condició es fa al final de cada iteració, i no abans de començar cada una.

Repte 4: apliqueu aquesta comprovació a algun dels exemples anteriors vistos fins al moment. Per exemple, altres casos en què té sentit comprovar si un valor pertany a un rang esperat per garantir que un preu és positiu, o si un nombre de mes està entre 1 i 12.

3.4 Repetir un cert nombre de vegades: la sentència "for"

Oblidar-se de modificar una variable comptador és un error molt típic quan s'usen estructures de repetició.

En algunes situacions especials ja es coneix, *a priori*, la quantitat exacta de vegades que caldrà repetir el bucle. En tal cas és útil disposar d'un mecanisme que representi de manera més clara la declaració d'una variable de control de tipus comptador, l'especificació de fins on s'ha de comptar, i que al final de cada iteració incrementi o disminueixi el seu valor de manera automàtica, en lloc d'haver de fer-ho nosaltres. Automatitzar aquest darrer punt és molt important, ja que evita que per un oblit no es faci i s'acabi generant un bucle infinit.

La sentència **for** permet repetir un nombre determinat de vegades un conjunt d'instruccions.

La majoria de llenguatges solen disposar de l'equivalent a aquesta sentència.

3.4.1 Sintaxi i estructura

La sintaxi d'aquesta sentència en llenguatge Java és una mica més complexa, ja que hi intervenen molts factors. Cal especificar tres apartats especials separats per punt i coma (`;`):

```
for (inicialització comptador ; expressió booleana ; increment comptador) {
  Instruccions per executar dins del bucle
}
```

for

El nom de la sentència `for` vol dir: "Per veure quantes vegades cal fer això, compta d'aquest valor a aquest altre".

La descripció de cada apartat és la següent:

- Inicialització comptador: es tracta de la inicialització d'una variable de tipus numèric que servirà com a comptador. És exactament igual que assignar un valor a una variable qualsevol (`identificador = valorInicial`). Si es vol, es permet declarar la variable alhora que s'inicialitza (`tipus identificador = valorInicial`).

- Expressió booleana: es tracta de la condició lògica que indica si cal fer una nova iteració o no, igual que en la resta d'estructures de repetició.

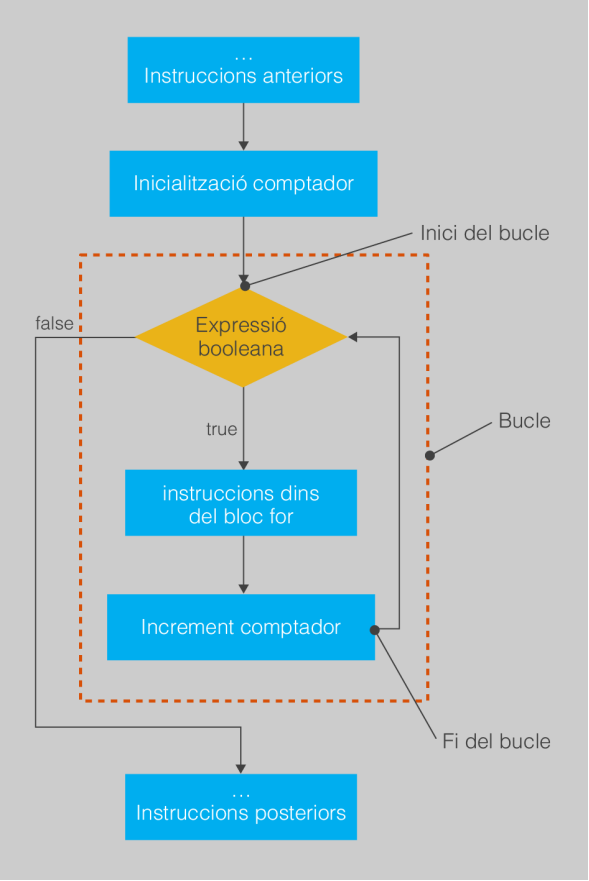
- Increment: es tracta d'una instrucció que modifica el valor del comptador, normalment una assignació. Aquesta instrucció s'executa automàticament al final de cada iteració. Tot i el seu nom, tant pot ser un increment com una disminució del valor.

La figura 3.4 mostra un diagrama de flux de control d'aquesta sentència i indica quan s'executa cada element de la declaració de la sentència i quan s'avalua l'expressió que denota la condició lògica. Noteu que la inicialització només es fa una vegada, tot just abans de fer la primera avaluació de la condició lògica. En canvi, l'increment del comptador es fa al final de cada iteració, com si fos la darrera instrucció escrita dins del bucle.

De fet, la sentència `for` és equivalent a una sentència `while` que segueixi l'estructura següent:

```
inicialització comptador
while (expressió booleana) {
  Instruccions per executar dins del bucle
  Increment comptador
}
```

FIGURA 3.4. Diagrama de flux de control d'una sentència "for"



3.4.2 Exemple: la taula de multiplicar, versió 2

La sentència `for` s'usa normalment en lloc de `while` en casos en què cal una variable de control que va variant el seu valor en una quantitat fixa a cada iteració, ja sigui incrementant-se o decreixent. És el que passa, per exemple, amb els casos controlats per un comptador, com els primers exemples de la sentència `while`.

L'operador postincrement/decrement

Per posar un exemple de la sentència `for`, s'introduiran uns nous operadors unaris existents en alguns llenguatges, com el Java. Aquests són especialment útils per especificar l'autoincrement en el tercer apartat de la sentència `for`. Es tracta del postincrement (`++`) i del postdecrement (`-`). El resultat d'aplicar-lo és que se suma o es resta 1 al valor de la variable, respectivament. La sintaxi és: `identificadorVariable++` i `identificadorVariable-`, en cada cas. Per exemple, `i++`, `valor-`, etc.

La seva particularitat és que, com a operador, es pot usar directament dins d'una expressió. En aquest cas, a l'hora d'avaluar-la la seva precedència és la primera.

Tot seguit hi ha l'adaptació a la sentència `for` de l'exemple de la taula de multiplicar. Comproveu que fa exactament el mateix.

```java
import java.util.Scanner;
//Un programa que mostra la taula de multiplicar d'un nombre, usant 'for'.
public class TaulaMultiplicarFor {
  public static void main (String[] args) {
    //S'inicialitza la biblioteca.
    Scanner lector = new Scanner(System.in);
    //Pregunta el nombre.
    System.out.print("Quina taula de multiplicar vols? ");
    int taula = lector.nextInt();
    lector.nextLine();
    //El comptador servirà per fer càlculs.
    //En lloc de 'i++' també es podria escriure 'i = i + 1'.
    for (int i = 0; i <= 10; i++) {
      int resultat = taula*i;
      System.out.println(taula + " * " + i + " = " + resultat);
    }
    System.out.println("Aquesta ha estat la taula del " + taula);
  }
}
```

De fet, la taula d'evolució del bucle és exactament igual que la de la versió original amb la sentència `while` (taula 3.2).

Repte 5: com passa amb els programes basats en la sentència `while` amb una variable de control de tipus comptador, el comptador d'una sentència `for` no necessàriament s'ha de modificar sumant/restant d'u en u. Adapteu l'exemple de la suma de múltiples de tres usant la sentència `for`.

3.4.3 Exemple: més enllà de sumar i restar

De la mateixa manera que res no obliga que un comptador s'incrementi d'u en u, tampoc no és imprescindible que l'operació sigui una suma o una resta. Qualsevol operació que garanteixi que el valor es va apropant a un cas en què la condició lògica avaluarà a `false` i se sortirà del bucle és suficient. Això també és cert per a l'apartat d'autoincrement de la sentència `for`.

Per exemple, el codi següent usa una sentència `for` per mostrar totes les potències de dos menors o iguals que una fita que estableix l'usuari. Per fer-ho, incrementa la variable de control (en aquest cas, amb el paper d'acumulador) fent multiplicacions. Compileu-lo i executeu-lo per veure que, efectivament, és així.

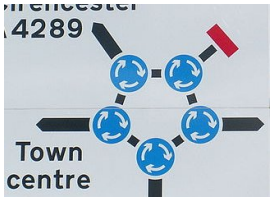
```java
import java.util.Scanner;
//Anem a calcular potències de dos.
public class Potencies {
  public static void main (String[] args) {
    Scanner lector = new Scanner(System.in);
    System.out.print("Fins a quin valor vols anar cercant potències de dos? ");
    int valor = lector.nextInt();
    lector.nextLine();
    //La variable "i" s'incrementa multiplicant en lloc de sumant.
    //"i" és el mateix resultat.
    for (int i = 1; i <= valor; i = i*2) {
      System.out.println(i);
    }
  }
}
```

La taula 3.10 mostra l'evolució del bucle per al cas de posar com a límit el valor 70. Recordeu que el valor inicial de la variable `i` en la primera iteració del bucle es deu a la inicialització dins del primer apartat de la sentència `for`, mentre que el valor al final es deu a l'autoincrement especificat en el tercer apartat (`i = 2*i`).

TAULA 3.10. Evolució del bucle per calcular potències de 2 usant una sentència "for"

Iteració	Inici del bucle		Fi del bucle
	'i' val	Condició val	'i' val
1	1	(1 $\leq$ 70), `true`	2 * 1 = 2
2	2	(2 $\leq$ 70), `true`	2 * 2 = 4
3	4	(4 $\leq$ 70), `true`	2 * 4 = 8
4	8	(8 $\leq$ 70), `true`	2 * 8 = 16
5	16	(16 $\leq$ 70), `true`	2 * 16 = 32
6	32	(32 $\leq$ 70), `true`	2 * 32 = 64
7	64	(64 $\leq$ 70), `true`	2 * 64 = 128
8	128	(128 $\leq$ 100), `false`	Ja hem sortit del bucle

3.5 Combinació d'estructures de selecció



Fer bucles dins d'altres bucles és com una "rotonda màgica" Imatge de Wikimedia Commons

Com passava amb les estructures de selecció, res no impedeix combinar diferents estructures de repetició, imbricades unes dins de les altres, per dur a terme tasques més complexes. En darrera instància, la combinació i imbricació d'estructures de selecció i repetició conformarà el vostre programa.

Quan es combinen estructures de repetició cal ser molt acurats i tenir present quines variables de control estan associades a la condició lògica de cada bucle. Tampoc no us oblideu de sagnar correctament cada bloc d'instruccions, per poder així identificar ràpidament on acaba i comença cada estructura.

3.5.1 Exemple: la taula de multiplicar, versió 3

Normalment, la combinació d'estructures de repetició sempre està fonamentada per la necessitat de fer diversos cops una tasca que ja de per si requereix una estructura de repetició. Per exemple, suposeu que en lloc de voler un programa que mostri una única taula de multiplicar, com en un exemple anterior, es volen mostrar diverses taules de multiplicar. Des de la de l'1 fins al valor que escolliu. Hi ha dues tasques repetitives: cal repetir N vegades una tasca que mostra una taula completa, la qual és a la vegada la repetició de 10 multiplicacions.

El codi que duria a terme aquesta tasca seria el següent. Com que es tracta d'una tasca basada en un comptador, s'usarà la sentència `for` per a les dues estructures repetitives.

```java
import java.util.Scanner;
//Un programa que mostra N taules de multiplicar.
public class TaulaMultiplicarN {
  public static void main (String[] args) {
    Scanner lector = new Scanner(System.in);
    System.out.print("Fins a quina taula de multiplicar vols conèixer? ");
    int taules = lector.nextInt();
    lector.nextLine();
    //Bucle de primer nivell: inici del bucle que genera N taules de
        multiplicar.
    for(int i = 1; i <= taules; i++) {
      System.out.println("La taula del " + i);
      //Bucle de segon nivell: inici del bucle que genera una taula concreta.
      for(int j = 1; j <= 10; j++) {
        int resultat = i*j;
        System.out.println(i + " * " + j + " = " + resultat);
      }
      //Fi del bucle que genera una taula concreta.
      System.out.println("----------");
    }
    //Fi del bucle que genera N taules de multiplicar.
  }
}
```

Aquest exemple també és força interessant, ja que té en compte l'àmbit de les variables que s'usen com a comptador per dur a terme la seva tasca. Així, doncs, la variable `i`, el comptador del bucle de primer nivell (el més "extern"), pot ser usada dins del bucle de segon nivell (el més "intern"), ja que es considera declarada fins a la clau que tanca el seu bloc de codi. El cas invers per a `j` no és cert i fer-ho produiria un error de sintaxi.

3.5.2 Exemple: endevinar el nombre secret, versió 3

Val la pena mostrar com a exemple un programa una mica més complex, també basat en la combinació d'estructures de repetició. En un exemple anterior s'ha vist que aquest tipus d'estructures són útils per controlar si l'entrada de dades és correcta. Si es combina aquest exemple amb el programa per endevinar un nombre secret, en què es pregunta repetides vegades a l'usuari quina resposta vol donar, llavors tindreu l'esquema del programa següent, que depèn de dues estructures de repetició, una dins de l'altra:

1. Decidir quin serà el nombre per endevinar i el màxim d'intents.
2. Demanar que s'introdueixi un nombre pel teclat.
3. Llegir-lo.
4. Mirar si el valor és correcte. Si no ho és, tornar al pas 2.
5. Si és correcte, ja s'hi pot operar.
6. Descomptar un intent.
7. Si el nombre no és el valor secret i no s'han esgotat els intents:

(a) Cal avisar que s'ha fallat.

(b) Tornar al pas 2.

8. En cas contrari, ja hem acabat.

9. Si el darrer nombre dit és el valor secret, cal dir que s'ha guanyat.

10. Si el darrer nombre dit no és el valor secret, cal dir que s'ha perdut.

El codi seria el que es mostra tot seguit. Tot i l'aparent complexitat, fixeu-vos que, bàsicament, es parteix del programa original d'endevinar el valor secret i que, a la part on es demanava el valor, s'insereix directament l'estructura de l'exemple de control de dades correctes. Compileu-lo i executeu-lo per comprovar que funciona.

```
1  import java.util.Scanner;
2  //Un programa en què cal endevinar un nombre.
3  public class EndevinaN {
4    //PAS 1
5    public static final int VALOR_SECRET = 4;
6    public static final int MAX_INTENTS = 3;
7    public static void main (String[] args) {
8      Scanner lector = new Scanner(System.in);
9      System.out.println("Comencem el joc.");
10     System.out.println("Endevina el valor enter entre 0 i 10 en tres intents.")
           ;
11     boolean haEncertat = false;
12     int intents = MAX_INTENTS;
13     int valorUsuari = 0;
14     //Inici bucle per endevinar el valor.
15     while ((!haEncertat)&&(intents > 0)) {
16       //Inici bucle per obtenir una dada des del teclat.
17       do {
18         //PAS 2 i 3: Com a mínim, segur que es pregunta un cop.
19         System.out.print("Introdueix un valor enter entre 0 i 10: ");
20         valorUsuari = lector.nextInt();
21         lector.nextLine();
22         //PAS 4
23       } while ((valorUsuari < 0)||(valorUsuari > 10));
24       //Fi bucle per obtenir una dada des del teclat.
25       //PAS 6
26       intents = intents - 1;
27       //PAS 7 i 8
28       if (VALOR_SECRET != valorUsuari) {
29         System.out.print("Has fallat! Torna a intentar-ho.");
30       } else {
31         haEncertat = true;
32       }
33     }
34     //Fi bucle per endevinar el valor.
35     if (valorUsuari == VALOR_SECRET) {
36       //PAS 9: S'ha sortit del bucle perquè el valor era correcte.
37       System.out.println("Enhorabona. Ho has encertat!");
38     } else {
39       //PAS 10: S'ha sortit del bucle perquè s'han esgotat els intents.
40       System.out.println("Intents esgotats. Has perdut!");
41     }
42   }
43 }
```

3.6 Solucions dels reptes proposats

Repte 1:

```
import java.util.Scanner;
public class LiniaPregunta {
  public static void main (String[] args) {
    //Part  modificada. Es pregunta el valor del comptador.
    Scanner lector = new Scanner(System.in);
    System.out.print("Quantes iteracions vols fer? ");
    boolean tipusCorrecte = lector.hasNextInt();
    //S'ha escrit un enter correctament. Ja es pot llegir.
    if (tipusCorrecte) {
      //Llegim el nombre d'iteracions.
      int iteracions = lector.nextInt();
      //Inicialitzem un comptador.
      int i = 0;
      //Ja hem fet això els cops que toca?
      while (i < iteracions) {
        System.out.print('-');
        //Ho hem fet un cop, sumem 1 al comptador.
        i = i + 1;
      }
      //Forcem un salt de línia.
      System.out.println();
    } else {
      //No s'ha escrit un enter.
      System.out.println("El valor introduït no és un enter.");
    }
    lector.nextLine();
  }
}
```

Repte 2:

```
import java.util.Scanner;
public class TaulaMultiplicarEnrere {
  public static void main (String[] args) {
    //S'inicialitza la biblioteca.
    Scanner lector = new Scanner(System.in);
    //Pregunta el nombre.
    System.out.print("Quina taula de multiplicar vols veure? ");
    int taula = lector.nextInt();
    lector.nextLine();
    //El comptador servirà per fer càlculs.
    int i = 10;
    while (i >= 1) {
      int resultat = taula*i;
      System.out.println(taula + " * " + i + " = " + resultat);
      i = i - 1;
    }
    System.out.println("Aquesta ha estat la taula del " + taula);
  }
}
```

Repte 3:

```java
import java.util.Scanner;
public class ModulIDivisio {
  public static void main (String[] args) {
    Scanner lector = new Scanner(System.in);
    //PAS 1 i 2
    System.out.print("Quin serà el dividend? ");
    int dividend = lector.nextInt();
    lector.nextLine();
    //PAS 2 i 3
    System.out.print("Quin serà el divisor? ");
    int divisor = lector.nextInt();
    lector.nextLine();
    //PAS 5
    int quocient = 0;
    //PAS 6
    while (dividend >= divisor) {
      //PAS 7
      dividend = dividend - divisor;
      //PAS 8
      quocient++;
      //PAS 9: Simplement equival a dir que fem la volta al bucle per avaluar
          la condició
      System.out.println("Bucle: per ara el dividend val " + dividend + ".");
    }
    //PAS 10: La variable "quocient" té el quocient en acabar el bucle!
    System.out.println("La divisió és " + quocient + ".");
    //PAS 11: La variable "dividend" té el mòdul en acabar el bucle!
    System.out.println("El mòdul és " + dividend + ".");
  }
}
```

Repte 4:

```java
import java.util.Scanner;
public class ValorMesFitat {
  public static void main (String[] args) {
    Scanner lector = new Scanner(System.in);
    int mes = 0;
    do {
      System.out.print("Introdueix un número de mes [1-12]: ");
      mes = lector.nextInt();
      lector.nextLine();
    } while ((mes < 1)||(mes > 12));
    System.out.println("Dada correcta. Has escrit " + mes);
    System.out.print("Els dies d'aquest mes són...");
    if (mes == 2) {
      System.out.println("28 o 29!");
    } else if ((mes == 4)||(mes == 6)||(mes == 9)||(mes == 11)) {
      System.out.println("30!");
    } else {
      System.out.println("31!");
    }
  }
}
```

Repte 5:

```java
import java.util.Scanner;
public class SumarMultiplesTresFor {
  public static void main (String[] args) {
    Scanner lector = new Scanner(System.in);
    System.out.print("Fins a quin valor vols sumar múltiples de 3? ");
    int limit = lector.nextInt();
    lector.nextLine();
    int resultat = 0;
    for(int i = 0; i <= limit; i = i + 3 ) {
      System.out.println("Afegim " + i +"...");
      resultat = resultat + i;
    }
    System.out.println("El resultat final és " + resultat + ".");
  }
}
```