Tipus de dades compostes

Joan Arnedo Moreno

Programació bàsica (ASX)
Programació (DAM)

Índex

Introducció

Qualsevol dada manipulada dins d'un programa pertany a algun dels tipus primitius, ja que són els únics que sap processar el maquinari de l'ordinador. Normalment, aquestes dades són emmagatzemades dins de variables, un dels elements bàsics d'un programa amb vista a manipular dades. Aquesta aproximació és útil sempre que el nombre de dades que es vol tractar sigui relativament limitat. Ara bé, precisament, un dels aspectes en què es fa especialment recomanable la resolució d'un problema usant un programa d'ordinador és quan el volum de dades per processar és molt gran. Suposeu que en lloc de treballar amb uns quants valors independents, que us cal desar i processar, n'heu de treballar amb molts: 50, 100, 2.000 o molts més. En aquest cas, l'aproximació de desar cada valor en una variable dins el programa faria la tasca del programador molt farragosa. El codi font tindria més instruccions de declaració de variables que no pas potser per resoldre el problema!

Per solucionar aquesta qüestió, els llenguatges de programació ofereixen uns tipus de dades especials, anomenats **tipus compostos**, que permeten aglutinar en una única variable una quantitat arbitrària de dades individuals pertanyents a algun tipus concret. Aquestes queden empaquetades d'una manera fàcil de gestionar a l'hora de consultar i modificar cada valor individual.

A l'apartat "Tipus de dades compostes: *arrays*" es presenta el tipus compost de dades més bàsic, l'**array**, disponible en la majoria de llenguatges d'alt nivell. Aquest permet emmagatzemar en una única variable un conjunt de valors en forma de seqüència, amb l'única restricció que tots pertanyin al mateix tipus de dades. D'aquesta manera, en una única variable podem disposar de qualsevol nombre d'enters, reals, etc. Dins l'apartat en veureu la sintaxi en Java i els mecanismes més habituals de tractament de dades emmagatzemades dins seu.

A part dels *arrays*, hi ha altres tipus compost de dades de diferents graus de complexitat i amb l'objectiu de solucionar problemàtiques semblants, però amb certs matisos. Un exemple és la manipulació de text. El tipus primitiu caràcter, per si mateix, no té excessiva utilitat, ja que normalment les persones usem paraules o, si més no, text de longitud arbitrària per comunicar-nos. Per tant, gestionar aquest tipus d'informació caràcter per caràcter també és poc viable. Seria interessant disposar d'algun mecanisme per gestionar de manera senzilla una combinació qualsevol de diversos caràcters. Aquesta és precisament la tasca destinada al tipus de dades anomenat **cadena de text**.

A l'apartat següent, "Tractament de cadenes de text", es presenta amb més detall aquest tipus compost de dades molt útil, anomenat `String` en el llenguatge Java. Si bé ja s'havia presentat de manera superficial en apartats anteriors, veureu com usar-lo per fer tasques més complexes que mostrar text per pantalla, com processar els caràcters que conté o convertir el text a altres tipus primitius.

Resultats d'aprenentatge

En finalitzar aquesta unitat, l'alumne/a:

1. Escriu i prova programes senzills reconeixent i aplicant els fonaments de la programació modular.

1. Tipus de dades compostes: "arrays"

Suposeu que voleu fer un programa per analitzar les notes finals d'un conjunt d'estudiants. Les operacions per fer poden ser diverses: calcular la nota mitjana, veure quants estudiants han aprovat, suspès o han superat cert llindar de nota, fer gràfiques de rendiment, etc. A més, no es descarta que en el futur el programa es modifiqui per afegir-hi noves funcionalitats. El *quid* de la qüestió és que, per poder preveure totes aquestes possibilitats i d'altres de futures encara desconegudes, us cal disposar dins del programa del valor de cada nota individual. En un cas com aquest, tal com s'ha plantejat fins ara l'aproximació per manipular dades dins d'un programa, això implica que cada nota s'ha d'emmagatzemar dins d'una variable diferent. Si hi ha 50 estudiants, això vol dir que caldria declarar 50 variables.

Malauradament, aquesta aproximació no funciona en un cas com aquest, en què el nombre de dades per processar és relativament alt. D'una banda, imagineu-vos l'aspecte que tindrà un codi font en què cal declarar i manipular 50 variables per fer tota mena de càlculs. Les expressions per fer càlculs entre elles, com simplement fer la mitjana (sumar-les totes i dividir pel nombre de valors), serien enormes. D'altra banda, què passa si cal processar les notes de dos-cents o mil estudiants? És factible declarar mil variables? Pitjor encara: què passa si el nombre d'estudiants de cada curs és totalment diferent? En cada curs el nombre de variables usades no encaixarà amb el d'estudiants i serà necessari modificar el codi font tenint en compte el nombre exacte de valors per tractar i tornar a compilar.

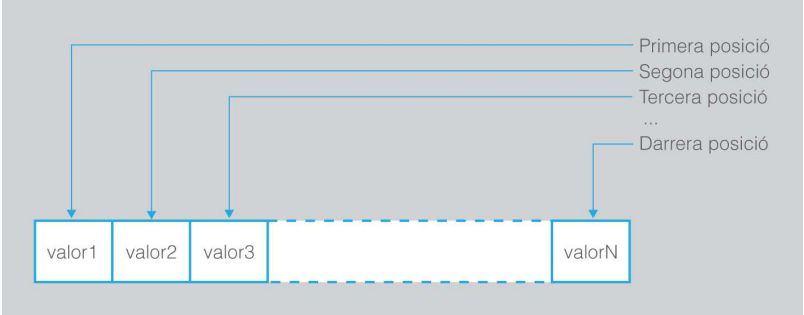
Ja a simple vista es pot apreciar que tot plegat és inviable. Cal un sistema flexible per emmagatzemar un nombre arbitrari de valors de manera que aquests siguin fàcils de manipular. Encara més, ha de ser possible que el nombre de valors emmagatzemats pugui ser diferent per a cada execució del programa. Per resoldre aquesta problemàtica, els llenguatges de programació d'alt nivell ofereixen el tipus de dada compost *array*, o taula.

Un **tipus de dada compost** és aquell que permet emmagatzemar més d'un valor dins d'una única variable. En el cas de l'**array**, aquest permet emmagatzemar, en forma de seqüència, una quantitat predeterminada de valors pertanyents al **mateix tipus** de dades.

Un "array" és com una calaixera.
Imatge de Liverpool Design Festival

Per tal de diferenciar els diferents valors emmagatzemats, l'*array* gestiona el seu contingut d'acord amb posicions que segueixen un ordre numèric: el valor emmagatzemat a la primera posició, a la segona, a la tercera, etc. A efectes pràctics, es pot considerar que cada posició individual es comporta exactament igual que una variable del tipus de dada triat. Tant es pot consultar el valor com emmagatzemar-hi dades. Aquest comportament s'esquematitza a la figura 1.1.

Figura 1.1. Organització de les dades emmagatzemades dins d'un "array"



Abans de veure amb més detall com funcionen, heu de tenir present que la sintaxi per usar *arrays* en els diferents llenguatges de programació pot ser relativament diferent. Aquest apartat se centrarà en la sintaxi concreta del Java, que té certes particularitats úniques. Tot i així, el concepte general d'accés a valors ordenats com una seqüència en forma d'*array* és aplicable a qualsevol llenguatge.

1.1 Declaració i inicialització d'"arrays"

Com passa amb dades de qualsevol altre tipus, abans de poder fer ús d'una variable de tipus *array*, cal declarar-la i atorgar-li un valor inicial correctament. La sintaxi per declarar un *array* és força semblant a la d'altres tipus, però la inicialització té unes particularitats especials a causa de les seves característiques pròpies:

- D'una banda, cal decidir de quantes posicions disposarà l'*array*: la seva **mida**. Mai no hi podrà haver emmagatzemats més valors que la seva capacitat.

- D'una altra banda, cal decidir el valor inicial per a cada posició individual de l'*array*. En aquest aspecte, cada posició té el mateix paper que una variable individual dins del programa, i per tant, ha d'estar inicialitzada amb algun valor.

> En el moment de la inicialització d'un *array*, la seva mida ha de ser coneguda. Un cop ha estat inicialitzat amb un nombre de posicions concretes, aquesta ja no pot canviar.

Abans de poder usar un *array* és obligatori inicialitzar-lo. Java donarà un error de compilació en cas contrari.

1.1.1 Inicialització a un valor concret

Com passa amb les variables individuals, hi ha la possibilitat que en el moment de generar el codi font ja tingueu uns valors concrets al cap amb els quals voleu inicialitzar cadascuna de les posicions de l'*array*. En aquest cas, la sintaxi de la inicialització és la següent:

```
1 paraulaClauTipus[] identificadorVariable = {valor1, valor2, ... , valorN};
```

La part esquerra és pràcticament igual que per a la declaració d'una variable qualsevol. El que indica que en realitat s'està declarant un *array* són els dos claudàtors `[]` immediatament després de la declaració del tipus. Cadascuna de les posicions de l'*array* només podrà emmagatzemar valors del tipus declarat aquí.

No hi pot haver dades de tipus barrejats dins d'un *array*.

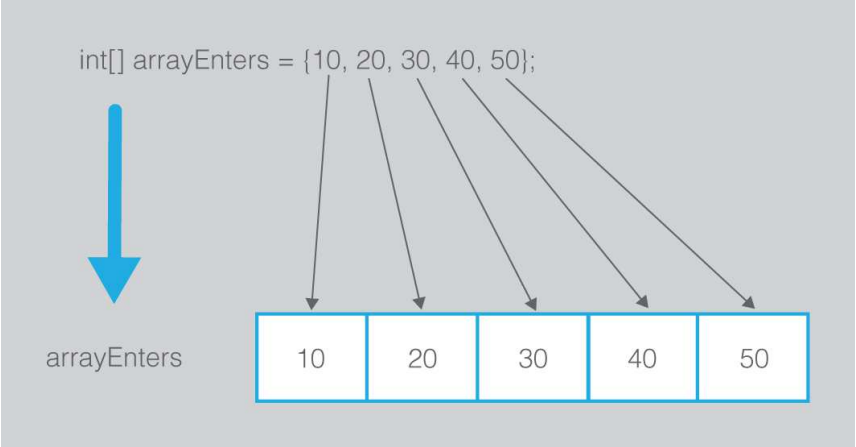
Per especificar els valors emmagatzemats inicialment a cada posició n'hi ha prou que aquests s'enumerin entre claus `{...}`, separats per comes `(,)`. Aquests valors han de ser del mateix tipus de dada que el declarat a la part esquerra i s'emmagatzemaran en el mateix ordre en què s'hagin enumerat: el primer valor a la primera posició de l'*array*, el segon a la segona posició, etc. En aquest cas, la mida de l'*array* serà automàticament igual que el nombre de valors enumerats.

Per exemple, per declarar un *array* de 5 posicions en què es poden emmagatzemar valors de tipus enter, cadascuna amb els valors inicials 10, 20, 30, 40 i 50, respectivament, es faria de la manera següent. La mida d'aquest *array* seria 5.

```
1 int[] arrayEnters = {10, 20, 30, 40, 50};
```

El resultat es mostra a la figura 1.2.

Figura 1.2. Esquema d'inicialització d'un "array" en Java amb valors concrets.



1.1.2 Inicialització a un valor per defecte

Moltes vegades les dades que es volen emmagatzemar depenen de la lectura d'una entrada (per exemple, per teclat), o bé són resultat de càlculs que són més còmodes de dur a terme de manera automatitzada dins del codi de programa (per exemple, una llista de 1.000 nombres primers). En aquest cas, no hi ha un valor inicial concret que calgui assignar a les posicions de l'*array*, ja que els valors s'emmagatzemaran *a posteriori*.

En aquest cas, una altra manera, més simple i habitual, de declarar i inicialitzar-lo, és assumint que totes les posicions prenen automàticament un valor per defecte, que és 0 per a les dades de tipus numèric i els caràcters, i `false` per a booleans. Més endavant ja assignareu a cada posició el valor que vulgueu.

Això es fa de la manera següent:

```
1 paraulaClauTipus[] identificadorVariable = new paraulaClauTipus[mida];
```

Només varia la part dreta, relativa a l'assignació dels valors inicials. En aquest cas la sintaxi és molt estricta i cal emprar el format indicat. S'usa la paraula clau `new`, seguida novament del tipus de les dades dins l'*array* i, entre claudàtors, la mida que vulguem. Els tipus de dades especificats a la part esquerra i dreta han de ser idèntics, o hi haurà un error de compilació.

No cal fer res més per assignar valors a les posicions de l'*array*, Java ja hi assigna automàticament el valor per defecte. Ara bé, alerta, aquesta és una propietat **específica** del Java i no ha de ser així necessàriament en altres llenguatges, en què el valor inicial de cada posició pot ser indeterminat.

Per exemple, per declarar un *array* de 10 posicions en què es poden emmagatzemar valors de tipus real, tots inicialment amb el valor 0, es faria de la manera següent:

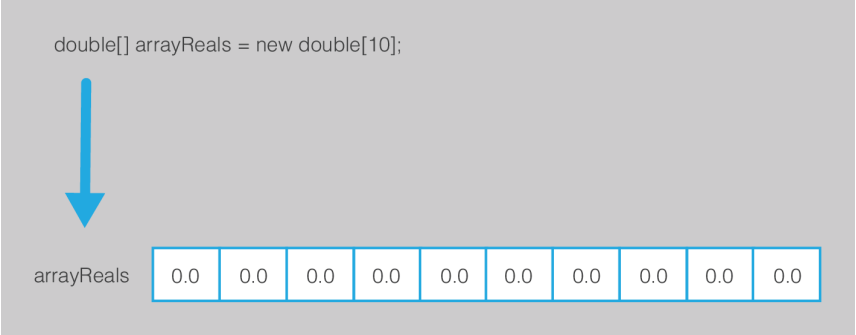
```
1 double[] arrayReals = new double[10];
```

Aquest codi és equivalent a fer:

```
1 double[] arrayReals = {0, 0, 0, 0, 0, 0, 0, 0, 0, 0};
```

El resultat de fer qualsevol de les dues instruccions es visualitza a la figura 1.3.

Figura 1.3. Esquema d'inicialització d'un "array" en Java amb valors per defecte



1.2 Manipulació de dades dins d'"arrays"

Tot i conformar un tipus de dada (com els enters, reals, etc.), els *arrays* són una mica especials a l'hora de ser manipulats, ja que no disposen de cap operació. No és possible usar l'identificador de l'*array* directament per invocar operacions i així manipular les dades contingudes, tal com es pot fer amb variables d'altres tipus. Per exemple, no és possible fer el següent:

En el Java, fer operacions entre *arrays* comporta un error de compilació.

```
...
int[] a = {10, 20, 30, 40, 50};
int[] b = {50, 60, 70, 80, 100};
int[] c = a + b;
...
```

Sempre que es vulgui fer alguna operació amb les dades emmagatzemades dins d'*arrays* cal manipular-les de manera individual, posició per posició. En aquest aspecte, cada posició d'un *array* té exactament el mateix comportament que una variable tal com les heu estudiades fins ara. Les podeu aprofitar tant per emmagatzemar dades com per llegir-ne el contingut.

Per tal de distingir entre les diferents posicions d'un *array*, cadascuna té assignat un **índex**, un valor enter que n'indica l'ordre dins de l'estructura. Sempre que us vulgueu referir a una de les posicions, n'hi ha prou d'usar l'identificador de l'*array* juntament amb l'índex de la posició buscada entre claudàtors `[]`. La sintaxi exacta és:

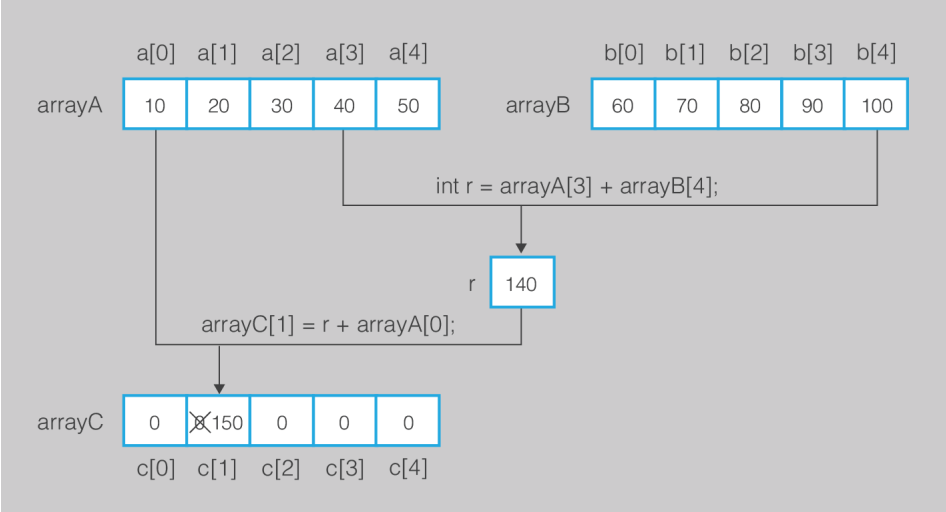
```
identificadorArray[índex]
```

El rang dels índexs pot variar segons el llenguatge de programació. En el cas del Java, aquests van de 0, per a la primera posició, a (mida - 1), per a la darrera. Per exemple, per accedir a les posicions d'un *array* de cinc posicions usaríeu els índexs 0, 1, 2, 3 i 4. Tot seguit es mostra un exemple de com s'accedeix a les dades d'un *array* mitjançant els índexs de les seves posicions:

```
...
int[] arrayA = {10, 20, 30, 40, 50};
int[] arrayB = {50, 60, 70, 80, 100};
int[] arrayC = new int[5];
//La variable de tipus enter "r" valdrà 140, 40 + 100.
int r = arrayA[3] + arrayB[4];
//La segona posició (índex 1) d'"arrayC" valdrà 150, 140 + 10.
arrayC[1] = r + arrayA[0];
...
```

La figura 1.4 esquematitza l'accés a les dades dins d'aquest codi.

Figura 1.4. Exemple de manipulació de dades dins d'"arrays"



És molt important que en accedir a un *array* s'utilitzi un índex correcte, d'acord amb el rang admissible. Si s'usa un valor no admissible, com per exemple un valor negatiu o un valor igual o superior a la seva mida, hi haurà un error d'execució. Concretament, es produirà una `IndexOutOfBoundsException`. Si això succeeix, podeu tenir la certesa que el programa té una errada i l'haureu de repassar i corregir. Per tant, els codi següent seria incorrecte:

```
...
//"arrayA" té mida 5.
//Els índexs vàlids van de 0 a (mida – 1), que és 4.
int[] arrayA = {10, 20, 30, 40, 50};
//S'assigna un valor a una posició incorrecta (índex 5).
arrayA[5] = 60;
//L'índex 6 és invàlid, no es pot consultar el valor.
int r = arrayA[2] + arrayA[4] + arrayA[6];
...
```

Per tal d'ajudar-vos en la tasca de controlar si un índex concret està dins del rang admissible per a una variable de tipus *array* concreta, Java disposa d'una eina auxiliar, l'atribut `length` (llargària). La seva sintaxi és la següent:

```
identificadorArray.length
```

El resultat d'aquesta instrucció és equivalent a avaluar una expressió que com a resultat obté la mida de l'*array* anomenat `identificadorArray`. Ara bé, recordeu que `length` us dóna la mida, però el rang d'índexs vàlid és 0 ... (mida - 1). Per exemple:

```
...
//"arrayA" té mida 5. Els índexs vàlids van de 0 a 4.
int[] arrayA = {10, 20, 30, 40, 50};
//"length" us diu la mida.
//Recordeu que l'índex màxim és (mida – 1).
if (index < arrayA.length) {
  System.out.println("La posició " + index + " val " + a[index]);
} else {
  System.out.println("Aquest array no té tantes posicions!");
}
...
```

1.3 Emmagatzematge de l'entrada de dades en un "array"

Com s'ha esmentat, els *arrays* són especialment útils per poder emmagatzemar de manera eficient un nombre arbitrari de dades provinents del sistema d'una entrada (per exemple, des del teclat). Normalment, les dades s'aniran llegint una per una, i a mesura que es faci, caldrà anar-les assignant a cadascuna de les posicions de l'*array* a partir del seu índex. Tot seguit veureu diferents esquemes per dur a terme aquesta tasca.

Emmagatzemar en "arrays" dades llegides seqüencialment és com una cadena d'embotellament

Si bé els exemples se centraran en l'ús del teclat, ja que és el sistema d'entrada que per ara sabeu utilitzar, heu de tenir en compte que aquests esquemes seran aplicables a qualsevol altre mecanisme d'entrada (per exemple, un fitxer). L'única condició és que les dades estiguin en forma de seqüència, de manera que es puguin anar llegint una per una, ordenadament.

Un cop exposats aquests esquemes, en la resta d'exemples de l'apartat s'obviarà el codi relatiu a l'entrada de dades pel teclat.

1.3.1 Entrada de seqüències de valors per teclat

Els *arrays* tenen sentit quan el programa ha de processar moltes dades, especialment provinents de l'entrada del programa (com el teclat). Abans de continuar, val la pena veure com es poden llegir seqüències de dades entrades des del teclat en una sola línia de text, de manera que sigui senzill emmagatzemar-les dins d'un *array*. Si bé ja sabeu com llegir dades individualment des del teclat, fer-ho així pot ser una mica avorrit i molest amb vista a l'execució dels programes en què s'introdueixen moltes dades, preguntant cada valor un per un i havent de pitjar la tecla de retorn cada vegada.

En realitat, quan s'usa una instrucció `lector.next...` i el programa s'atura esperant que l'usuari introdueixi dades pel teclat, res no impedeix que, en lloc d'una única dada, aquest n'escrigui més d'una abans de pitjar la tecla de retorn. O sigui, una seqüència de valors. L'única condició és que cada valor individual estigui separat de la resta per almenys un espai, de manera que siguin fàcilment identificables. Quan això succeeix, tots els valors de la seqüència queden latents, pendents de lectura. Successives invocacions a noves instruccions de lectura automàticament aniran consumint aquests valors pendents, en lloc de causar una nova espera d'una entrada de l'usuari. Mentre quedin valors a la seqüència pendents de llegir, les instruccions de lectura mai no causaran una espera d'entrada de dades. Un cop la seqüència ha estat consumida totalment, llavors sí que la propera instrucció de lectura causarà que el programa es torni a aturar esperant una nova entrada de l'usuari. I així el cicle de lectura torna a començar.

Dins d'aquest procés hi ha una crida amb un comportament especial: `lector.nextLine();`. Quan aquesta crida s'invoca, automàticament es descar-

ten tots els valors pendents de llegir de la seqüència actual. La propera crida a una instrucció de lectura sempre causarà que el programa s'aturi de nou i esperi una entrada de l'usuari.

La taula 1.1 mostra un exemple d'aquest mecanisme de lectura, en què es llegeixen valors de tipus enter.

Taula 1.1. Exemple de lectura d'una seqüència de valors enters

Acció del programa	Resultat obtingut	Dades pendents de lectura
(Inici del programa)	El programa s'inicia	{ } (no n'hi ha cap)
Llegir valor	El programa espera entrada de l'usuari	{ } (no n'hi ha cap)
	L'usuari entra els valors "1 2 4 8 16 32" i pitja la tecla de retorn	
	La lectura avalua 1	{ 2, 4, 8, 16, 32 }
Llegir valor	La lectura avalua 2	{ 4, 8, 16, 32 }
Llegir valor	La lectura avalua 4	{ 8, 16, 32 }
Llegir valor	Es buida la seqüència de dades pendents	{ }
Llegir valor	El programa espera entrada de l'usuari	{ }
Es torna a començar el cicle de lectura d'una nova seqüència de valors...		

Un fet important quan es llegeixen seqüències de diversos valors escrites des del teclat és que res no impedeix que hi pugui haver valors de diferents tipus de dades barrejats. Això tant pot ser per error de l'usuari en entrar les dades, com perquè el programa realment espera una seqüència amb valors de diferents tipus intercalats. En qualsevol cas, és important que abans de fer cap lectura d'una dada es comprovi si el tipus és el que correspon usant les instruccions `lector.hasNext...`.

Per llegir múltiples valors el més senzill és fer-ho mitjançant una estructura de repetició que vagi invocant successivament les instruccions de lectura de dades. En usar aquest mecanisme cal tenir present un fet ben important. S'ha de saber exactament quan heu obtingut ja totes les dades necessàries i cal deixar llegir. Normalment hi ha dues aproximacions depenent de si es coneix la quantitat de dades exactes que cal llegir o no.

Quantitat de dades coneguda

Si el nombre de valors que es vol llegir és conegut per endavant, per tractar la lectura de la seqüència de valors hi ha prou d'usar una estructura de repetició basada en un comptador. Aquest comptador controlarà el nombre de valors llegits i les iteracions finalitzaran en assolir el nombre de lectures que vulguem.

Compileu i executeu l'exemple següent, en què es mostra com cal fer la lectura d'un seguit de valors enters. Si se n'escriuen més dels esperats, la resta es descarten. Fixeu-vos que no és necessari escriure tots els valors en una sola línia de text. Si la línia introduïda, abans de pitjar la tecla de retorn, no disposa de suficients valors, el programa s'atura esperant la resta. A més a més, aquest codi

controla mitjançant una estructura de selecció si el valor que és a punt de ser llegit serà realment un enter o no.

```
import java.util.Scanner;
//Un programa que llegeix un nombre conegut de valors enters.
public class LectorValorsConeguts {
  //Es llegiran 10 valors.
  public static final int NUM_VALORS = 10;
  public static void main (String[] args) {
    Scanner lector = new Scanner(System.in);
    System.out.println("Escriu " + NUM_VALORS + " enters. Es pot fer en
        diferents línies.");
    //Es llegeixen exactament NUM_VALORS valors.
    int numValorsLlegits = 0;
    while (numValorsLlegits < NUM_VALORS) {
      //Abans de llegir, comprovem si realment hi ha un enter.
      if (lector.hasNextInt()) {
        int valor = lector.nextInt();
        System.out.println("Valor " + numValorsLlegits + " llegit: " + valor);
        numValorsLlegits++;
      } else {
        //Si el valor no és enter, es llegeix però s'ignora.
        //No s'avança tampoc el comptador.
        lector.next();
      }
    }
    //Els valors que sobrin a la darrera línia escrita es descarten.
    lector.nextLine();
    System.out.println("Ja s'han llegit " + NUM_VALORS + " valors.");
  }
}
```

Quantitat de dades desconeguda

Un cas més complex és quan el nombre de valors no és conegut *a priori*, ja que pot variar en diferents execucions del programa. Quan això succeeix, una solució simple seria preguntar simplement, abans de llegir cap valor, quantes dades s'introduiran, o bé fer que el primer valor dins de la seqüència n'indiqui la longitud.

Si això no és possible, o es considera que no s'ha de fer, una altra opció és establir un valor especial que no forma part de les dades per tractar dins el programa, sinó que es considerarà com a **marca de final de seqüència**. Tan bon punt es llegeixi aquest valor, la lectura s'ha de donar per finalitzada. La condició per poder usar aquesta estratègia és que la marca no sigui un valor que pugui aparèixer mai entre els valors introduïts. Ha de ser únic. Per exemple, dins una llista de notes, qualsevol valor negatiu o major que 10 serviria. La resta de valors no es poden usar, ja que poden correspondre a una nota real que cal tractar.

Si tots els valors possibles són acceptables dins de la seqüència una opció és usar un valor d'un tipus de dada diferent. Per exemple, un programa que processa valors reals arbitraris per fer càlculs matemàtics podria usar un caràcter com a marca de fi.

En aquest cas, l'estratègia per utilitzar és usar una estructura de repetició basada en semàfor. Aquest us indica si ja s'ha llegit la marca de fi i cal deixar d'iterar o no.

El codi següent d'exemple llegeix una seqüència de valors enters de llargària arbitrària. La lectura finalitza tan bon punt es llegeix el valor `-1`. Compareu aquest codi amb l'exemple del cas anterior.

```
1  import java.util.Scanner;
2  public class LectorValorsDesconeguts {
3    public static final int MARCA_FI = -1;
4    public static void main (String[] args) {
5      Scanner lector = new Scanner(System.in);
6      System.out.print("Escriu diferents valors enters.");
7      System.out.println("Després del darrer valor escriu un " + MARCA_FI);
8      //Es llegeixen exactament NUM_VALORS valors.
9      boolean marcaTrobada = false;
10     while (!marcaTrobada) {
11       //Abans de llegir, comprovem si realment hi ha un enter.
12       if (lector.hasNextInt()) {
13         int valor = lector.nextInt();
14         //És la marca de fi?
15         if (valor == MARCA_FI) {
16           //Sí que ho és.
17           marcaTrobada = true;
18         } else {
19           //No. És un valor que ha de ser tractat.
20           System.out.println("Valor llegit: " + valor);
21         }
22       } else {
23         //Si el valor no és enter, es llegeix però s'ignora.
24         lector.next();
25       }
26     }
27     //Els valors que sobrin a la darrera línia escrita es descarten.
28     lector.nextLine();
29     System.out.println("Ja s'han llegit tots els valors.");
30   }
31 }
```

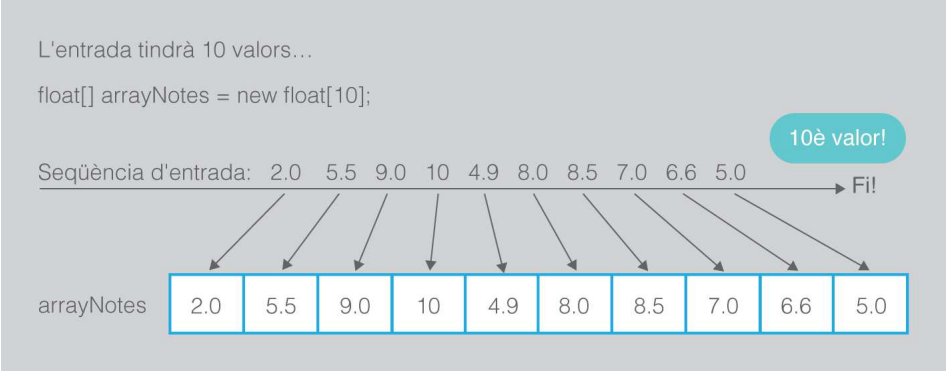
1.3.2 "Arrays" completament ocupats

L'esquema més simple d'emmagatzematge de dades des d'una entrada a les posicions d'un *array* és el cas en què la quantitat de dades que es llegirà és coneix prèviament. Això es deu a la important restricció que, per inicialitzar un *array*, prèviament se n'ha d'establir la mida. Aquest és un requisit indispensable, ja que en cas contrari qualsevol intent de desar-hi dades resultarà en un error d'execució. En aquest cas, n'hi ha prou d'inicialitzar l'*array* amb les seves posicions amb valor per defecte i una mida igual al nombre de dades que es llegiran. Un cop fet, només cal llegir de l'entrada un nombre de dades esperat i anar assignant els valors d'aquestes dades a cadascuna de les posicions de l'*array*, consecutivament.

Un fet que heu de tenir amb compte és que pot ser que s'indiqui que el nombre de dades per entrar és un valor, però després, per error, se'n proporcionin més. En qualsevol cas, mai no es poden llegir més dades que la capacitat real de l'*array*, per la qual cosa totes aquestes dades sobrants s'ignoraran. Un cop l'*array* està al màxim de la seva capacitat, s'ha d'acabar el procés de lectura o us exposeu a un error d'execució en sobrepassar la mida de l'*array*.

La figura 1.5 mostra un esquema de la lectura d'una seqüència de valors per ocupar totalment un *array*.

Figura 1.5. Esquema d'entrada de dades per crear un "array" totalment ocupat



Com a exemple, suposeu un programa per tractar les notes d'avaluació d'un conjunt d'estudiants. Per obtenir les diferents notes, aquest llegeix una seqüència, de llargària coneguda, composta de nombres reals. Amb els valors llegits s'omple completament un *array*. Compileu i executeu el codi següent per dur a terme aquesta operació. Fixeu-vos que, en aquest cas, es comproven un seguit d'errors possibles a l'entrada de les dades abans d'emmagatzemar el valor final a l'*array*: si els valors introduïts són realment de tipus real i si estan dins del rang vàlid per a una nota (entre 0 i 10).

```
1  import java.util.Scanner;
2  //Programa que llegeix una seqüència de valors reals, de longitud coneguda.
3  public class LectorSequencia {
4    public static void main (String[] args) {
5      Scanner lector = new Scanner(System.in);
6      //Llegeix la longitud de la seqüència. Comprova tots els errors.
7      int nombreValors = 0;
8      while (nombreValors <= 0) {
9        System.out.print("Quantes notes vols introduir? ");
10       if (lector.hasNextInt()) {
11         nombreValors = lector.nextInt();
12       } else {
13         //Si no és enter, es llegeix i s'ignora.
14         lector.next();
15       }
16     }
17     //Si s'han entrat més valors, s'ignoren. Només se'n necessita un.
18     lector.nextLine();
19     System.out.println("Es llegiran " + nombreValors + " valors reals.");
20     System.out.println("En pots escriure diversos en una sola línia, separats
           per espais.");
21     //Els desarem en un array. Ja en coneixem la mida.
22     float[] arrayNotes = new float[nombreValors];
23     //Cal llegir tants reals com la mida de l'array.
24     //Estructura de repetició amb comptador.
25     int index = 0;
26     while (index < arrayNotes.length) {
27       if (lector.hasNextFloat()) {
28         //S'ha llegit una nota, però és vàlida (entre 0 i 10)?
29         float nota = lector.nextFloat();
30         if ((nota >= 0)&&(nota <= 10)) {
31           arrayNotes[index] = nota;
32           index++;
33         }
34         //Si no és vàlida, la ignorem. No s'assigna enlloc.
35       } else {
36         //Si no era un real, simplement llegim el valor com una cadena de text
37         //però no en fem res. Es perd.
38         lector.next();
39       }
```

```
40      }
41      //Ignorem els valors sobrants de la darrera línia.
42      lector.nextLine();
43      System.out.println("La seqüència llegida és:");
44      for (int i = 0; i < arrayNotes.length;i++) {
45        System.out.println(arrayNotes[i]);
46      }
47    }
48  }
```

1.3.3 "Arrays" parcialment ocupats

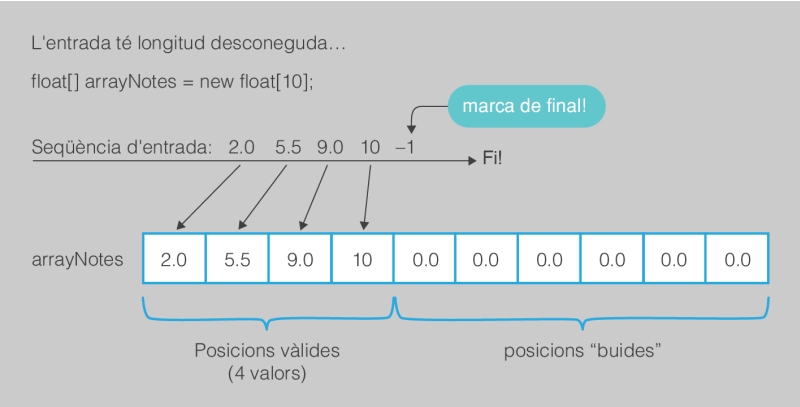
Malauradament, hi ha circumstàncies sota les quals no és possible inicialitzar un *array* de manera que la seva mida s'ajusti exactament a la llargària de la seqüència de valors d'entrada. Hi ha dos motius que poden dur a aquesta situació. D'una banda, si simplement no es coneix *a priori* el nombre de dades que s'entraran, ja que en lloc del programa es pregunta prèviament, i per tant l'entrada de dades és d'una llargària desconeguda. D'altra banda, això també succeeix en llenguatges de programació en què a l'hora de declarar un *array* i inicialitzar-lo no és possible usar una variable per especificar-ne la mida, i només es pot usar un literal.

Cada posició d'un *array* ocupa tanta memòria com el tipus primitiu que pot emmagatzemar.

En un cas com aquest, us haureu de conformar a triar un valor concret prefixat per a la mida de l'*array* i donar per fet que el nombre de dades d'entrada que el programa pot tractar és limitat. Si per algun motiu, en llegir les dades d'entrada, se supera aquest límit, com que no és possible emmagatzemar els valors excedents, els haureu d'ignorar. Per tant, caldrà triar aquesta mida arribant a un compromís entre un valor que sigui prou gran per no trobar-vos gaire sovint amb aquesta situació, però tampoc massa exagerat, per no malbaratar memòria. Per exemple, si ja sabeu que el nombre d'estudiants en una classe pot oscil·lar, però mai no es permetrà que n'hi hagi més de 80 (en aquest cas, s'obriria una nova aula), es pot triar aquest valor com a mida de l'*array*. Cal estudiar cada cas concret.

Col·loquialment, una posició en què no hi ha cap valor vàlid assignat es diu que està "buida".

Figura 1.6. Esquema d'entrada de dades que genera un "array" parcialment ocupat



En un cas com aquest, en què es disposa d'un *array* amb una mida que mai no serà exactament igual al nombre de dades llegides des de l'entrada (de fet, normalment, serà inferior), només hi ha un rang de posicions en què hi ha dades llegides de

l'entrada. La resta de posicions tenen valors per defecte que no tenen res a veure amb l'entrada, i per tant, no han de ser previstes a l'hora de fer cap tasca. La figura 1.6 esquematitza aquest fet partint de la seqüència d'entrada `2.0, 5.5, 9.0, 10`.

Un altre aspecte important és que l'atribut `length` ara no és suficient per saber el rang de les dades vàlides. Només indica la capacitat de l'*array*. Per tant, serà necessari disposar d'una variable auxiliar addicional, de tipus enter, en què s'emmagatzemi el nombre de dades que realment hi ha dins de l'*array*. Qualsevol accés a l'*array* només obtindrà dades correctes si l'índex emprat es troba entre 0 i aquest valor auxiliar - 1.

El codi següent és l'equivalent d'entrada de notes però per al cas d'una entrada de dades de llargària desconeguda. Com que no se sap quantes dades cal llegir, és necessari indicar d'alguna manera quan s'ha acabat d'entrar dades, amb una marca de final. En aquest cas, es tracta del valor `-1`. El límit de dades que és capaç d'emmagatzemar s'ha establert en 80, definit com la constant `MAX_VALORS`.

Proveu-lo al vostre entorn de treball i estudieu-ne el funcionament.

```
1  import java.util.Scanner;
2  //Llegeix un conjunt de dades sense saber quantes són.
3  public class LectorSequenciaDimDesconeguda {
4    public static final int MAX_VALORS = 80;
5    public static final int MARCA_FI = -1;
6    public static void main (String[] args) {
7      //No se sap quantes n'entraran. Cal fitar un valor màxim. Posem 80.
8      float[] arrayNotes = new float[MAX_VALORS];
9      //Cal un comptador de posicions actuals en què hi ha valors.
10     int elements = 0;
11     Scanner lector = new Scanner(System.in);
12     System.out.print("Escriure fins a " + MAX_VALORS + " valors.");
13     System.out.println("En pots escriure diversos en una sola línia, separats
           per espais.");
14     //Caldrà alguna manera de saber que s'ha acabat d'escriure.
15     System.out.println("(Per acabar, escriu un" + MARCA_FI + ")");
16     //Anar llegint fins a trobar la marca de fi. Però el màxim és la mida de l'
           array.
17     //Si s'entren més valors, els ignorem, ja que no queda lloc on emmagatzemar
           -los.
18     //Estructura de repetició amb semàfor.
19     while (elements < arrayNotes.length) {
20       //Hi ha un real?
21       if (lector.hasNextFloat()) {
22         //Cal veure si és un valor vàlid o final de seqüència.
23         float nota = lector.nextFloat();
24         if ((nota >= 0)&&(nota <= 10)) {
25           //Tot correcte. Ara hi ha un element més a l'array.
26           arrayNotes[elements] = nota;
27           elements++;
28         } else if (nota == MARCA_FI) {
29           //Marca de final, sortim del bucle.
30           break;
31         }
32         //Si no és ni una cosa ni l'altra, l'ignorem.
33       } else {
34         //Si no era un real, l'ignorem.
35         lector.next();
36       }
37     }
38     //Ignorem els valors sobrants de la darrera línia.
39     lector.nextLine();
40     System.out.println("La seqüència llegida és:");
```

```
41      for (int i = 0; i < elements;i++) {
42        System.out.println(arrayNotes[i]);
43      }
44    }
45  }
```

A un *array* parcialment ocupat se li poden afegir noves dades posteriorment, sempre que no se superi la seva mida. Per fer-ho, aquestes s'haurien d'assignar a partir de la posició en què es troba el darrer element vàlid.

1.4 Esquemes fonamentals en la utilització d'"arrays"

El potencial dels *arrays* va molt més enllà de la mera capacitat de gestionar conjunts de dades de grandària arbitrària tan sols declarant una única variable. Aquests també són una eina extremadament versàtil per processar totes les dades que contenen amb poques línies de codi de manera entenedora i eficient. El motiu principal és que les seves posicions són accessibles a partir d'índexs numèrics. Això en fa ideal el tractament usant estructures de repetició, que mitjançant un comptador accedeixin una per una a totes les posicions.

De fet, aquesta circumstància ja ha quedat patent en l'exemple de lectura d'una seqüència de dades des del teclat i l'emmagatzemament en un *array*. Reflexioneu sobre com seria el codi sense usar una estructura de repetició o *arrays*.

Tot seguit veureu un seguit d'estratègies basades en aquesta circumstància que val la pena que tingueu presents a l'hora de treballar amb les dades contingudes dins d'un *array*. Gairebé tots els programes basats en *arrays* n'usen alguna. Per simplificar el codi dels exemples utilitzats, en lloc de partir d'una entrada per teclat es treballarà sobre un *array* inicialitzat amb valors concrets. Els algorismes no varien en absolut respecte dels *arrays* producte d'una entrada per teclat.

1.4.1 Inicialització procedural

Hi ha situacions en les quals els valors inicials dins d'alguna o totes les posicions de l'*array* no necessàriament han de dependre d'una entrada, però usar el sistema d'inicialització amb valors concrets pot ser una mica incòmode. Per exemple, suposeu que us cal disposar dels valors de les 20 primeres potències de 2. En tractar-se de bastants valors, usar un *array* és ideal. Ara bé, tot i que aquests valors es poden expressar directament com a literals, primer els hauríeu de calcular pel vostre compte i, tot seguit, inicialitzar l'*array*. A més, el resultat final és una línia força llarga dins del codi, i n'afecta la llegibilitat.

Posats a haver de calcular prèviament per la vostra banda els valors amb els quals cal inicialitzar l'*array*, surt més a compte que sigui el programa mateix qui faci aquest càlcul i inicialitzi les posicions a mesura que trobi els resultats.

S'anomena **generació procedural** el fet de generar contingut algorísmicament, en lloc de fer-ho manualment partint de valors preestablerts. És un terme especialment usat dins del disseny de videojocs.

Com que quan es fa una inicialització d'aquest tipus el nombre de valors és conegut, es pot assignar una mida a l'*array* de manera que estigui totalment ocupat. Compileu i executeu el codi següent, que inicialitza proceduralment un *array* de manera que el valor a cada posició sigui el doble de l'anterior, partint d'una primera posició amb valor 1.

```
1  //Un programa que inicialitza proceduralment un array amb valors relacionats
       entre si.
2  public class InicialitzacióProcedural {
3    public static void main (String[] args) {
4      //Caldrà emmagatzemar 20 enters.
5      int[] arrayValorsDobles = new int[20];
6      //La primera posició la posem directament.
7      arrayValorsDobles[0] = 1;
8      //La resta es va omplint seqüencialment, a força de càlculs.
9      for(int i = 1; i < arrayValorsDobles.length; i++) {
10       arrayValorsDobles[i] = 2 * arrayValorsDobles[i - 1];
11     }
12    System.out.print("S'ha generat l'array: [ ");
13     for (int i = 0; i < arrayValorsDobles.length; i++) {
14       System.out.print(arrayValorsDobles[i] + " ");
15     }
16     System.out.print("]");
17   }
18 }
```

Per estudiar el funcionament del programa de manera més detallada, resulta útil usar una taula que indiqui l'evolució de les variables a cada iteració, tal com fa la taula 1.2.

Taula 1.2. Evolució del bucle per inicialitzar un "array" amb potències de 2

Iteració	Inici del bucle		Fi del bucle	
	'i' val	Condició val	Posició modificada	Valor assignat
1	1	(1 < 20), `true`	arrayValorsDobles[1]	2*arrayValorsDobles[0], 2 * 1 = 2
2	2	(2 < 20), `true`	arrayValorsDobles[2]	2*arrayValorsDobles[1], 2 * 2 = 4
3	3	(3 < 20), `true`	arrayValorsDobles[3]	2*arrayValorsDobles[2], 2 * 4 = 8
4	4	(4 < 20), `true`	arrayValorsDobles[4]	2*arrayValorsDobles[3], 2 * 8 = 16
...				
18	19	(19 < 20), `true`	arrayValorsDobles[19]	2*arrayValorsDobles[18], 2*262144 = 524288
19	20	(20 < 20), `false`	Ja hem sortit del bucle	

Repte 1: feu un programa que inicialitzi proceduralment un *array* amb els 100 primers nombres parells (Nota: el 0 és parell).

1.4.2 Recorregut

Recorregut

Parlem de recorregut quan cal tractar **tots** els valors de l'*array*.

Quan treballeu amb *arrays*, molt sovint haureu de fer càlculs en què estan implicats tots els valors emmagatzemats o, si més no, una bona part. En aquest cas, cal fer-ne un recorregut seqüencial de les posicions i anar-les processant per obtenir un resultat final. Un exemple d'una situació com aquesta pot ser calcular la mitjana aritmètica d'un seguit de notes, que requereix sumar els valors de totes i cadascuna i dividir el resultat entre el seu nombre.

El cas d'un *array* totalment ocupat és el més senzill, ja que és suficient recórrer l'*array* en tota la seva extensió fins arribar al final de l'extensió, donada per la mida.

Com a exemple, executeu el codi següent, que calcula la mitjana aritmètica d'un seguit de notes, expressades com a valors reals, contingudes dins d'un *array*.

```
1  //Un programa que fa un recorregut d'un array: càlcul de la mitjana aritmètica.
2  public class RecorregutTotalmentOcupat {
3    public static void main(String[] args) {
4      //Es parteix d'un array que conté tots els valors.
5      float[] arrayNotes =  {2f, 5.5f, 9f, 10f, 4.9f, 8f, 8.5f, 7f, 6.6f, 5f, 9f,
           7f};
6      //Acumulador de la suma de valors.
7      float acumulat = 0;
8      //Cal recórrer tot l'array d'extrem a extrem.
9      for(int i = 0; i < arrayNotes.length; i++) {
10       acumulat = acumulat + arrayNotes[i];
11     }
12     float resultat = acumulat / arrayNotes.length;
13     System.out.println("El resultat és " + resultat);
14   }
15 }
```

A la taula 1.3, mitjançant l'estudi de l'evolució del bucle, es pot veure com es van recorrent una per una totes les posicions de l'*array*.

Taula 1.3. Evolució del bucle per recórrer l'array i calcular la suma de tots els valors

Iteració	Inici del bucle		Fi del bucle	
	'i' val	Condició val	Posició accedida	'acumulat' val
1	0	(0 < 12), `true`	arrayNotes[0], 2.0	0 + 2.0 = 2.0
2	1	(3 < 12), `true`	arrayNotes[1], 5.5	2.0 + 5.5 = 7.5
3	2	(4 < 12), `true`	arrayNotes[2], 9.0	7.5 + 9.0 = 16.5
...				
12	11	(11 < 12), `true`	arrayNotes[11], 7.0	75.5 + 7.0 = 82.5
13	12	(12 < 12), `false`	Ja hem sortit del bucle	

En el cas d'un *array* parcialment ocupat cal anar amb compte, ja que el recorregut no necessàriament ha de ser fins a la darrera posició, sinó fins a la darrera posició ocupada. L'algorisme de recorregut a escala general serà el mateix, però la condició de sortida del bucle canviarà. Per exemple, si el nombre de valors en l'*array* es troba en la variable `darreraPosicio`, el codi seria:

```
...
//Només es tracten els valors realment vàlids.
for(int i = 0; i < darreraPosicio; i++) {
...
```

Repte 2: modifiqueu el codi del recorregut perquè, en lloc de la mitjana, calculi quina és la nota màxima.

1.4.3 Cerca

Cerca

Parlem de cerca quan només cal tractar els valors de l'*array* **fins a trobar** l'objectiu.

Una altra tasca que sovint es fa en treballar amb *arrays* és cercar valors concrets dins seu. Per això, cal anar mirant totes les posicions fins a trobar-lo. Per exemple, veure si entre un seguit de notes algú ha estat capaç de treure un 10.

Fins a cert punt, es pot considerar que una cerca no és més que un tipus especial de recorregut. En aquest cas, l'èmfasi es fa en la circumstància que no sempre cal recórrer tots els elements de l'*array*, només fins a trobar el valor que busquem. De fet, arribar al final de l'*array* és el que indica que no s'ha trobat el valor cercat.

Executeu el codi següent, que fa la cerca d'algun valor igual a 10 entre un seguit de notes. Fixeu-vos que en aquest codi se surt del bucle tan bon punt es troba el valor que busquem. Ja no cal seguir iterant. Tal com es presenta aquest codi, el 10 es troba a la quarta iteració. Podeu veure què passa si no se'n troba cap si abans de compilar modifiqueu manualment quest valor per un altre en la inicialització de l'*array*.

```
//Un programa per veure si algú ha tret un 10.
public class Cerca {
  public static void main(String args[]) {
    //Es parteix d'un array que conté tots els seus valors.
    float[] arrayNotes =  {2f, 5.5f, 9f, 10f, 4.9f, 8f, 8.5f, 7f, 6.6f, 5f, 9f,
         7f};
    //Semàfor. S'ha trobat?
    boolean trobat = false;
    //Comptador de posicions.
    int i = 0;
    //Es va mirant cada posició, mentre no s'arriba al final i no es trobi un
        10.
    while ((i < arrayNotes.length)&&(!trobat)) {
      if (arrayNotes[i] == 10) {
        trobat = true;
      }
      i = i + 1;
    }
    //S'ha trobat?
    if (trobat) {
      System.out.println("Algú ha tret un 10.");
    } else {
      System.out.println("Ningú no ha tret un 10");
    }
  }
}
```

Un cop més, una taula per veure l'evolució del bucle us pot ajudar a veure com, tan bon punt es troba el valor cercat, les iteracions acaben immediatament. Com

es pot veure a la taula 1.4, no cal recórrer tot l'*array*. A la darrera iteració se subratlla la part de l'expressió que causa que avaluï a `false`.

Taula 1.4. Evolució del bucle per recórrer l'array i calcular la suma de tots els valors

Iteració		Inici del bucle		Fi del bucle	
	'i' val	'trobat' val	Condició val	Posició accedida	'trobat' val
1	0	`false`	(0 < 12)&&(!false), `true`	arrayNotes[0], 2.0	`false`
2	1	`false`	(1 < 12)&&(!false), `true`	arrayNotes[1], 5.5	`false`
3	2	`false`	(2 < 12)&&(!false), `true`	arrayNotes[2], 9.0	`false`
4	3	`false`	(3 < 12)&&(!false), `true`	arrayNotes[3], 10.0	`true`
5	4	`true`	(4 < 12)&&(!true), `false`	Ja hem sortit del bucle	

Repte 3: modifiqueu el codi de l'exemple per poder veure si hi ha algú que ha suspès l'assignatura o no. Feu-ho usant la sentència `break`.

Si bé fins al moment els esquemes d'ús d'*arrays* s'han centrat en valors numèrics, també són aplicables a altres tipus primitius com els caràcters. Saber aplicar-los sobre conjunts de caràcters és especialment interessant per entendre les estratègies per tractar text. A títol il·lustratiu, tot seguit es mostra un exemple d'*array* basat en la cerca d'un caràcter concret. Observeu com el codi relatiu a la cerca, a grans trets, és idèntic al relatiu a valors numèrics. Només varia en els petits detalls relatius al tipus de dada que conté l'*array*.

```
1  //Cerca si hi ha la lletra 'w' entre un seguit de caràcters.
2  public class CercaCaracter {
3    public static void main (String[] args) {
4      //Es parteix d'un array que conté un seguit de caràcters.
5      char[] arrayNotes =  {'a', 'z', 'g', 'd', 'w', 'o', 'h', 'e', 'x', 's'};
6      //Semàfor. S'ha trobat?
7      boolean trobat = false;
8      //Comptador de posicions.
9      int i = 0;
10     //Es va mirant cada posició, mentre no s'arriba al final i no es trobi una
           "w".
11     while ((i < arrayNotes.length)&&(!trobat)) {
12       if (arrayNotes[i] == 'w') {
13         trobat = true;
14       }
15       i = i + 1;
16     }
17     //S'ha trobat?
18     if (trobat) {
19       System.out.println("La lletra 'w' és a la llista.");
20     } else {
21       System.out.println("La lletra 'w' no és a la llista.");
22     }
23   }
24 }
```

1.4.4 Ús d'"arrays" auxiliars

La utilització d'un *array* dins del codi d'un programa pot anar més enllà d'emmagatzemar seqüències de dades de longitud arbitrària obtingudes des d'una entrada. També són útils com a manera senzilla de gestionar molts valors diferents amb un únic identificador, en lloc d'haver de declarar diverses variables. Aquesta via és especialment útil per a casos en què cal gestionar dades que tenen un cert vincle entre elles. Una altra utilitat dels *arrays* es pot extreure de la seva capacitat d'accedir a les seves posicions usant un índex per simplificar el codi.

Per exemple, suposeu que voleu fer un programa que, a partir de les notes dels estudiants d'una aula, generi un gràfic de barres (o histograma) en què s'indiqui el nombre d'estudiants que han tret suspès, aprovat, notable o excel·lent. Per poder dur a terme aquesta tasca, us caldrà disposar d'un seguit de comptadors, un per a cada categoria de nota, i fer un recorregut de totes les notes. Segons el rang al qual pertanyi cada nota, s'incrementarà un comptador o un altre.

Tot seguit teniu el codi del programa que fa aquesta tasca usant un *array*, en lloc de quatre variables diferents. Compileu i proveu que funciona. En aquest cas, l'*array* de notes ja està donat directament, però podríeu adaptar el codi de manera que llegís els valors pel teclat.

```
1  public class Histograma {
2    public static void main (String[] args) {
3      float[] arrayNotes =  {2f, 5f, 9f, 6.5f, 10f, 4.5f, 8.5f, 7f, 6f, 7.5f, 9f,
           7f};
4      //Inicialització dels 10 comptadors, per a cada barra del gràfic.
5      int[] barres = new int[4];
6      //Càlcul dels comptadors. Es fa un recorregut de les notes.
7      for (int i = 0; i < arrayNotes.length; i++) {
8        //A quin rang pertany?
9        if ((arrayNotes[i] >=0 )&&(arrayNotes[i] < 5)) {
10         barres[0]++;
11       } else if (arrayNotes[i] < 6.5) {
12         barres[1]++;
13       } else if (arrayNotes[i] < 9) {
14         barres[2]++;
15       } else if (arrayNotes[i] <= 10) {
16         barres[3]++;
17       }
18       //Si no pertany a cap rang, nota incorrecta. La ignorem.
19     }
20     System.out.println("Gràfic de barres de les notes");
21     System.out.println("——————————————————————————");
22     //S'imprimeix el gràfic.
23     //Anem recorrent els comptadors i imprimim el valor en estrelles.
24     for (int i = 0; i < barres.length; i++) {
25       //Aprofitem l'índex per saber quin títol cal escriure a la barra actual.
26       switch(i) {
27         case 0:
28           System.out.print("Suspès:     ");
29           break;
30         case 1:
31           System.out.print("Aprovat:    ");
32           break;
33         case 2:
34           System.out.print("Notable:    ");
35           break;
36         case 3:
```

```
37            System.out.print("Excel·lent: ");
38            break;
39        }
40        //Imprimim tantes estrelles com el valor del comptador.
41        for (int j = 0; j < barres[i]; j++) {
42          System.out.print("*");
43        }
44        System.out.println();
45      }
46    }
47  }
```

En aquest cas, la granularitat del gràfic és baixa, i per això es podria fer amb quatre variables diferents sense problemes. Ara bé, hi ha una part del programa en què disposar d'un accés als quatre valors per índex resulta de gran utilitat per simplificar el codi: en escriure el gràfic per pantalla. Si s'usessin quatre variables, el codi per imprimir el gràfic seria:

```
1  ...
2      System.out.println("Gràfic de barres de les notes");
3      System.out.println("————————————————————————————————");
4      //S'imprimeix el gràfic.
5      //Primer els suspesos.
6      System.out.print("Suspès:     ");
7      for (int j = 0; j < comptadorSuspes; j++) {
8        System.out.print("*");
9      }
10     System.out.println();
11     //Ara els aprovats.
12     System.out.print("Aprovat:    ");
13     for (int j = 0; j < comptadorAprovats; j++) {
14       System.out.print("*");
15     }
16     System.out.println();
17     //Ara els notables.
18     System.out.print("Notable:    ");
19     for (int j = 0; j < comptadorNotable; j++) {
20       System.out.print("*");
21     }
22     System.out.println();
23     //Ara els excel·lents.
24     System.out.print("Excel·lent: ");
25     for (int j = 0; j < comptadorExcellent; j++) {
26       System.out.print("*");
27     }
28     System.out.println();
29     ...
```

Per escriure el resultat final en forma de gràfic cal escriure tantes estrelles com val cada comptador individual. Si s'usen quatre variables separades, caldria repetir el mateix codi per a cada variable. Usant un *array*, n'hi ha prou d'usar una estructura de repetició i fer un recorregut. A més a més, el valor de l'índex al qual s'està accedint també serveix per saber quin comptador s'està tractant i, per tant, quin títol cal escriure per a cada iteració. Ara imagineu que es vol modificar el programa per comptar la gent que ha tret notes dins de rangs amb més granularitat (0-1, 1-2, ..., 9-10).

Per tant, aquest és un exemple de cas en què usar *arrays* auxiliars pot ser de gran utilitat per no haver de repetir el codi.

1.4.5 Còpia

La no-creació d'una còpia real en dur a terme una assignació, sinó dues vies d'accés a una mateixa dada, és un tret comú a tots els tipus de dades compostos.

Com s'ha dit, el tipus de dada *array*, al contrari que els tipus de dades primitius, no té cap operació disponible. Només es pot operar en les posicions individuals. El que sí que està permès és fer assignacions usant directament les identificadors de les variables de tipus *array*. Ara bé, un fet certament curiós és que l'assignació no crea una còpia de les dades de l'*array* original, sinó que el que fa és configurar l'identificador de destinació perquè, a partir de llavors, per mitjà d'aquest s'accedeixi a **les mateixes dades** en memòria que l'identificador origen.

La millor manera de veure la diferència de comportament entre una assignació entre tipus de dades primitius i els *arrays* és amb un exemple de codi.

```
...
  //Assignació amb tipus primitius:
  int i = 0;
  int j = 5;
  //A la variable "i" se li assigna el valor de "j".
  i = j;
  //Es modifica el valor de "j".
  j = 10;
  //"i" conserva el valor copiat des de "j", 5.
  System.out.println(i);
  ...
```

```
...
  //Assignació amb arrays:
  int[] arrayA = {10, 20, 30, 40, 50};
  int[] arrayB = {60, 70, 80, 90, 100};
  //A la variable "arrayA" se li assigna el valor d'"arrayB".
  arrayA = arrayB;
  //Es modifica el valor de l'índex 2 només d'"arrayB".
  arrayB[2] = 60;
  //"arrayA" també ha vist modificat el seu valor a l'índex 2!
  System.out.println(arrayA[2]);
  //Es modifica el valor de l'índex 4 només d'"arrayA".
  arrayA[4] = 70;
  //"arrayB" també ha vist modificat el seu valor a l'índex 4!
  System.out.println(arrayB[4]);
  //De fet, "a" i "b" accedeixen exactament a les mateixes dades.
  ...
```

En fer assignacions entre variables de tipus *array*, l'origen i destinació ni tan sols han de tenir la mateixa mida.

Una pregunta que pot sorgir ara és: què ha passat amb les dades que hi havia inicialment a la variable "arrayB"? La resposta és que s'han perdut. Ja no hi ha manera d'accedir-hi. En el cas del Java, això no és cap inconvenient, ja que quan unes dades deixen d'estar accessibles, s'esborren de la memòria i el seu espai passa a estar disponible per a altres aplicacions. En altres llenguatges, les implicacions poden ser diferents, o fins i tot es considera un error greu de programació.

Tenint en compte el comportament de l'assignació, per generar una còpia independent d'un *array* cal tractar les posicions com a elements individuals, com passava amb la resta d'operacions.

```
//Un programa que crea una còpia idèntica, i independent, d'un altre array.
public class Copia {
  public static void main (String[] args) {
```

```
    float[] llistaNotes = {2f, 5.5f, 9f, 10f, 4.9f, 8f, 8.5f, 7f, 6.6f, 5f, 9f,
         7f};
    //La còpia ha de tenir la mateixa mida que l'original.
    float[] copiaLlistaNotes = new float[llistaNotes.length];
    //Anem copiant posició per posició.
    for (int i = 0; i < llistaNotes.length; i++) {
      copiaLlistaNotes[i] = llistaNotes[i];
    }
    //Mostrem la còpia per pantalla.
    System.out.print("L'array copiat és: [ ");
    for (int i = 0; i < llistaNotes.length;i++) {
      System.out.print(copiaLlistaNotes[i] + " ");
    }
    System.out.println("]");
  }
}
```

1.4.6 Canvi de mida

Un cop us heu decidit per un valor concret per a la mida d'un *array*, ja no hi ha volta enrere. No es pot canviar. Això té certs inconvenients, ja que pot passar que, en determinades condicions, us quedeu sense espai per poder emmagatzemar dades. Però si bé no és possible canviar la mida d'un *array*, aquest problema no és irresoluble. Només cal seguir les passes següents:

1. Generar un nou *array* en una variable diferent, amb una mida superior a l'original.
2. Copiar-hi les dades de l'*array* original en el mateix ordre. En fer-ho, el nou *array* quedarà amb posicions "buides".
3. Aprofitant les propietats de l'assignació entre *arrays*, s'assigna el nou *array* a l'*array* original.
4. Finalment, afegir al nou *array* els valors extra que ja no cabien en l'original.

Arribats a aquest punt, és com si a efectes pràctics s'hagués fet més gran l'*array* original. El codi següent mostra un exemple d'aquest esquema.

Un exemple de bocí de codi que fa aquestes accions seria el que es mostra a continuació. En aquest, la mida de l'*array* original s'incrementa de manera que s'hi afegeixen `MAX_VALORS`, noves posicions.

```
...
float[] arrayNotes = new float[...];
...
//PAS 1
float[] arrayNou = new float[arrayNotes.length + MAX_VALORS];
//PAS 2
for (int i = 0; i < arrayNotes.length; i++) {
 arrayNou[i] = arrayNotes[i];
}
//PAS 3
arrayNotes = arrayNou;
//Ara ja es poden desar MAX_VALORS nous valors extra dins d'''arrayNotes''
...
```

El codi següent adapta la lectura per teclat d'una seqüència de llargària desconeguda de manera que, si se supera la capacitat de l'*array* en què s'emmagatzemen els valors, la seva mida s'incrementa per poder encabir-ne de nous. Compileu-lo i executeu-lo per veure com funciona.

```
import java.util.Scanner;
public class CanviMida {
  public static final int MAX_VALORS = 5;
  public static void main (String[] args) {
    //D'entrada, la mida serà 5.
    //S'escull un valor molt baix perquè de seguida es forci el canvi de mida.
    float[] arrayNotes = new float[MAX_VALORS];
    //Cal un comptador de posicions en què hi ha valors vàlids.
    int elements = 0;
    Scanner lector = new Scanner(System.in);
    System.out.print("Vés escrivint notes (valors reals entre 0 i 10).");
    System.out.println("En pots escriure diversos en una sola línia, separats
        per espais.");
    //Caldrà alguna manera de saber que s'ha acabat d'escriure.
    System.out.println("(Per acabar, escriu un -1)");
    //Anar llegint fins a trobar la marca de fi. Però el màxim és la mida de l'
        array.
    //Si s'entren més valors, canviem la mida de l'array i continuem llegint.
    //Estructura de repetició amb semàfor.
    boolean marcaFi = false;
    while (!marcaFi) {
      //Hi ha un real?
      if (lector.hasNextFloat()) {
        //Cal veure si és un valor vàlid o final de seqüència.
        float nota = lector.nextFloat();
        if ((nota >= 0)&&(nota <= 10)) {
          //Tot correcte. Però hi ha lloc a l'array?
          if (elements == arrayNotes.length) {
            //Cal canviar la mida de l'array. Es fa 5 posicions més llarg.
            //Per veure quan succeeix, avisem per pantalla.
            System.out.println("Capacitat superada. Afegim " + MAX_VALORS + "
                posicions...");
            float[] arrayNou = new float[arrayNotes.length + MAX_VALORS];
            for (int i = 0; i < arrayNotes.length; i++) {
              arrayNou[i] = arrayNotes[i];
            }
            arrayNotes = arrayNou;
          }
          //Afegim el nou valor. Segur que hi ha lloc.
          arrayNotes[elements] = nota;
          elements++;
        } else if (nota == -1) {
          //Marca de final, sortim del bucle.
          break;
        }
        //Si no és ni una cosa ni l'altra, ho ignorem.
      } else {
        //Si no era un real, l'ignorem.
        lector.next();
      }
    }
    //Ignorem els valors sobrants de la darrera línia.
    lector.nextLine();
    System.out.println("A l'array hi ha emmagatzemats " + elements + " elements
        : ");
    for (int i = 0; i < elements;i++) {
      System.out.println(arrayNotes[i]);
    }
  }
}
```

Cal dir que fer aquest tipus de tasca és especialment ineficient i té un impacte sobre el rendiment del programa. És millor només usar aquest esquema si realment no hi ha més remei. Si es fa, normalment, a l'hora de triar la mida del nou *array* és millor no ajustar massa i deixar espai de sobres per omplir noves posicions si és necessari sense haver-vos de veure forçats a repetir aquest procés constantment.

1.4.7 Ordenació

Hi ha vegades en què resulta molt útil reorganitzar els valors que hi ha dins l'*array* de manera que estiguin ordenats de manera ascendent o descendent. En molts casos, mostrar les dades de manera ordenada és simplement un requisit de l'aplicació, de manera que l'usuari les pugui visualitzar i extreure'n conclusions de manera més còmoda. Però hi ha casos en què disposar de les dades ordenades facilita la codificació del programa i l'eficiència.

Per exemple, en aquest mateix apartat heu vist com cercar, a partir d'un seguit de notes, si algun estudiant ha tret un 10. Per fer aquesta tasca, calia una estructura de repetició i anar mirant un per un tots els valors dins l'*array*. Però si les dades estan ordenades en ordre ascendent, llavors podem usar una estratègia més eficaç. Si algú té un 10, aquesta nota estarà per força a la darrera posició. Per tant, tota la cerca queda reduïda directament al codi:

```
...
if (arrayNotes[arrayNotes.length - 1] == 10) {
  System.out.println("Algú ha tret un 10.");
} else {
  System.out.println("Ningú no ha tret un 10");
}
...
```

De cop i volta el codi es fa més ràpid i curt, ja que no cal l'estructura de repetició. En general, les dades ordenades resulten d'utilitat a l'hora de fer cerques o recorreguts en què les dades per tractar tenen un rang o un valor concret, ja que permeten establir estratègies que minimitzin els nombre d'iteracions dels bucles.

L'algorisme d'ordenació més ràpid i eficient que hi ha és el QuickSort.

Per ordenar una seqüència de valors hi ha diversos algorismes. Els més eficients són força complexos i es troben més enllà dels objectius d'aquest mòdul. N'hi ha prou de veure'n un de més senzill.

> L'**algorisme de la bombolla** (*BubbleSort*) serveix per ordenar elements mitjançant recorreguts successius al llarg de la seqüència, en què es van comparant parelles de valors. Cada cop que es troben dues parelles en ordre incorrecte, s'intercanvia la seva posició.

Bàsicament, el que es fa és cercar el valor més baix, recorrent tot l'*array*, i posar-lo a la primera posició. Amb això ja tenim la garantia d'haver resolt quin valor ha d'acabar a la primera posició. Llavors se cerca el segon valor més baix, recorrent la seqüència des de la segona posició fins al final, i es posa a la segona posició.

I l'operació es va repetint fins a arribar a la darrera posició. Per fer aquesta tasca s'opera directament sobre els valors emmagatzemats a l'*array*, intercanviant-los cada cop que trobem un valor més baix que el que hi ha a la posició que s'està tractant.

El codi seria el següent. Analitzeu-lo per entendre què està fent i proveu que funciona correctament:

```
1  //Un programa per ordenar valors usant l'algorisme de la bombolla.
2  public class Ordenar {
3    public static void main (String[] args) {
4      float[] llistaNotes = {5.5f, 9f, 2f, 10f, 4.9f};
5      //Bucle extern.
6      //S'anirà posant a cada posició tractada, començant per la 0,
7      //el valor més baix que es trobi.
8      for (int i = 0; i < llistaNotes.length - 1; i++) {
9        //Bucle intern.
10       //Se cerca entre la resta de posicions quin és el valor més baix.
11       for(int j = i + 1; j < llistaNotes.length; j++) {
12         //La posició tractada té un valor més alt que el de la cerca. Els
               intercanviem.
13         if (llistaNotes[i] > llistaNotes[j]) {
14           //Per intercanviar valors cal una variable auxiliar.
15           float canvi = llistaNotes[i];
16           llistaNotes[i] = llistaNotes[j];
17           llistaNotes[j] = canvi;
18         }
19       }
20     }
21     //El mostrem per pantalla.
22     System.out.print("L'array ordenat és: [");
23     for (int i = 0; i < llistaNotes.length;i++) {
24       System.out.print(llistaNotes[i] + " ");
25     }
26     System.out.println("]");
27   }
28 }
```

Com que aquest algorisme d'ordenació es basa en dos bucles imbricats, val la pena fer una ullada més detallada al seu funcionament. La taula 1.5 mostra un seguit d'iteracions fins a tenir en les dues primeres posicions els valors més baixos. Fixeu-vos com el valor de la variable comptador `j` sempre és superior a `i`. A les cel·les en què s'indica l'estat de l'*array* després de cada iteració, se subratllen les posicions que s'han intercanviat, si és el cas.

Taula 1.5. Evolució de l'algorisme de la bombolla

Inici del bucle				**Fi del bucle**
'i' val	**'j' val**	**'llistaNotes' val**	**llistaNotes[i] > llistaNotes[j]**	**'llistaNotes' val**
Inici primera iteració bucle extern. Se cerca el valor més petit per posar-lo a la posició 0.				
0	1	{5.5, 9, 2, 10, 4.9}	5.5 > 9, `false`	{5.5, 9, 2, 10, 4.9}
0	2	{5.5, 9, 2, 10, 4.9}	5.5 > 2, `true`	{2, 9, 5.5, 10, 4.9}
0	3	{2, 9, 5.5, 10, 4.9}	2 > 10, `false`	{2, 9, 5.5, 10, 4.9}
0	4	{2, 9, 5.5, 10, 4.9}	2 > 4.9, `false`	{2, 9, 5.5, 10, 4.9}
Fi primera iteració bucle extern. A `llistaNotes[0]` hi ha el valor més petit.				
Inici segona iteració bucle extern. Se cerca el segon valor més petit per posar-lo a la posició 1.				
1	2	{2, 9, 5.5, 10, 4.9}	9 > 5.5, `true`	{2, 5.5, 9, 10, 4.9}
. . . . .				

Taula 1.5 (continuació)

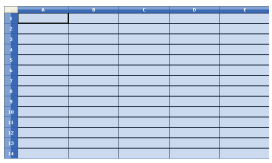
Inici del bucle				Fi del bucle
1	3	{2, 5.5, 9, 10, 4.9}	5.5 > 10, `false`	{2, 5.5, 9, 10, 4.9}
1	4	{2, 5.5, 9, 10, 4.9}	5.5 > 4.9, `true`	{2, 4.9, 9, 10, 5.5}
	Fi segona iteració bucle extern. A `llistaNotes[1]` hi ha el segon valor més petit.			
	Inici tercera iteració bucle extern. Se cerca el tercer valor més petit per posar-lo a la posició 2.			
2	3	{2, 4.9, 9, 10, 5.5}	9 > 10, `false`	{2, 4.9, 9 , 10, 5.5}
2	4	{2, 4.9, 9, 10, 5.5}	9 > 5.5, `true`	{2, 4.9, 5.5, 10, 9}
	Fi tercera iteració bucle extern. A `llistaNotes[2]` hi ha el tercer valor més petit.			
	Inici quarta iteració bucle extern. Se cerca el quart valor més petit per posar-lo a la posició 3.			
3	4	{2, 4.9, 5.5, 10, 9}	10 > 9, `true`	{2, 4.9, 5.5, 9, 10}
	Fi tercera iteració bucle extern. A `llistaNotes[3]` hi ha el quart valor més petit.			
	Per força, a la posició 4 hi ha el valor més alt.			
4	4	Hem sortit del bucle extern (4 > llistaNotes.length - 1)		

Repte 4: modifiqueu l'exemple d'ordenació de manera que, un cop ordenat l'*array*, es calculi la mitjana aritmètica només dels estudiants que han suspès. El bucle del recorregut ha de fer exactament tantes iteracions com estudiants suspesos hi hagi, i no pas com tota la mida de l'*array*. Reflexioneu atentament sobre com el fet que l'*array* estigui ordenat és determinant per assolir aquesta darrera condició.

1.5 "Arrays" multidimensionals

Quan es declara un *array*, hi ha una propietat especial, a part del seu identificador de tipus i mida, que fins al moment s'ha obviat. Es tracta de la seva **dimensió**, el nombre d'índexs que cal usar per establir la posició que es vol tractar. Tots els *arrays* amb els quals heu treballat fins ara eren **unidimensionals**. N'hi havia prou amb un sol valor d'índex per indicar la posició que es volia accedir. Ara bé, res no impedeix que el nombre d'índexs sigui més gran que 1.

L'ordre dels índexs és important.



Un "array" de dimensió 2: una taula

Un **array multidimensional** és aquell en què per accedir a una posició concreta, en lloc d'usar un sol valor com a índex, s'usa una seqüència de diversos valors. Cada índex serveix com a coordenada per a una dimensió diferent.

Entre els *arrays* multidimensionals, el cas més usat és el dels *arrays* bidimensionals, de dues dimensions. Aquest és un dels més fàcils de visualitzar i és, per tant, un bon punt de partida. Com que es tracta d'un tipus d'*array* en què calen dos índexs per establir una posició, es pot considerar que el seu comportament és com el d'una matriu, taula o full de càlcul. El primer índex indicaria la fila, mentre que el segon indicaria la columna on es troba la posició que es vol accedir.

Per exemple, suposeu que el programa per tractar notes d'estudiants ara ha de gestionar les notes individuals de cada estudiant per a un seguit d'exercicis, de manera que es poden fer càlculs addicionals: calcular la nota final segons els resultats individuals, veure l'evolució de cada estudiant al llarg del curs, etc. En aquest cas, tota aquesta informació es pot resumir de manera eficient en una taula o full de càlcul, en què els estudiants es poden organitzar per files, on cada columna correspon al valor d'una prova, i en què la darrera columna pot ser la nota final.

1.5.1 Declaració d'"arrays" bidimensionals

La sintaxi per declarar i inicialitzar un *array* de dues dimensions és semblant a la dels unidimensionals, si bé ara cal especificar que hi ha un índex addicional. Això es fa usant un parell de claus extra, `[]`, sempre que calgui especificar un nou índex.

Per declarar-ne un d'inicialitzat amb totes les posicions dels seus valors per defecte, caldria usar la sintaxi:

```
1 paraulaClauTipus[] identificadorVariable = new paraulaClauTipus[nombreFiles][
      nombreColumnes];
```

Com passava amb els *arrays* unidireccionals, el valor dels índexs, files i columnes, no ha de ser necessàriament un literal. Pot ser definit d'acord amb el contingut d'una variable, diferent per a cada execució del programa. El valor del nombre de files i columnes tampoc no ha de ser necessàriament el mateix.

La inicialització mitjançant valors concrets implica indicar tants valors com el nombre de files per columnes. Per fer-ho, primer s'enumeren els valors individuals que hi ha a cada fila de la mateixa manera que s'inicialitza un *array* unidimensional: una seqüència de valors separats per comes i envoltats per claudàtors, `{...}`. Un cop enumerades les files d'aquesta manera, aquestes s'enumeren al seu torn separades per comes i envoltades per claudàtors. Conceptualment, és com declarar un *array* de files, en què cada posició té un altre *array* de valors. Això es veu més clar amb la sintaxi, que és la següent:

```
1 paraulaClauTipus[][] identificadorVariable = {
2                                              {Fila0valor1, Fila0valor2, ... , Fila0valorN},
3                                              {Fila1valor1, Fila1valor2, ... , Fila1valorN},
4                                              ...,
5                                              {FilaNvalor1, FilaNvalor2, ... , FilaNvalorN}
6                                              };
```

Fixeu-vos com a l'inici i al final hi ha els claudàtors extra que tanquen la seqüència de files i després de cada fila hi ha una coma, per separar-les entre si. Per fer més clara la lectura, s'han usat diferents línies de text per indicar els valors emmagatzemats a cada fila, però res no impedeix escriure-ho tot en una sola línia molt llarga. De tota manera, és recomanable usar aquest format i intentar deixar espais on pertoqui de manera que els valors quedin alineats verticalment, ja que això facilita la comprensió del codi font.

Per exemple, per declarar i inicialitzar amb valors concrets un *array* bidimensional de tres files per cinc columnes de valors de tipus enter es faria:

```
int[][] arrayBidi = {
                    {1 ,2 ,3 ,4 ,5 },
                    {6 ,7 ,8 ,9 ,10},
                    {11,12,13,14,15}
                    };
```

Des del punt de vista d'una taula o matriu, La primera fila seria {1, 2, 3, 4, 5}, la segona {6 ,7 ,8 ,9 ,10} i la tercera {11,12,13,14,15}, tal com mostra la taula 1.6.

Taula 1.6. Resultat d'inicialitzar amb valors concrets un array bidimensional de tres files per cinc columnes

1	2	3	4	5
6	7	8	9	10
11	12	13	14	15

En usar aquest tipus d'inicialització cal assegurar-se que les mides de totes les files siguin exactament igual (que tinguin el mateix nombre de valors separats entre comes), ja que cal garantir que totes les files tenen el mateix nombre de columnes.

1.5.2 Esquemes d'ús d'"arrays" bidimensionals

L'accés a les posicions d'un *array* bidimensional és pràcticament idèntic a l'unidireccional, però tenint en compte que ara hi ha dos índexs per preveure referents a les dues coordenades.

```
identificadorVariable[índexFila][índexColumna]
```

La taula 1.7 fa un resum de com s'accediria a les diferents posicions d'un *array* de 4 * 5 posicions. En qualsevol cas, en accedir-hi, sempre cal mantenir present que cada índex mai no pot ser igual o superior al nombre de files o columnes, segons correspongui. En cas contrari, es produeix un error.

Taula 1.7. Aquesta taula mateixa és un array bidimensional de 4 * 5 posicions

a[0][0]	a[0][1]	a[0][2]	a[0][3]	a[0][4]
a[1][0]	a[1][1]	a[1][2]	a[1][3]	a[1][4]
a[2][0]	a[2][1]	a[2][2]	a[2][3]	a[2][4]
a[3][0]	a[3][1]	a[3][2]	a[3][3]	a[3][4]

Assegureu-vos d'usar variables diferents per indicar cada índex.

Com que ara hi ha dos índexs, tots els esquemes fonamentals per treballar amb *arrays* bidimensionals es basen en dues estructures de repetició imbricades, una per tractar cada fila i l'altra per tractar cada valor individual dins de la fila.

En el cas d'*arrays* bidimensionals, l'atribut `length` té un comportament una mica especial, ja que ara hi ha dos índexs. Per entendre'l, cal recordar que per inicialitzar l'*array* el que es fa és enumerar una llista de files, dins de les quals hi ha els valors emmagatzemats realment. Per tant, el que us diu l'atribut és el nombre de files que hi ha.

Per saber el nombre de valors d'una fila heu de fer:

```
identificadorVariable[índexFila].length
```

Per exemple, si la taula 1.7 correspon a una variable anomenada `arrayBidi`, `arrayBidi.length` avalua a 4 (hi ha quatre files) i `arrayBidi[0].length`, `arrayBidi[1].length`, etc. tots avaluen a 5 (hi ha 5 columnes).

Inicialització procedural

Novament, es pot donar el cas en què vulgueu assignar a cada posició valors que cal calcular prèviament i que, per tant, calgui anar assignant un per un a totes les posicions de l'*array* bidimensional.

A mode d'exemple, compileu i executeu el codi font següent, que emmagatzema a cada posició la suma dels índexs d'un *array* bidimensional de dimensions arbitràries i després visualitza el resultat per pantalla, ordenat per files.

```
import java.util.Scanner;
//Un programa que inicialitza un array bidimensional.
public class InicialitzacioBidi {
  public static void main (String[]  args) {
    Scanner lector = new Scanner(System.in);
    //Llegeix les files.
    int nombreFiles = 0;
    while (nombreFiles <= 0) {
      System.out.print("Quantes files tindrà la taula? ");
      if (lector.hasNextInt()) {
        nombreFiles = lector.nextInt();
      } else {
        lector.next();
        System.out.print("Aquest valor no és correcte. ");
      }
    }
    lector.nextLine();
    //Llegeix les columnes.
    int nombreColumnes = 0;
    while (nombreColumnes <= 0) {
      System.out.print("Quantes files tindrà la taula? ");
      if (lector.hasNextInt()) {
        nombreColumnes = lector.nextInt();
      } else {
        lector.next();
        System.out.print("Aquest valor no és correcte. ");
      }
    }
    lector.nextLine();
    //Inicialització amb valors per defecte.
    int[][] arrayBidi = new int[nombreFiles][nombreColumnes];
    //Observeu l'ús de l'atribut "length".
    System.out.println("Hi ha " + arrayBidi.length + " files.");
    System.out.println("Hi ha " + arrayBidi[0].length + "columnes.");
    //Bucle per recórrer cada fila.
    //"i" indica el número de fila.
    for(int i = 0; i < nombreFiles; i++) {
      //Bucle per recórrer cada posició dins de la fila (columnes de la fila).
      //"j" indica el número de fila.
      for (int j = 0; j < nombreColumnes; j++)  {
        //Valor assignat a la posició: suma dels índex de fila i columna.
        arrayBidi[i][j] = i + j;
      }
    }
    //Es visualitza el resultat, també calen dos bucles.
```

```
46      for(int i = 0; i < nombreFiles; i++) {
47        //Inici de fila, obrim claudàtors.
48        System.out.print("Fila " + i + " { ");
49        for (int j = 0; j < nombreColumnes; j++)  {
50          System.out.print(arrayBidi[i][j] + " ");
51        }
52        //Al final de cada fila es tanquen claudàtors i es fa un salt de línia.
53        System.out.println("}");
54      }
55    }
56  }
```

Recorregut

En el cas d'*arrays* bidimensionals, també pot passar que sigui necessari fer càlculs fent un recorregut per tots els valors continguts. Novament, cal anar fila per fila, mirant totes les posicions de cadascuna.

Per exemple, suposeu un programa en què es desa a cada fila el valor de les notes dels estudiants d'un curs. Es demana calcular la nota final de cada estudiant i emmagatzemar-la a la darrera columna, i també saber la mitjana de totes les notes finals. El codi es mostra tot seguit; analitzeu-lo i proveu que funciona. En aquest cas, per facilitar-ne l'execució, els valors de les notes estan inicialitzats per a valors concrets. En el cas de la nota final de cada estudiant, inicialment es deixa a 0 i serà el programa el qui la calculi.

Ara sí: observeu atentament l'ús de l'atribut `length` per controlar els índexs en recórrer l'*array* i saber quin és el nombre de valors en una fila o el nombre de files.

```
1  //Un programa que calcula notes mitjanes en un array bidimensional.
2  public class RecorregutBidi {
3    public static void main (String[] args) {
4      //Dades de les notes.
5      float[][] arrayBidiNotes = {
6                                  { 4.5f, 6f  , 5f  , 8f  , 0f  },
7                                  { 10f , 8f  , 7.5f, 9.5f, 0f  },
8                                  { 3f  , 2.5f, 4.5f, 6f  , 0f  },
9                                  { 6f  , 8.5f, 6f  , 4f  , 0f  },
10                                 { 9f  , 7.5f, 7f  , 8f  , 0f  }
11                                 };
12     //Mitjana aritmètica del curs per a tots els estudiants.
13     float sumaFinals = 0f;
14     //Es fa tractant fila per fila, indicada per "i". Cada fila és un estudiant
           .
15     //"arrayBidiNotes.length" avalua al nombre de files.
16     for(int i = 0; i < arrayBidiNotes.length; i++) {
17       //Aquí s'acumulen les notes de l'estudiant tractat.
18       float sumaNotes = 0f;
19       //Tractem cada fila (cada estudiant). Cada nota la indexa "j".
20       //"arrayBidiNotes[i].length" avalua al nombre de posicions de la fila.
21       for(int j = 0; j < arrayBidiNotes[i].length; j++) {
22           //Estem a la darrera posició de la fila?
23         if(j != (arrayBidiNotes[i].length - 1)) {
24           //Si no és la darrera posició, anem acumulant valors.
25           sumaNotes = sumaNotes + arrayBidiNotes[i][j];
26         } else {
27           //Si ho és, cal escriure la mitjana.
28           //Hi ha tantes notes com la mida d'una fila - 1.
29           float notaFinal =  sumaNotes/(arrayBidiNotes[i].length - 1);
30           arrayBidiNotes[i][j] = notaFinal;
31           System.out.println("L'estudiant " + i + " ha tret " + notaFinal +
                 ".");
```

```
          //S'actualitza la suma de mitjanes de tots els estudiants.
          sumaFinals = sumaFinals + notaFinal;
        }
      }
      //Fi del tractament d'una fila.
    }
    //Fi del tractament de totes les files.
    //Es calcula la mitjana: suma de notes finals dividit entre nombre d'
        estudiants.
    float mitjanaFinal = sumaFinals / arrayBidiNotes.length;
    System.out.println("La nota mitjana del curs és " + mitjanaFinal);
  }
}
```

Cerca

En *arrays* bidimensionals la cerca també implica haver de gestionar dos índexs per anar avançant per files i columnes. Ara bé, aquest cas és especialment interessant, ja que en assolir l'objectiu cal sortir de les dues estructures de repetició imbricades. Per tant, no es pot usar simplement una sentència `break`, ja que només serveix per sortir d'un únic bucle. Cal tenir un semàfor que s'avaluï en tots dos bucles.

Per exemple, suposeu que el programa de gestió de notes vol cercar si algun estudiant ha tret un 0 en algun dels exercicis al llarg del curs. Caldrà anar mirant fila per fila totes les notes de cada estudiant, i quan es trobi un 0, acabar immediatament la cerca. El codi per fer-ho, basant-se exclusivament en un semàfor que diu si ja s'ha trobat el valor, seria el següent. Proveu-lo al vostre entorn de treball.

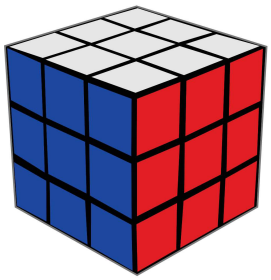
En un cas com aquest, cal anar amb molt de compte a inicialitzar i incrementar correctament l'índex per recórrer la posició de cada fila (és a dir, que mira cada columna), ja que en usar una sentència `while` en lloc de `for` no es fa automàticament. Si us despisteu i no ho feu, el valor serà incorrecte per a successives iteracions del bucle que recorre les files i el programa no actuarà correctament.

```
//Un programa que cerca un 0 entre els valors d'un array bidimensional.
public class CercaBidi {
  public static void main (String[] args) {
    //Dades de les notes.
    float[][] arrayBidiNotes = {
                               { 4.5f, 6f  , 5f  , 8f  },
                               { 10f , 8f  , 7.5f, 9.5f},
                               { 3f  , 2.5f, 0f  , 6f  },
                               { 6f  , 8.5f, 6f  , 4f  },
                               { 9f  , 7.5f, 7f  , 8f  }
                               };
    //Mirarem quin estudiant ha tret el 0.
    //Inicialitzem a un valor invàlid (de moment, cap estudiant té un 0)
    //Aquest valor fa de semàfor. Si pren un valor diferent, cal acabar la
        cerca.
    int estudiant = -1;
    //"i indica la fila.
    int i =0;
    //Es va fila per fila.
    //S'acaba si s'ha trobat un 0 o si ja s'ha cercat a totes les files.
    while ((estudiant == -1)&&(i < arrayBidiNotes.length)){
      //"j indica la "columna".
      int j =0;
```

```
23      //Se cerca en una fila.
24      //S'acaba si s'ha trobat un 0 o si ja s'ha cercat a totes les posicions.
25      while ((estudiant == -1)&&(j < arrayBidiNotes[i].length)){
26        //Aquesta nota és un 0?
27        if  (arrayBidiNotes[i][j] == 0f) {
28          //L'índex que diu l'estudiant és el de la fila.
29          estudiant = i;
30        }
31        //S'avança a la posició següent dins de la fila.
32        j++;
33      }
34      //Fi del tractament d'una fila.
35      //S'avança a la fila següent.
36      i++;
37    }
38    //Fi del tractament de totes les files.
39    if (estudiant == -1) {
40      System.out.println("Cap estudiant no té un 0.");
41    } else {
42      System.out.println("L'estudiant " + estudiant + " té un 0.");
43    }
44  }
45 }
```

Repte 5: modifiqueu el programa de manera que se cerqui si algú ha tret un 0, però només en el segon exercici (posició 1 de cada fila). Reflexioneu atentament sobre si ara realment cal usar dues estructures de repetició o no.

1.5.3 "Arrays" de més de dues dimensions

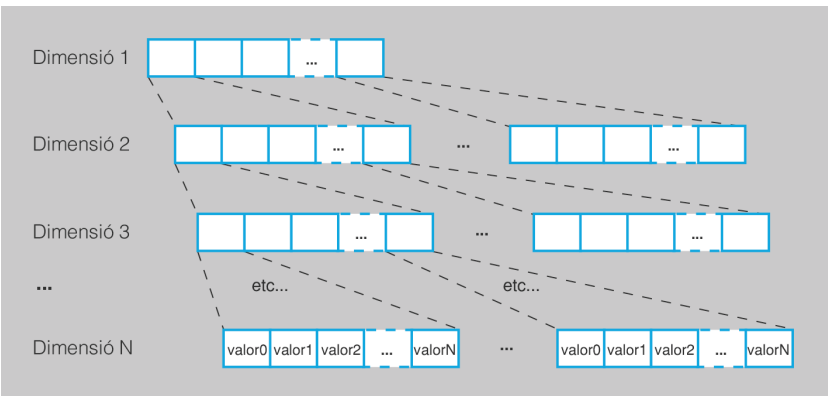


Un "array" amb tres dimensions: el cub de Rubik. Imatge de Wikimedia Commons

El cas dels *arrays* bidimensionals és el més comú, però res no impedeix que el nombre d'índexs necessaris per situar una posició sigui també més gran de dos, d'un valor arbitrari. Un *array* de tres dimensions encara es pot visualitzar com una estructura tridimensional en forma de cub, però dimensions superiors ja requereixen més grau d'abstracció, i per això només es presentaran, sense entrar en més detall.

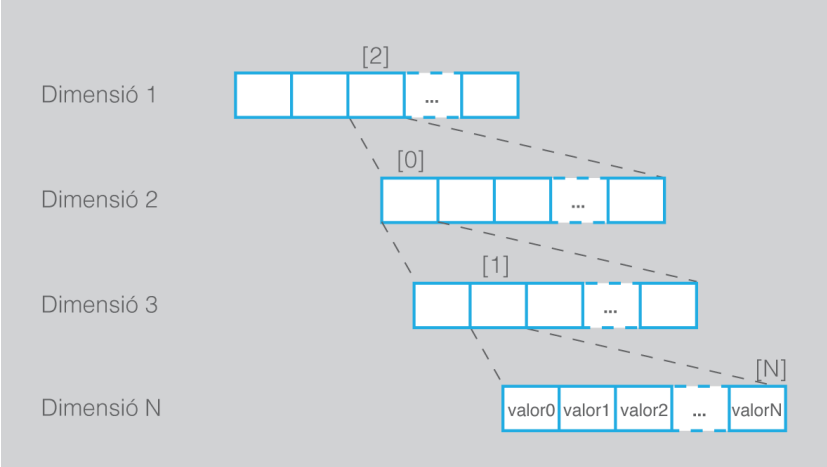
Conceptualment, es pot considerar que defineixen una estructura d'*arrays* imbricats a diferents nivells com el que es mostra a la figura 1.7. Cada nivell correspon a una dimensió diferent.

Figura 1.7. Un "array" multidimensional: "arrays" dins d'"arrays" dins d'"arrays"...



Cada índex individual identifica una casella de l'*array* triat d'una dimensió determinada, de manera que el conjunt d'índexs serveix com a coordenada per establir el valor final (que ja no serà un altre *array*) al qual es vol accedir. La figura 1.8 en mostra un exemple de quatre dimensions.

Figura 1.8. Accés a un "array" de quatre dimensions (calen 4 índexs)



Tot i que aquesta estructura sembla força complexa, no és gaire diferent d'una estructura de carpetes i fitxers, en què les carpetes prenen el paper dels *arrays* i els fitxers el de les dades que es volen desar. Podeu tenir una única carpeta per emmagatzemar totes les fotos, que seria el cas d'un únic nivell d'*array* (o sigui, els *arrays* tal com s'han vist fins ara). Però també les podríeu organitzar amb una carpeta que al seu torn conté altres carpetes, dins de les quals es desen diferents fotos. Això seria equivalent a disposar de dos nivells d'*arrays*. I així podríeu anar creant una estructura de carpetes amb tants nivells com volguéssiu. Per triar un fitxer n'hi ha prou a saber en quin ordre cal anar obrint les carpetes fins a arribar a la carpeta final, on ja només us cal l'ordre del fitxer.

Estenent les restriccions que compleixen els *arrays* bidireccionals al cas dels *arrays* multidimensionals:

- En una mateixa dimensió, tots els *arrays* tindran exactament la mateixa mida.
- El tipus de dada que es desa a les posicions de la darrera dimensió serà el mateix per a tots. Per exemple, no es poden tenir reals i enters barrejats.

En llenguatge Java, per afegir noves dimensions per sobre de la segona, cal anar afegint claus `[]` addicionals a la declaració, una per a cada dimensió.

```
1 paraulaClauTipus[]...[] identificadorVariable = new paraulaClauTipus[midaDim1
      ]...[midaDimN];
```

Per exemple, per declarar amb valors per defecte un *array* de quatre dimensions, la primera de mida 5, la segona de 10, la tercera de 4 i la quarta de 20, caldria la instrucció:

```
...
int[][][][] arrayQuatreDimensions = new int[5][10][4][20];
...
```

L'accés seria anàleg als *arrays* bidimensionals, però usant quatre índexs envoltats per parells de claus.

```
...
int valor = arrayQuatreDimensions[2][0][1][20];
...
```

En qualsevol cas, es tracta d'estructures complexes que no se solen usar excepte en casos molt específics.

1.6 Solucions dels reptes proposats

Repte 1:

```
public class InicialitzaParells {
  public static void main(String[] args) {
    //Caldrà emmagatzemar 100 enters.
    int[] arrayParells = new int[100];
    //Anem omplint cada posició.
    for(int  i = 0; i < arrayParells.length; i++) {
      arrayParells[i] = 2*i;
    }
  }
}
```

Repte 2:

```
public class RecorregutMaxima {
  public static void main(String[] args) {
    //Es parteix d'un array que conté tots els seus valors.
    float[] arrayNotes =  {2f, 5.5f, 9f, 10f, 4.9f, 8f, 8.5f, 7f, 6.6f, 5f, 9f,
         7f};
    //Valor màxim fins al moment.
    float notaMaxima = 0;
    //Cal recòrrer tot l'array d'extrem a extrem i desar el valor més gran.
    for(int i = 0; i < arrayNotes.length; i++) {
      if (notaMaxima < arrayNotes[i]) {
        notaMaxima = arrayNotes[i];
      }
    }
    System.out.println("La nota màxima és " + notaMaxima);
  }
}
```

Repte 3:

```
public class CercaSuspes {
  public static void main (String[] args) {
    //Es parteix d'un array que conté tots els seus valors.
    float[] arrayNotes =  {2f, 5.5f, 9f, 10f, 4.9f, 8f, 8.5f, 7f, 6.6f, 5f, 9f,
         7f};
    int i = 0;
    //Es va mirant cada posició, mentre no s'arriba al final i no es trobi un
        suspès.
    //Es pot usar tant "break" com "for".
    //En trobar un suspès, directament sortim del bucle.
    while (i < arrayNotes.length) {
      if (arrayNotes[i] < 5) {
        break;
      }
      i++;
    }
    if (i >  arrayNotes.length) {
      //Si s'ha superat la mida de l'array, és que no s'ha trobat cap suspès.
      System.out.println("Ningú no ha suspès.");
    } else {
      //Si s'ha sortit abans d'acabar l'array, és que hi havia un suspès.
      System.out.println("Algú ha suspès.");
    }
  }
}
```

Repte 4:

```java
public class MitjanaSuspesosOrdenats {
  public static void main (String[] args) {
    float[] llistaNotes = {5.5f, 9f, 2f, 10f, 4.9f};
    //Bucle extern.
    //S'anirà posant a cada posició tractada, començant per la 0,
    //el valor més baix que es trobi.
    for (int i = 0; i < llistaNotes.length - 1; i++) {
      //Bucle intern.
      //Se cerca entre la resta de posicions quin és el valor més baix.
      for(int j = i + 1; j < llistaNotes.length; j++) {
        //La posició tractada té un valor més gran que el de la cerca; els intercanviem.
        if (llistaNotes[i] > llistaNotes[j]) {
          //Per intercanviar valors cal una variable auxiliar.
          float canvi = llistaNotes[i];
          llistaNotes[i] = llistaNotes[j];
          llistaNotes[j] = canvi;
        }
      }
    }
    //El mostrem per pantalla.
    System.out.print("L'array ordenat és: [");
    for (int i = 0; i < llistaNotes.length;i++) {
      System.out.print(llistaNotes[i] + " ");
    }
    System.out.println("]");
    //Inici de les modificacions.
    //Valor acumulat per fer la mitjana.
    float acumulat = 0;
    //Nombre de suspesos.
    int numSuspes = 0;
    //Inici de la nova part.
    //Com l'array està ordenat, només cal fer un recorregut fins trobar un aprovat,
    //no cal arribar al final. Un cop es troba el primer aprovat, la resta de
    //posicions segur que també són aprovats, ja que l'array està ordenat.
    for (int i = 0; i < llistaNotes.length; i++) {
      //És la nota d'un aprovat?
      if (llistaNotes[i] >= 5) {
        //S'ha trobat el primer aprovat, sortim del bucle!
        break;
      } else {
        //És un suspès, anem calculant.
        acumulat = acumulat + llistaNotes[i];
        //S'incrementa el nombre de suspesos.
        numSuspes++;
      }
    }
    float resultat = acumulat/numSuspes;
    System.out.println("La nota mitjana dels suspesos és " + resultat);
  }
}
```

Repte 5:

```
public class CercaBidiZero {
  public static void main (String[] args) {
    //Dades de les notes.
    float[][] arrayBidiNotes = {
                               { 4.5f, 6f  , 5f  , 8f  },
                               { 10f , 8f  , 7.5f, 9.5f},
                               { 3f  , 2.5f, 0f  , 6f  },
                               { 6f  , 8.5f, 6f  , 4f  },
                               { 9f  , 7.5f, 7f  , 8f  }
                               };
    //Calen dues estructures de repetició si els dos índexs han d'anar
    //variant per recórrer totes les posicions. En aquest cas, un valor és
    //sempre el mateix (posició 1 de cada fila); només varia el número de fila.
    //Per tant, amb una única estructura n'hi ha prou.
    //Mirarem quin estudiant ha tret el 0.
    int estudiant = -1;
    for (int numFila = 0; numFila < arrayBidiNotes.length; numFila++) {
      if (arrayBidiNotes[numFila][1] == 0) {
        //Aquest estudiant té un 0.
        estudiant = numFila;
        //Ja podem sortir del bucle.
        //Ho podem fer, ja que només hi ha una única estructura de repetició.
        break;
      }
    }
    if (estudiant == -1) {
      System.out.println("Cap estudiant no té un 0.");
    } else {
      System.out.println("L'estudiant " + estudiant + " té un 0 en el segon
          exercici.");
    }
  }
}
```

2. Tractament de cadenes de text

Fins ara, el tipus de dada caràcter no ha estat gaire rellevant a l'hora d'exposar exemples significatius de codi font de programes. És força habitual que en el vostre dia a dia tracteu amb conjunts de dades numèriques individuals, cadascuna independent de les altres: dates, distàncies, intervals de temps, quantitats, etc. Per tant, són dades que té sentit emmagatzemar i processar mitjançant un programa per automatitzar-ne el càlcul. En canvi, segurament no és gaire habitual que constantment useu conjunts de caràcters individuals i independents: lletres 'a', 'b', 'c', etc. La veritable utilitat dels caràcters és poder usar-los agrupats en forma de paraules o frases: un nom, una pregunta, un missatge, etc. Per tant, si un llenguatge de programació ha de ser útil, s'ha de tenir en compte aquesta circumstància.

El tipus de dada compost **cadena de text** o **String** serveix per representar i gestionar de manera senzilla una seqüència de caràcters.

String

En Java, el tipus de dada compost cadena de text s'identifica amb la paraula clau `String`.

2.1 La classe "String"

En llenguatge Java, la manipulació de qualsevol dada de tipus compost, incloent-hi les cadenes de text, té un seguit de particularitats i una terminologia força diferent de la dels tipus primitius o els *arrays*. Si bé no és necessari conèixer-ne tots els detalls, abans de veure com manipular cadenes de text o altres tipus de dades compostos és important aprendre un seguit de termes específics, ja que apareixen amb freqüència a la documentació de Java.

L'ús dels termes classe i objecte està estretament lligat a la característica de llenguatge Java de ser orientat a objectes.

En Java, exceptuant el cas de l'*array*, s'usa el terme **classe** per referir-se a un tipus de dada compost.

A les variables de tipus de dades compostos també els podem assignar directament valors, com a les de tipus primitius. Ara bé, amb algunes particularitats en la nomenclatura.

S'usa el terme **objecte** per referir-se al valor assignat a una variable d'un tipus de dades compost.

En llenguatge Java, la classe que representa una cadena de text s'anomena explícitament `String`. Així, doncs, a partir d'ara, en lloc de parlar del "tipus de dada compost `String`", direm sempre "la classe `String`", i en lloc de dir que

Per convenció de codi, els noms de les classes al Java s'escriuen sempre usant *UpperCamelCase*.

una variable "és del tipus de dada compost `String`" es dirà que "és de la classe o que pertany a la classe `String`". Per norma general, també es parlarà de "l'objecte assignat a la variable `holaMon`" en lloc d'"el valor assignat a la variable `holaMon`", o d'"un objecte de la classe `String`" en lloc d'"un valor de tipus `String`".

En realitat, els conceptes de classe i objecte van molt més enllà del que s'ha exposat aquí, però això ja ens val per poder explicar com manipular cadenes de text de manera que se segueixi estrictament la terminologia Java tal com s'usa a les seves fonts de referència.

2.1.1 Inicialització d'objectes "String"

Treballar amb classes no modifica en absolut la sintaxi per declarar una variable, però sí que pot tenir algunes implicacions a l'hora d'inicialitzar-la, ja que cal assignar un objecte. Com es representa un objecte pot variar depenent de la classe emprada. Afortunadament, en el cas de la classe `String`, la sintaxi per declarar i inicialitzar una variable és idèntica a com es faria amb un tipus primitiu qualsevol, ja que és possible representar un objecte mitjançant un literal directament. Els literals assignables a una variable de la classe `String` prenen la forma de qualsevol combinació de text envoltada entre cometes dobles, ''...''. Un exemple de declaració i inicialització d'una variable que conté una cadena de text seria:

```
1 String holaMon = "Hola, món!";
```

Aquesta instrucció declara una variable anomenada `holaMon`, pertanyent a la classe `String`, i li assigna l'objecte que representa la cadena de text "Hola, món!".

Val la pena comentar que hi ha el concepte de cadena de text "buida", és a dir, que no conté absolutament cap caràcter. Es representa simplement no posant res entre les cometes dobles. Expressat amb codi seria:

```
1 String cadenaBuida = "";
```

2.1.2 Manipulació d'objectes

A diferència dels tipus primitius, qualsevol objecte emmagatzemat en una variable pertanyent a una classe (és a dir, qualsevol valor emmagatzemat en una variable d'un tipus de dada compost), no es pot manipular directament mitjançant operadors de cap tipus. No hi ha operadors que manipulin o comparin directament objectes.

En alguns casos, la restricció en l'ús d'operacions ja pot resultar evident de manera intuïtiva. Aquest és el cas dels operadors aritmètics o lògics. Per exemple, amb quin resultat avaluaria una divisió de cadenes de text (objectes de la classe `String`) o la seva negació lògica? Ara bé, en el cas dels operadors relacionals, tot i que

sí que pot tenir sentit intentar comparar dues cadenes de text, usar-los també és incorrecte. Per tant, les expressions següents no són vàlides.

- `unString * unAltreString`
- `unString && unAltreString`
- `unString < unAltreString`
- `unString == unAltreString`
- etc.

En el cas de la igualtat cal tenir cura especial, ja que el compilador de Java no ho considera un error de sintaxi.

En Java, la manipulació d'objectes es fa mitjançant la **invocació de mètodes** sobre les variables en què es troben emmagatzemats.

Podeu trobar la llista de mètodes de la classe `String` en el web `http://download.oracle.com/javase/6/docs/api/java/lang/String.html`.

La llista de mètodes que poden ser invocats sobre una variable on s'emmagatzema un objecte depèn exclusivament de la seva classe. Classes diferents permeten la invocació de diferents mètodes. Per exemple, la classe `String` especifica el conjunt de mètodes que és possible invocar sobre una variable on hi ha assignada una cadena de text.

Per generar una instrucció en què es crida un mètode, s'usa l'identificador de la variable, seguida d'un punt, i llavors el nom del mètode que volem, escollit entre la llista que ofereix la classe a la qual pertany la variable. Depenent del mètode utilitzat, pot ser necessari indicar un seguit d'informació addicional, en forma de llista de valors entre parèntesis, separats per comes `(..., ..., ...)`. Aquesta llista és el que s'anomenen els seus **paràmetres**. Si el mètode no requereix cap paràmetre, només cal obrir i tancar parèntesis, `()`. La sintaxi seria:

```
1 identificadorVariable.nomMètode(paràmetre1, paràmetre2,..., paràmetreN);
```

Per convenció de codi, els noms dels mètodes de les classes del Java estan escrits sempre usant *lowerCamelCase*.

Escriure una invocació d'un mètode sobre una variable és equivalent a escriure una expressió que avalua un cert resultat. Cada mètode diferent fa una operació o transformació diferent sobre l'objecte emmagatzemat en aquella variable. Per saber què fa cadascun i a quin tipus de dada pertany el seu resultat, no queda més remei que mirar la documentació de Java.

La invocació d'un mètode té la màxima precedència en avaluar una expressió.

La màxima precedència d'una invocació d'un mètode té especialment sentit en aquest cas, ja que fins que el seu resultat concret no s'ha avaluat, no es disposa de cap valor a partir del qual es pugui continuar avaluant una expressió més complexa. Per exemple, en el codi següent, en què s'usa el mètode `length()`, que avalua la mida d'un objecte de la classe `String`, el resultat és 43, per l'ordre de precedència

dels operadors. Fins que no s'ha avaluat `length()` i s'ha esbrinat que la mida del text és 22, ni tant sols seria possible iniciar el càlcul de la multiplicació. Això evidencia que la seva precedència ha de ser la màxima.

```
public class Precedencia {
  public static void main (String[] args) {
    //El text té una mida de 22 caràcters.
    String text = "Hi havia una vegada...";
    //ordre: 1) mètode, 2) multiplicació, 3) resta.
    int dobleMidaMenysUn = 2 * text.length() - 1;
    System.out.println(dobleMidaMenysUn);
  }
}
```

2.1.3 Accés als caràcters d'una cadena de text

Per la manera com organitzen els conjunts de caràcters que contenen, les cadenes de text són molt semblants als *arrays*. En el fons, no deixen de ser una seqüència de tipus primitius, en aquest cas caràcters (`char`), en què cada lletra ocupa una posició concreta. Per tant, molts dels aspectes generals aplicables als *arrays* en el Java també es poden aplicar a les cadenes de text. Novament, l'accés es fa per l'índex de la posició que ocupa cada caràcter, començant des de 0 i fins a la seva llargària - 1.

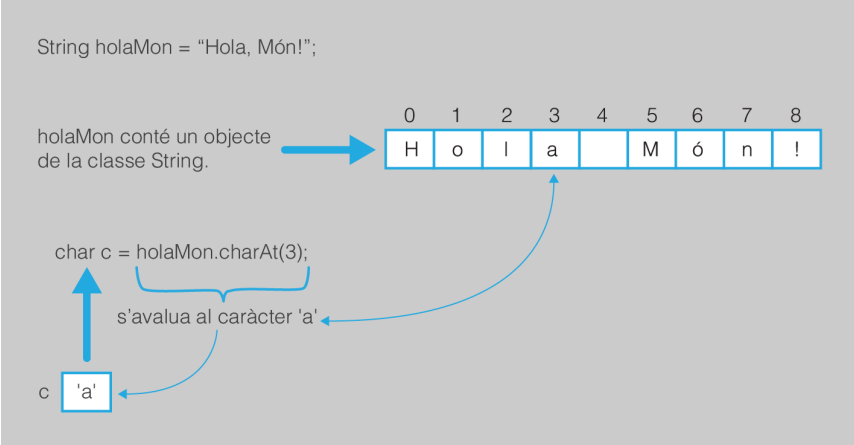
Ara bé, atès que les cadenes de text s'emmagatzemen en Java en variables de la classe `String` en forma d'objectes, l'accés o manipulació de les seves dades s'ha de fer mitjançant la invocació d'algun dels mètodes definits en la classe `String`. Per al cas de l'accés a caràcters individuals, el mètode que cal usar és l'anomenat `charAt(numPosició)`. Aquest mètode requereix un únic paràmetre de tipus enter en què s'indica quina posició cal consultar, i avalua igual el caràcter en aquella posició.

Per exemple, veure quin caràcter és a la tercera posició d'una cadena de text es faria de la manera següent. Fixeu-vos que la invocació d'un mètode és equivalent a avaluar una expressió:

```
...
String holaMon = "Hola, món!";
char c = holaMon.charAt(3);
//A la variable "c" hi ha el caràcter 'l'
System.out.println("A la posició 3 hi ha el caràcter " + c);
...
```

La figura 2.1 mostra el comportament de la invocació en el mètode `charAt`. Fixeu-vos que, en el fons, aquest és equivalent a avaluar una expressió. El seu resultat pot ser immediatament assignat a una variable, o usat en una altra expressió.

Figura 2.1. Avaluació de la invocació del mètode charAt



Com amb els *arrays*, la posició també ha de ser un valor vàlid o es produirà un error. Per evitar això, en Java les cadenes de text també incorporen un mecanisme per establir quina és la seva llargària, similar a l'atribut `length` dels *arrays*. Es tracta del mètode `length()`. Alerta, fixeu-vos que, en tractar-se d'un mètode sense paràmetres, immediatament al final porta dos parèntesis, `()`. En el cas dels *arrays*, aquests parèntesis no s'usen.

Per exemple, el codi següent escriu en línies diferents els caràcters individuals de la cadena de text “Hola, món!”. Proveu que és així.

```
1  //Mostra els caràcters individuals en una cadena de text.
2  public class MostraCaracters {
3    public static void main (String[] args) {
4      String holaMon = "Hola, món!";
5      //Es recorren les posicions de la cadena de text una per una.
6      for (int i = 0; i < holaMon.length(); i++) {
7        System.out.println(holaMon.charAt(i));
8      }
9    }
10 }
```

Finalment, heu de tenir en compte que un punt molt important de l'ús dels mètodes és el fet que aquests **es comporten exactament igual que expressions**. Per tant, no és possible fer assignacions sobre aquests. En el cas de les variables de la classe `String` pot ser fàcil confondre's, ja que el seu comportament és molt semblant als *arrays*, però no del tot. Tot i que si partiu del funcionament d'un *array* podria semblar lògic fer el següent, en realitat és un error. Seria com si, operant amb valors enters, intentéssiu assignar un valor al resultat d'una expressió suma:

```
1  ...
2  holaMon.charAt(3) = 'b'
3  //Erroni. No és realment igual que fer "array[3] = ...;".
4  //Seria més aviat com intentar fer "(3 + 4) = a;".
5  ...
```

De fet, la classe `String` de Java no ofereix cap mètode per canviar el valor del caràcter en una posició determinada. **Repte 1:** feu un programa que mostri per pantalla una cadena de text, però escrita del revés. Per exemple, si es parteix de la cadena “Hola, món!”, per pantalla ha de mostrar ”!nóm ,aloH”.

2.1.4 Concatenació de cadenes de text

Si bé s'ha dit que els objectes no es poden manipular mitjançant operadors, únicament mitjançant la invocació de mètodes, això no és del tot cert. Hi ha un únic cas en què sí que és possible aplicar l'operació de suma entre objectes. Es tracta de la concatenació de cadenes de text (objectes de la classe `String`). De fet, aquest és un cas realment molt excepcional, ja que, si us hi fixeu, també es permet fer aquesta operació entre cadenes de text i dades de diferents tipus primitius, cosa que en teoria tampoc està permesa. Cap altra mena d'objectes permet en Java aquesta mena de comportament.

El resultat de la concatenació amb cadenes de text sempre dóna com a resultat una nova cadena de text, que es pot usar tant per generar missatges mostrats directament per pantalla com per fer assignacions a variables de la classe `String`. Per tant, recordeu que es pot fer:

```
1  //Mostra el resultat d'una divisió simple entre reals.
2  public class Dividir{
3    public static void main(String[] args) {
4      double dividend = 18954.74;
5      double divisor = 549.12;
6      double resultatDivisio = dividend/divisor;
7      //S'assigna el resultat de la concatenació a una variable de la classe
           string.
8      String missatge = "El resultat obtingut és " + resultatDivisio + ".";
9      System.out.println(missatge);
10   }
11 }
```

Cal que tingueu present que us trobeu davant una excepció, incorporada pels creadors del Java per facilitar la vostra tasca com a programadors a l'hora de crear missatges de text.

2.1.5 "Arrays" de cadenes de text

De la mateixa manera que es poden declarar i usar *arrays* de tipus primitius, també es poden declarar i usar *arrays* d'objectes, com les cadenes de text. La sintaxi en aquest aspecte no varia en absolut, si bé heu de tenir en compte que la part de la instrucció vinculada a la definició del tipus ara indica el nom de la classe. Per exemple, per declarar un *array* amb capacitat per a 5 cadenes de text es faria de la manera següent, segons si s'usa el mecanisme amb valors concrets o per defecte:

```
1  //Inicialització a valor concret.
2  //Un literal d'una cadena de text es representa amb text entre cometes dobles.
3  String[] arrayCadenaConcret= {"Valor1", "Valor2", "Valor3", "Valor4", "Valor5
       "};
4  //Inicialització a valor per defecte.
5  String[] arrayCadenaDefecte = new String[5];
```

El cas de la inicialització a valors per defecte té una particularitat, i és que les posicions prenen un valor especial que indica que la posició està "buida", anomenat `null`. Alerta, ja que aquest valor és un cas especial i no és equivalent a una cadena buida o sense text (representada amb el literal ""). Mentre una posició és "buida", es compleix que:

- Si es mostra per pantalla, apareix el text: "null".
- Qualsevol invocació d'un mètode sobre seu dóna un error.

Per accedir a una posició d'un *array* d'objectes, la sintaxi és igual que per als tipus primitius, usant un índex entre claus (`[i]`). Tot i així, els objectes tenen la particularitat de permetre la invocació de mètodes. En aquest aspecte, una posició es comporta com una variable qualsevol. Això es pot veure en l'exemple següent, especialment en el segon bucle, en què s'invoca el mètode `length()` directament sobre la cadena de text que hi ha a cada posició de l'*array*. Compileu-lo i executeu-lo per esbrinar què fa:

```
1  public class ArrayString {
2    public static void main(String[] args) {
3      String[] text = {"Hi", "havia", "una", "vegada..."};
4      System.out.println("El text de l'array és:");
5      for (int i = 0; i < text.length; i++) {
6        System.out.println("Posició " + i + ": " + text[i]);
7      }
8      System.out.println("Les seves mides són:");
9      for (int i = 0; i < text.length; i++) {
10       System.out.println("Posició " + i + ": " + text[i].length());
11     }
12   }
13 }
```

2.2 Entrada de cadenes de text

De la mateixa manera que pot ser necessari introduir dades de tipus primitius en el programa, de manera que s'hi pugui treballar, també pot ser necessària l'entrada de cadenes de text. Per al cas de l'entrada de cadenes de text, però, teniu dues possibilitats.

D'una banda, es pot usar el teclat tal com heu fet fins ara. De fet, aquesta és la finalitat més natural per a la qual s'empra el teclat, ja que es tracta d'un perifèric especialment dissenyat per escriure text. Per fer-ho cal que useu les instruccions de lectura que ja heu après anteriorment, però de manera que sigui possible llegir tant paraules individuals com frases de text completes.

D'una altra banda, hi ha una possibilitat nova: l'entrada mitjançant arguments en el mètode principal. Sempre que s'executa un programa al vostre ordinador és possible incloure directament un seguit d'arguments addicionals en forma de text. Si bé en els sistemes operatius moderns, basats en entorns gràfics de finestres, aquesta circumstància queda oculta, aquesta opció està present. Java permet

esbrinar si en executar el vostre programa s'han inclòs aquests arguments i, si és el cas, quins són.

2.2.1 Lectura des de teclat

El procés de lectura de cadenes de text des de teclat és molt semblant a la lectura de tipus primitius. Només cal saber quines són les instruccions adients per obtenir cadenes de text des de teclat, en aquest cas en forma d'objectes de la classe `String`. Abans de veure quines són aquestes instruccions, però, val la pena fer una petita revisió dels elements que intervenen en el procés de lectura de dades per teclat aprofitant que s'han introduït els conceptes de classe, objecte i mètode, sota una nova perspectiva.

Revisió de la lectura per teclat

Torneu a fer una ullada a un exemple simple de lectura de dades per teclat. Per exemple, un programa que llegeix un valor enter i el mostra per pantalla, comprovant abans que el tipus del valor escrit sigui el correcte.

```
1  import java.util.Scanner;
2  //Llegeix un enter, comprovant que sigui correcte.
3  public class LecturaEnter {
4    public static void main (String[] args) {
5      Scanner lector = new Scanner(System.in);
6      int valor = 0;
7      System.out.print("Escriu un valor enter: ");
8      if (lector.hasNextInt()) {
9        valor = lector.nextInt();
10       System.out.println("El valor era " + valor + ".");
11     } else {
12       lector.next();
13       System.out.print("El valor no era enter.");
14     }
15     lector.nextLine();
16   }
17 }
```

Aquest codi té les particularitats següents:

- `Scanner` és una classe. Aquesta ofereix mètodes per llegir dades des del teclat.
- `lector` és una variable en què s'emmagatzema un objecte pertanyent a aquesta classe.
- `hasNextInt()`, `nextInt()` i `next()` són mètodes, oferts per la classe `Scanner`. De fet, totes les instruccions `next...` i `hasNext...` vinculades a la lectura de dades pel teclat són mètodes d'aquesta classe. En invocar-los, provoquen que es faci una lectura des del teclat i avaluen la dada llegida. Segons el tipus de dada que es vol llegir, cal usar un mètode o un altre.

L'únic punt que encara és una incògnita ara mateix és com funciona la inicialització d'una variable de la classe `Scanner`, de manera que contingui un objecte. No és necessari que l'entengueu, simplement n'hi ha prou de saber que per a aquesta classe es fa així. Ara bé, és important tenir en compte que sense una correcta inicialització de la variable `lector` amb un objecte la lectura de dades no serà correcta.

Lectura de paraules individuals

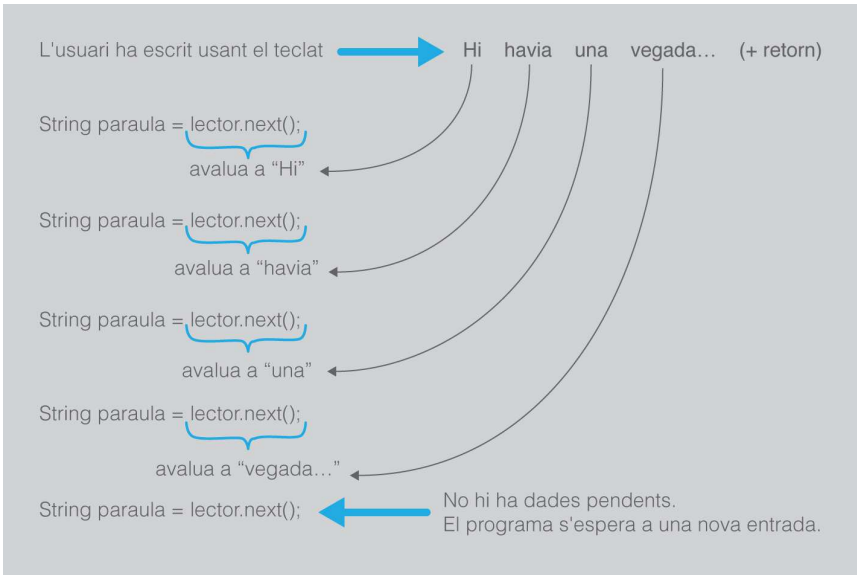
Igual que fins ara s'han llegit tipus primitius individuals entrats per teclat, també és possible llegir paraules.

> El mètode de la classe `Scanner` vinculat a la lectura d'una cadena de text composta d'una única paraula és `next()`.

El seu comportament és exactament el que heu vist fins al moment per llegir tipus primitius, amb l'única diferència que la invocació d'aquest mètode sobre un objecte de la classe `Scanner` avalua una cadena de text. Recordeu que si en una mateixa línia escriviu més d'una paraula, successives invocacions a aquest mètode no bloquejaran el programa, sinó que aniran avaluant les successives dades pendents de llegir. Aquest comportament queda esquematitzat a la figura 2.2.

Cal tenir en compte que, en aquest context, quan es parla de *paraula*, en realitat s'està referint a qualsevol combinació de lletres o nombres que no conté cap espai enmig. Per tant, aquest mètode pot llegir els textos individuals "Hola,", "món!", "1024", "3.1426", "base10", etc. Ara bé, el resultat d'una lectura usant aquest mètode sempre avalua una cadena de text, mai un tipus numèric. Fins i tot quan el valor llegit és un nombre.

Figura 2.2. Lectura de seqüències de paraules



Un fet remarcable en el cas de l'entrada de cadenes de text des de teclat és que no cal fer cap comprovació de tipus prèvia a la lectura (mètodes `hasNext...`) , ja que tota dada escrita mitjançant el teclat sempre és pot interpretar com una cadena de text. Mai no es pot produir una errada en aquest sentit.

La millor manera de veure'n el comportament és amb un exemple. Compileu i executeu el programa següent, que llegeix fins a 10 paraules i les va mostrant per pantalla. Proveu d'escriure més d'una paraula a cada línia (una frase sencera) i observeu com les llegeix totes una per una abans d'esperar novament una entrada de dades. Fixeu-vos que les paraules queden discriminades en la lectura, en estar separades amb espais.

```
1  import java.util.Scanner;
2  //Llegeix un cert nombre de paraules (en aquest cas, 10).
3  public class LecturaParaules {
4    public static final int NUM_PARAULES = 10;
5    public static void main (String[] args) {
6      Scanner lector = new Scanner(System.in);
7      System.out.println("Escriu " + NUM_PARAULES + " paraules separades per
           espais.");
8      System.out.println("Les pots escriure en línies de text diferent, si vols
           .");
9      for (int i = 0; i < NUM_PARAULES; i++) {
10       //Es van llegint paraules una per una.
11       //Recordar el comportament lectura de seqüències de dades pel teclat.
12       String paraula = lector.next();
13       System.out.println("Paraula " + i + ": Has escrit \"" + paraula + "\".");
14     }
15     //Es llegeix la resta de la línia i s'ignora el contingut.
16     lector.nextLine();
17   }
18 }
```

Lectura de caràcters individuals

Hi ha casos en què només es vol llegir un caràcter, per tal de simplificar l'entrada de dades. Per exemple, com a sistema abreujat per escollir opcions d'un menú, o en un programa en què es donin diferents opcions etiquetades amb lletres individuals (com podria ser resoldre una pregunta de tipus test). En un cas com aquest, i atès que la classe `Scanner` no ofereix cap mètode per llegir caràcters individuals, el que cal fer és llegir l'entrada com una cadena de text, comprovar que aquesta només es compon d'un únic caràcter i extreure'l.

- Per llegir una cadena de text hi ha el mètode `next()`, com tot just s'ha vist.
- Per establir la llargària d'una cadena de text es disposa del mètode `length()`. En aquest cas, cal veure si és 1.
- Per extreure un caràcter individual, tenim el mètode `charAt(numPosició)`. En aquest cas, la posició per consultar és la 0.

En el programa següent d'exemple cal escollir la resposta correcta entre quatre opcions etiquetades amb una lletra. Per simplificar el codi, només es permet un sol intent. Executeu-lo i observeu atentament cadascuna de les passes tot just descrites per veure si s'ha llegit realment una única lletra i extreure-la.

```java
import java.util.Scanner;
//Mostra una pregunta tipus test i mira si s'endevina.
public class LecturaCaracter{
  public static final char RESPOSTA_CORRECTA = 'b';
  public static void main (String[] args) {
    Scanner lector = new Scanner(System.in);
    System.out.println("Endevina la pregunta.");
    System.out.println("Quin dels següents no és un tipus primitiu?");
    System.out.println("a) Enter");
    System.out.println("b) Scanner");
    System.out.println("c) Caràcter");
    System.out.println("d) Booleà");
    System.out.print("La teva resposta és l'opció: ");
    //Es llegeix la cadena de text.
    String paraula = lector.next();
    //És una paraula d'un únic caràcter?
    if (paraula.length() == 1) {
      //S'extreu el caràcter de la cadena de text.
      char caracter = paraula.charAt(0);
      //És un caràcter vàlid? (a, b, c o d)
      if ((caracter >= 'a')&&(caracter <= 'd')) {
        //La resposta final és correcta?
        if (caracter == RESPOSTA_CORRECTA) {
          System.out.println("Efectivament, la resposta era '" +
              RESPOSTA_CORRECTA + "'.");
        } else {
          System.out.println("La resposta '" + caracter + "' és incorrecta.");
        }
      } else {
        System.out.println("'" + caracter + "' és una opció incorrecta.");
      }
    } else {
      //No ho era.
      System.out.println("\"" + paraula + "\" no és un caràcter individual.");
    }
    lector.nextLine();
  }
}
```

Repte 2: modifiqueu el programa anterior per permetre fins a tres intents de resposta en lloc d'un.

Lectura de frases senceres

Fins al moment, només ha estat possible llegir elements individuals, un per un i separats per espais, dins d'una seqüència entrada per teclat. Ara bé, sovint és útil llegir frases senceres, compostes per un conjunt de paraules separades per espais. Per exemple, entrar un nom i cognoms, o una adreça postal. En aquest cas, voldríem desar tota la cadena de text dins d'una única variable.

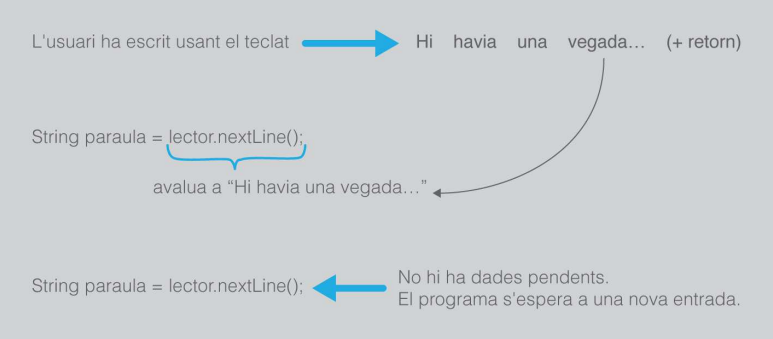
El mètode de la classe `Scanner` vinculat a la lectura d'una cadena de text en forma de frase en què hi ha diverses paraules separades per espais és `nextLine()`.

Fins al moment, aquest mètode ha estat usat per descartar tota la llista d'elements pendents de llegir del teclat (en el cas que s'hagi escrit més d'un element en una única línia). En realitat, aquest mètode el que fa és llegir tots els elements pendents de llegir a la línia actual i avaluar la cadena de text que els conté tots. Per tant,

serveix per llegir conjunts d'elements de cop, independentment dels espais en blanc que els separen.

A efectes pràctics amb vista a la lectura de cadenes de text, si mai no es combina l'ús de `nextLine()` amb altres mètodes de lectura d'elements individuals (`next()`, `nextInt()`, etc.) és possible usar-lo per llegir frases senceres directament. Cada cop que s'invoca, el programa es bloqueja i espera que l'usuari entri una frase sencera i pitgi la tecla de retorn. Llavors, avalua tota la frase escrita (totes les paraules amb espais inclosos). En aquest cas, i al contrari que amb la resta de mètodes de lectura, no és possible escriure més d'un element dins una mateixa línia de text, ja que llegeix tota la línia al complet des de la darrera lectura fins a trobar un salt de línia. Aquest comportament s'esquematitza a la figura 2.3. Contrasteu-lo amb el mostrat a la figura 2.2.

Figura 2.3. Lectura de frases



El programa d'exemple següent mostra com es llegeix una línia de text completa escrita pel teclat, de manera molt semblant a l'exemple anterior de llegir paraules individuals. Tot i així, fixeu-vos que hi ha algunes diferències en el codi i en el comportament. Concretament, mai no es donarà el cas que una lectura pel teclat no ocasioni el bloqueig del programa a l'espera que escriviu quelcom. Absolutament sempre s'aturarà a esperar que escriviu un text i pitgeu la tecla de retorn. Proveu el programa per comprovar que és així.

```
1  import java.util.Scanner;
2  //Llegeix frases escrites pel teclat.
3  public class LecturaFrase {
4    public static final int NUM_FRASES = 4;
5    public static void main (String[] args) {
6      Scanner lector = new Scanner(System.in);
7      System.out.println("Escriu " + NUM_FRASES + " frases.");
8      System.out.println("Per acabar una frase, pitja la tecla de retorn.");
9      for (int i = 0; i < NUM_FRASES; i++) {
10       //Es van llegint frases una per una.
11       String frase = lector.nextLine();
12       System.out.println("Frase " + i + ": Has escrit \"" + frase + "\".");
13     }
14     //Ara no cal llegir la resta de cap línia, ja que sempre es llegeixen
15     //línies senceres...
16   }
17 }
```

2.2.2 Argument del mètode principal

Aquesta opció permet establir una entrada de dades al programa sense haver d'interrompre'n l'execució per haver de preguntar quelcom a l'usuari i esperar una entrada per teclat.

Una pregunta que us podeu plantejar és quina és la utilitat d'aquest sistema quan ja hi ha l'entrada pel teclat. Un exemple de situació en què això és molt útil és en programes que es comportaran de manera diferent segons una entrada de dades, però en els quals no es vol, o no es pot, esperar que l'usuari estigui pendent. Per exemple, suposeu un programa que voleu que s'executi cada nit per fer còpies de seguretat d'un seguit de fitxers. D'una banda, no té sentit que el programa estigui sempre en marxa comprovant si ha arribat l'hora de fer la còpia. És més lògic programar-ne l'execució periòdica amb alguna eina oferta pel sistema operatiu. Ara bé, és evident que no és factible que algú estigui pendent del teclat cada vegada que s'executa, però perquè funcioni correctament ha de disposar de certa informació que tampoc no pot estar escrita dins el codi font, ja que el valor dependrà de cada usuari que l'utilitzi: la llista de directoris que es vol copiar, on es desa la còpia, etc. Per tant, és útil disposar d'una manera simple d'entrar dades en un programa sense que hagi de ser sempre pel teclat.

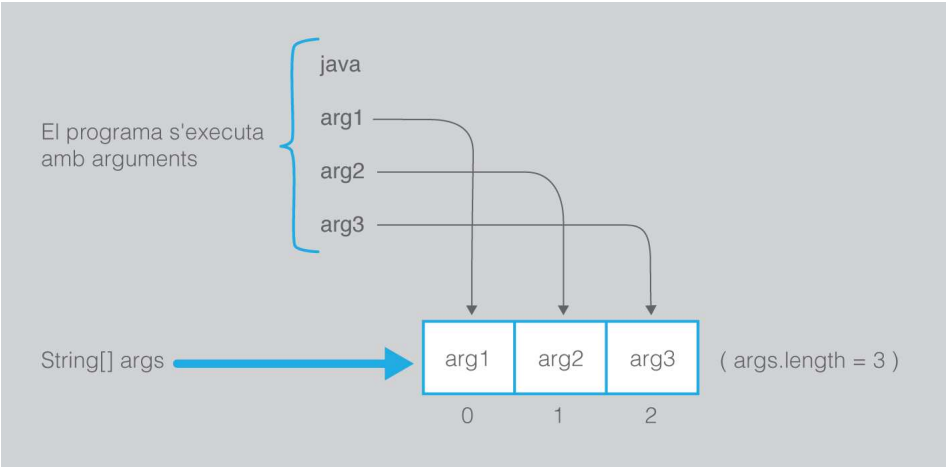
Exemples d'eines que permeten l'execució diferida de programes són les tasques programades en el Windows o la instrucció `cron` de Unix.

Normalment, la llista d'arguments d'un programa s'introdueix com un conjunt de paraules a continuació del nom del fitxer executable, separades per espais.

La llista de paraules que conformen els arguments d'un programa en Java es pot obtenir directament a partir de la variable especial que es defineix en la declaració del mètode principal: `String[] args`. Concretament, pren la forma d'un *array* de cadenes de text, en què aquestes ocupen les posicions en el mateix ordre que s'han escrit els arguments en executar el programa, tal com mostra la figura 2.4.

Disposeu d'un annex en què us expliquem com cal incloure arguments en executar un programa mitjançant l'IDE que usareu al llarg del curs.

Figura 2.4. Assignació d'arguments d'entrada en l'execució del programa "java" a la variable "args"



Executar un programa amb arguments

Normalment, els exemples bàsics de programes executats amb arguments són els vinculats a la línia d'ordres. Entre aquests, es pot trobar l'intèrpret del Java mateix, l'executable `java` (en sistemes Unix) o `java.exe` (en sistemes Windows). Si obriu una finestra de línia d'ordres i hi escriviu `java -help`, apareix una llista dels arguments possibles per executar-lo sense la necessitat d'un IDE: els fitxers amb *bytecode* Java que ha d'executar, si cal mostrar-ne la versió, si ha de mostrar informació de depuració addicional, etc. De fet, "-help" és un argument en si mateix.

El vostre IDE automatitza l'execució i el pas d'arguments a aquest programa sense la necessitat d'haver-vos de preguntar res. Partint dels fitxers que heu generat, ell sol genera la llista d'arguments que cal usar.

Per saber amb quants arguments s'ha executat el programa cal usar l'atribut `length` associat als *arrays*. Si no s'ha passat cap argument, el seu valor serà 0. Aneu amb compte amb aquesta situació, ja que llavors qualsevol accés a una posició de l'*array* serà sempre incorrecte.

A mode d'exemple, proveu el programa següent, que enumera els arguments amb els quals s'ha executat. A efectes pràctics, es tracta d'un recorregut sobre un *array*, si bé els elements emmagatzemats a les seves posicions ara són cadenes de text en lloc de valor numèrics.

```
1  //Programa que escriu per pantalla els arguments d'un programa.
2  public class LecturaArguments {
3    //Els arguments estan a la variable "args".
4    public static void main(String[] args) {
5      //Primer cal mirar si n'hi ha algun.
6      if (args.length > 0) {
7        //N'hi ha. Es mostren per pantalla ordenadament.
8        for (int i = 0; i < args.length; i++) {
9          System.out.println("Argument " + i + ": " + args[i]);
10       }
11     } else {
12       //No n'hi ha cap.
13       System.out.println("No hi ha cap argument.");
14     }
15   }
16 }
```

2.3 Manipulació de cadenes de text

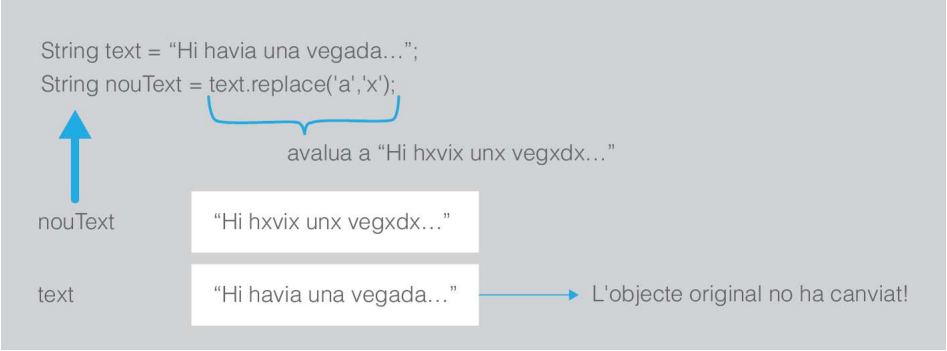
Tot i que una cadena de text organitza els caràcters que emmagatzema d'una manera molt semblant a com ho faria un *array*, en forma de seqüència ordenada en què cada element ocupa una posició identificada per un índex, hi ha dues diferències destacades en el seu comportament i que tenen gran incidència a l'hora de manipular-les.

D'una banda, la classe `String` ofereix un ventall molt ampli de mètodes amb els quals es poden manipular, de manera que en la immensa majoria de casos no caldrà fer un tractament individual de cada posició (per exemple, mitjançant estructures de repetició). Normalment, tots els esquemes d'ús i manipulació de cadenes de text es basen simplement en la invocació del mètode adient.

D'una altra banda, una altra particularitat que les diferencia dels *arrays* és que són immutables. La invocació d'un mètode pot consultar les dades dins d'una cadena de text, o pot generar una nova cadena de text amb certes modificacions respecte de l'original. Però mai no es veurà modificat el valor inicial de la cadena de text original. Si volem, aquesta nova cadena pot ser assignada a la variable original, de manera que es reemplaça el valor inicial pel nou.

La figura 2.5 mostra aquest comportament per la invocació del mètode `replace(caracterNou, caracterVell)`, que reemplaça totes les aparicions d'un caràcter (`caràcterVell`) per un altre de diferent (`caràcterNou`).

Figura 2.5. Els mètodes invocats sobre cadenes de text no en modifiquen el valor, en creen de noves modificades



2.3.1 Cerca

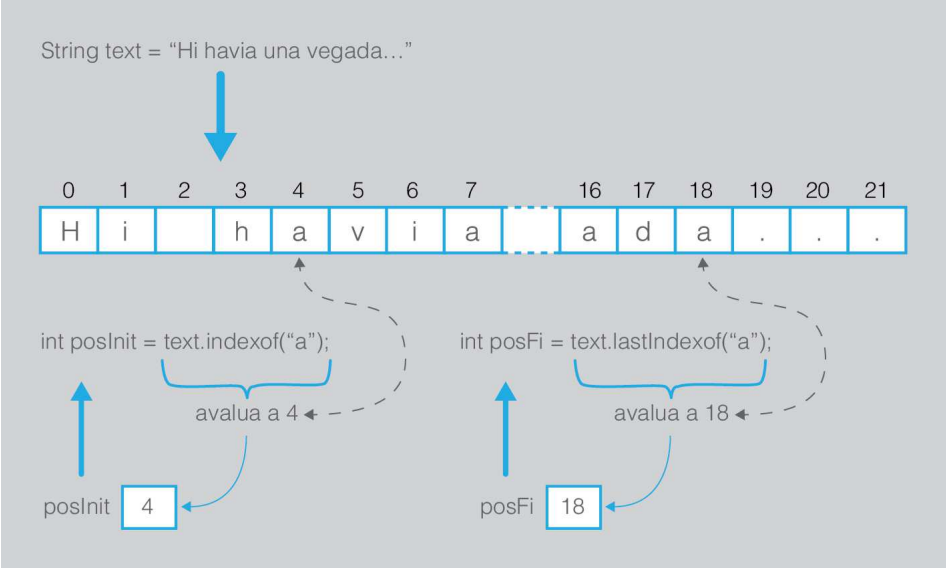
Normalment, el procés de cerca d'un valor concret dins d'un *array* s'hauria de fer amb una estructura de repetició que anés comprovant si aquest valor existeix en alguna de les posicions, fins a trobar-lo o arribar al final de l'*array*. Això es podria replicar amb les cadenes de text usant el mètode `charAt`. Per sort, la classe `String` ofereix dos mètodes que estalvien aquesta tasca, ja que avaluen directament la posició on es troba un caràcter o una subcadena de text concreta.

- `indexOf(textACercar)` avalua la primera posició dins de la cadena de text on es troba el text per cercar.
- `lastIndexOf(textACercar)` avalua la darrera posició dins de la cadena de text on es troba el text per cercar.

En tots dos casos, `caràcterACercar` sempre ha de ser una cadena de text (ja sigui mitjançant un literal o una variable) i si el text cercat no existeix, avaluen a -1. Per tant, no solament és possible saber si existeix el valor cercat, sinó també establir directament posicions on es pot trobar.

La figura 2.6 mostra el comportament dels dos mètodes si se cerca el caràcter `'a'` a la frase "Hi havia una vegada...".

Figura 2.6. Comportament dels mètodes "indexOf" i "lastIndexOf"



El codi següent mostra un exemple de cerca d'un caràcter dins d'un text qualsevol (compost per diverses paraules). Compileu-lo, proveu-lo i estudieu-ne el comportament.

```
1  import java.util.Scanner;
2
3  //Cerca un caràcter concret dins d'una cadena de text qualsevol.
4  public class CercaCaracter {
5
6    public static void main(String[] args) {
7
8      Scanner lector = new Scanner(System.in);
9
10     System.out.println("Escriu una línia de text qualsevol i pitja retorn:");
11     String text = lector.nextLine();
12
13     System.out.println("Quin caràcter vols cercar? ");
14     String charText = lector.next();
15     lector.nextLine();
16
17     char charCerca = charText.charAt(0);
18
19     int posInit = text.indexOf(charCerca);
20     int posFi = text.lastIndexOf(charCerca);
21     if (posInit > -1){
22       System.out.println("Les aparicions del caràcter '" + charCerca + "' són
             :");
23       System.out.println("Primer cop - " + posInit);
24       System.out.println("Darrer cop - " + posFi);
25     } else {
26       System.out.println("Aquest caràcter no es troba al text.");
27     }
28
29   }
30
31 }
```

Repte 3: feu un programa que, en un seguit de cadenes de text passades com a arguments al mètode principal, compti quantes contenen el caràcter 'a'.

2.3.2 Comparació

Una tasca que es fa molt sovint amb cadenes de text és veure si un text entrat per l'usuari correspon a una opció concreta entre les disponibles. Això és especialment freqüent en el cas d'entrada de dades via arguments del mètode principal, ja que en molts programes aquests arguments indiquen opcions per a l'execució.

Malauradament, els objectes no accepten cap mena d'operació entre ells, i això inclou aquelles vinculades als operadors relacionals. Per tant, no es poden comparar directament tampoc, tal com es faria entre valors de tipus primitius. També cal usar algun mètode disponible a la classe a la qual pertanyen. Si la seva classe no ofereix cap mètode que us ajudi a comparar, llavors és impossible comparar-los.

La classe `String` ofereix diferent mètodes per comparar cadenes de text.

- `equals(textPerComparar)` avalua un valor booleà (`true/false`) segons si la cadena de text continguda a la variable en què s'invoca i el text usat com a paràmetre són idèntics o no. El text usat com a paràmetre pot ser tant un literal, ''...'', com una altra variable en què hi hagi emmagatzemada una altra cadena de text.

  Com a exemple, es pot comprovar si els arguments passats a un programa corresponen a alguna de les opcions possibles. En el programa següent, és possible detectar si els arguments corresponen a algun dels acceptats: `-version`, `-help` i `-server`. Per a cada cas, es fa una opció diferent. Si hi ha algun argument que no és cap d'aquests tres, el programa informa que hi ha un error en els arguments. Proveu que és així.

equalsIgnoreCase(text)
Aquest altre mètode fa la mateixa funció, però ignorant majúscules i minúscules.

```
1  //Comprova si els arguments es corresponen amb un conjunt preestablert.
2  public class ComprovaArguments {
3    public static void main (String args[]) {
4      //No oblideu mai comprovar si l'array conté arguments!
5      if (args.length > 0) {
6        //Es va recorrent l'array i es mira quin valor hi ha.
7        for (int i = 0; i < args.length; i++) {
8          if (args[i].equals("-version")) {
9            System.out.println("S'ha usat l'argument \"-version\"");
10         } else if (args[i].equals("-help")) {
11           System.out.println("S'ha usat l'argument \"-help\"");
12         } else if (args[i].equals("-server")) {
13           System.out.println("S'ha usat l'argument \"-server\"");
14         } else {
15           System.out.println("L'argument \"" + args[i] + "\" no és vàlid!");
16         }
17       }
18     }  else {
19       System.out.println("No hi havia cap argument.");
20     }
21   }
22 }
```

De vegades pot ser útil comparar dues cadenes de text des del punt de vista de l'ordre alfabètic, és a dir, de quina seria la seva posició relativa en un diccionari.

Aquest tipus de comparació seria equivalent als operadors relacionals “major que” (>) i “menor que” (<).

compareToIgnoreCase(text)

Aquest mètode és l’equivalent ignorant majúscules i minúscules.

- `compareTo(textPerComparar)` es comporta de manera semblant a `equals`, però en aquest cas avalua a 0, qualsevol valor positiu o qualsevol valor negatiu depenent de si la cadena emmagatzemada en la variable en què s’invoca el mètode és alfabèticament igual, major o menor (respectivament) que la usada com a paràmetre.

  La millor manera de veure-ho és amb un exemple. En el programa següent cal endevinar un text secret partint del fet que, per a cada resposta, el programa us diu com a pista quina és la posició relativa de la paraula secreta segons el seu ordre alfabètic. Reflexioneu sobre quines diferències hi ha entre aquest codi, que tracta cadenes de text, i un programa que fes exactament el mateix amb valors de tipus enter (en aquest cas, per ordre numèric).

```
1  import java.util.Scanner;
2  //Joc d'endevinar una paraula, donant pistes del seu ordre alfabètic.
3  public class EndevinaParaula {
4    //La paraula per endevinar és "objecte".
5    public static final String PARAULA_SECRETA = "java";
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 de la paraula del diccionari.");
10     boolean haEncertat = false;
11     while (!haEncertat) {
12       System.out.print("Quina paraula creus que és? ");
13       String paraulaUsuari = lector.next();
14       lector.nextLine();
15       int posicio = paraulaUsuari.compareTo(PARAULA_SECRETA);
16       if (posicio < 0) {
17         //La paraula de l'usuari és anterior a la secreta.
18         System.out.println("Has fallat! La paraula va després...");
19       } else if  (posicio > 0) {
20         //La paraula de l'usuari és posterior a la secreta
21         System.out.println("Has fallat! La paraula va abans...");
22       } else {
23         //Si val 0, és que s'ha encertat.
24         haEncertat = true;
25       }
26     }
27     System.out.println("Enhorabona. Has encertat!");
28   }
29 }
```

Repte 4: usant el mètode `compareTo`, ordeneu un *array* de cadenes de text. Per facilitar l’entrada de les dades, aquest *array* pot estar determinat directament a partir d’arguments en l’execució del programa (la variable `args`).

2.3.3 Creació de subcadenes

El mètode `charAt` permet l’extracció de dades d’una cadena de text, però només de caràcters individuals. Tot i així, l’aplicabilitat d’aquest mecanisme és limitada.

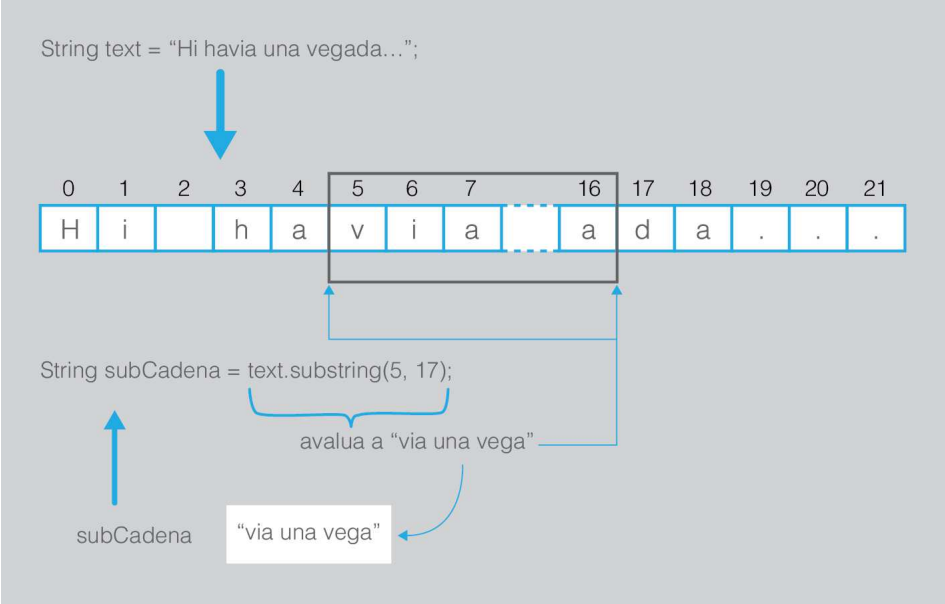
Molt sovint us trobareu que, en realitat, us resultaria més còmode poder extreure cadenes de text completes contingudes dins d'una altra cadena. Per exemple, parts d'una frase o paraules individuals. La classe `String` ofereix dos mètodes que permeten fer això.

- `substring(posicióInicial, posicióFinal)` avalua una nova cadena de text composta pels caràcters que van des de `posicióInicial` fins a `posicióFinal` - 1.

  Heu d'anar amb compte en usar aquest mètode, ja que els valors han d'estar dins del rang permès. Si algun supera el valor de les posicions que ocupen els caràcters dins de la cadena de text, es produirà un error.

  La figura 2.7 mostra un esquema del comportament del mètode `substring` si s'extreu una subcadena de la frase "Hi havia una vegada...".

Figura 2.7. Comportament del mètode "substring"



Com a exemple del funcionament de la invocació d'aquest mètode, tot seguit trobareu el codi d'un programa que extreu la primera i la darrera paraula d'una frase i les mostra per pantalla. Això és equivalent a mostrar una subcadena de text entre el primer i el darrer espai en blanc. Si hi ha menys de tres paraules, llavors no mostra res.

```
1  import java.util.Scanner;
2  //Un programa que extreu tot el text d'una frase, excepte la primera i la
       darrera paraula.
3  public class ExtreureParaules {
4    public static void main (String[] args) {
5      Scanner lector = new Scanner(System.in);
6      System.out.println("Escriu una frase de text i pitja retorn:");
7      String text = lector.nextLine();
8      //Cerca el primer i el darrer espai en blanc.
9      int iniciSubcadena = text.indexOf(' ');
10     int fiSubcadena = text.lastIndexOf(' ');
11     System.out.println("El text sense la primera i la darrera paraula és:");
12     if  (iniciSubcadena == fiSubcadena) {
13       //O bé no hi ha espais (els dos mètode avaluen a -1).
```

```
14        //O bé només hi ha dues paraules (els dos mètodes avaluen la mateixa
              posició).
15        //No es mostra res.
16        System.out.println("(No hi ha res per escriure...)");
17      } else {
18        //Es mostra la cadena intermèdia.
19        //La segona paraula comença una posició després del primer espai en blanc
              .
20        //La darrera paraula comença una posició després del darrer espai en
              blanc.
21        String textFinal = text.substring(iniciSubcadena + 1, fiSubcadena);
22        System.out.println(textFinal);
23      }
24    }
25  }
```

Repte 5: feu un programa que mostri per pantalla cadascuna de les paraules individuals d'una frase en línies diferents. Per fer-ho, abans haureu d'anar esbrinant les posicions on hi ha espais en blanc per poder delimitar on comença i acaba cada paraula.

Per obtenir les paraules d'una frase, no cal cercar on hi ha espais i anar extraient el text entre aquests. Hi ha un altre mètode que facilita molt més la feina.

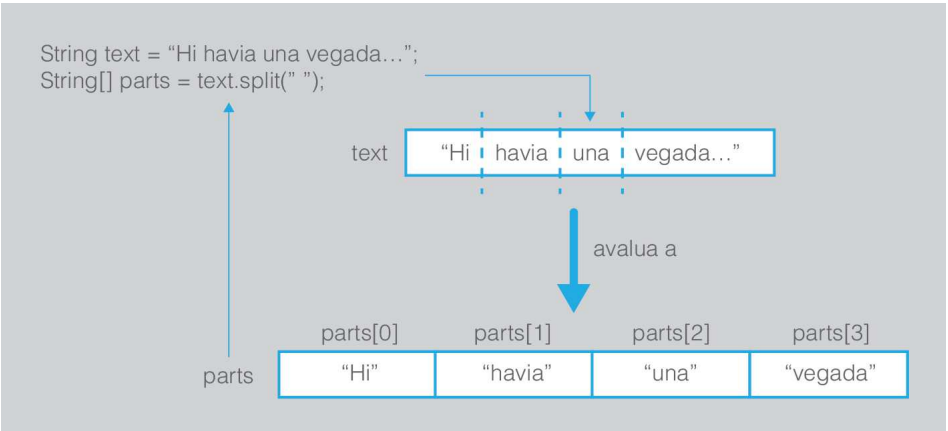
Recordeu que el contingut de la cadena original **mai** no es modifica.

- `split(textSeparació)` avalua un *array* de cadenes de text, en què en cada posició hi ha el resultat de fer el següent. S'agafa la cadena de text original i se separa en trossos usant com a divisor `textSeparació`. Cada tros, una nova cadena de text, s'ubica en una posició de l'*array*. El valor de `textSeparació` ha de ser una cadena de text.

  Per exemple, si com a text de separació d'aquest mètode s'usa un espai en blanc, " ", una frase es dividiria en paraules. El resultat seria un *array* de cadenes de text en què en cada posició hi ha cadascuna de les paraules. Ara bé, el separador pot ser qualsevol text que vulgueu (comes, punts, combinacions de lletres, etc.).

  La figura 2.8 mostra un esquema del comportament d'aquest mètode si es divideix la frase "Hi havia una vegada–" usant com a text de separació l'espai en blanc, de manera que queda dividida en paraules.

Figura 2.8. Comportament del mètode "split"



Com a exemple, estudieu el codi font següent, que pertany un programa per invertir l'ordre de les paraules d'una frase. Proveu en el vostre entorn de treball que és així.

```
import java.util.Scanner;
//Programa que inverteix l'ordre de les paraules d'una frase.
public class InverteixOrdreParaules {
  public static void main(String[] args) {
    Scanner lector = new Scanner(System.in);
    System.out.println("Escriu una línia de text qualsevol i pitja retorn:");
    String text = lector.nextLine();
    //Dividim la frase en paraules, separades per espais.
    String[] paraules = text.split(" ");
    System.out.println("El text invertit és:");
    //Imprimim les paraules en ordre invers.
    for (int i = paraules.length - 1; i >= 0 ; i--) {
      System.out.print(paraules[i]);
      //El mètode split elimina el text de separació (en aquest cas, els espais
          )
      System.out.print(" ");
    }
    System.out.println();
  }
}
```

Repte 6: feu un programa que generi automàticament acrònims. Partint d'una frase determinada, ha de mostrar per pantalla el text compost per les inicials de cada paraula individual que forma la frase.

2.3.4 Transformacions entre cadenes de text i tipus primitius

En molts casos, una part important dels programes es basa en les operacions fetes sobre dades de tipus primitius. Ara bé, amb vista a la interacció amb l'usuari, ja sigui tan per mostrar-les com per llegir-les, és necessari operar amb cadenes de text. Per tant, un mecanisme útil és el de la transformació de tipus primitius en cadenes de text i viceversa.

Conversió de tipus primitiu a cadena de text

El cas més senzill és la transformació d'una dada de tipus primitiu a un objecte de la classe `String`. Si els valors que es volen representar formen part d'una frase o una cadena de text més llarga (per exemple: "El resultat és ", i el valor), tan sols cal usar l'operació suma, com s'ha explicat a l'apartat 1.1.5. Java s'encarrega de fer la conversió de tipus automàticament si en algun dels operands dins d'una concatenació hi ha alguna dada de tipus primitiu. Per al cas de mostrar directament un valor per pantalla, la instrucció `System.out.println` també transforma directament qualsevol tipus primitiu que se li proporcioni en una cadena de text que serà mostrada per pantalla.

Si només volem convertir un valor directament, sense que aquest hagi d'estar inclòs dins d'una frase o amb l'únic objectiu de mostrar-lo per pantalla posteriorment, llavors es pot usar la instrucció `String.valueOf(valor)`. On posa `valor` es pot usar qualsevol dada corresponent a qualsevol tipus primitiu, ja sigui una

variable, un literal o una expressió. Aquesta instrucció avalua el seu equivalent a cadena de text.

Ateses les capacitats innates de la concatenació i la impressió per pantalla per transformar dades de tipus primitius en cadenes de text automàticament, els casos on resulta aplicable l'ús de la instrucció `String.valueOf(valor)` és limitat. En la immensa majoria dels casos, si voleu convertir un tipus primitiu en cadena de text serà per mostrar-lo per pantalla, ja sigui sol o dins d'una frase. Tot i així, hi pot haver casos en què tractar una dada com a cadena de text en lloc de com a valor numèric pot facilitar la tasca del programador. Per exemple, quan es vol tractar aquest valor com una seqüència de dígits individuals.

Suposeu un programa que mostra els dígits en ordre invers d'un nombre real de certa llargària, amb qualsevol nombre de decimals. Penseu-hi una mica i veureu que, tractant-lo com a valor numèric i usant operacions entre reals, no és un problema la resolució del qual resulti evident. Aquest és un cas en què l'algorisme per resoldre el problema és una mica més senzill si el valor per processar es tracta com una seqüència de caràcters que no pas si es tracta com un nombre. Només caldria fer un recorregut per cadascun dels caràcters individuals (les xifres) usant una estructura de repetició. Per tant, és útil transformar-lo en una cadena de text i processar-lo com a tal.

Proveu en el vostre entorn el codi font, mostrat tot seguit:

```
1  //Mostra les xifres d'un nombre real en ordre invers.
2  public class InverteixOrdre {
3    public static void main (String[] args) {
4      float numero = 3.1415926535f;
5      //El convertim en una cadena de text.
6      String numeroText = String.valueOf(numero);
7      //Recorrem els seus dígits un per un, començant pel final.
8      for (int i = numeroText.length() - 1; i >= 0; i--) {
9        //Imprimim cada dígit individual (inclòs el punt decimal).
10       System.out.print(numeroText.charAt(i));
11     }
12   }
13 }
```

Conversió de cadena de text a tipus primitiu

Tot i que ja sabeu com cal llegir dades dels tipus primitius directament des del teclat, de vegades, partint d'una cadena de text, us interessaria transformar-la a un tipus primitiu (normalment, un valor numèric). Per exemple, aquest pot ser el cas d'un programa no interactiu que obté les dades d'entrada per al pas d'arguments al mètode principal. En aquest cas, sempre es parteix de valors en format cadena de text i, per tant, cal fer una conversió si algun dels arguments és un valor numèric amb el qual cal operar.

Per a aquest cas, Java disposa d'un seguit d'instruccions, una per a cada tipus primitiu, que serveix per fer aquesta conversió. Ara bé, com passa amb les instruccions de lectura de tipus primitius per teclat, cal garantir que la cadena de text per convertir és correcta, ja que en cas contrari el programa donarà un error. Per exemple, no és possible convertir a un valor de tipus enter les cadenes de text

"2,67334", "1a88", "true", etc. Aquestes instruccions s'enumeren a la taula 2.1. On diu `text` cal posar un objecte de tipus `String`, ja sigui mitjançant una variable, un literal o una concatenació de cadenes.

Taula 2.1. Instruccions per convertir cadenes de text en tipus primitius

Instrucció	Tipus de dada convertit
`Byte.parseByte(text)`	`byte`
`Integer.parseInt(text)`	`int`
`Double.parseDouble(text)`	`double`
`Float.parseFloat(text)`	`float`
`Long.parseLong(text)`	`long`
`Short.parseShort(text)`	`short`

Per exemple, suposeu un programa que escriu una cadena de text un cert nombre de vegades. Per fer-ho, es basa en dos arguments en el mètode principal. El primer és la cadena de text per escriure, i el segon el nombre de vegades (un valor enter). Observeu-ne el codi, proveu-lo i executeu-lo en el vostre entorn. Observeu també què passa si el segon argument no és un valor de tipus enter.

```
1  //Mostra una cadena de text N vegades, partint d'arguments d'entrada.
2  public class MostraNVegades {
3    public static void main (String[] args) {
4      //Es mira si hi ha els 2 arguments que corresponen.
5      if (args.length != 2) {
6        System.out.println("El nombre d'arguments no és correcte!");
7      }  else {
8        //Hi ha 2 arguments, el segon és el nombre de vegades.
9        int numVegades =  Integer.parseInt(args[1]);
10       //Fem una estructura de repetició. Cal mostrar el primer argument N
             vegades.
11       for (int i = 0 ; i < numVegades; i++) {
12         System.out.println(args[0]);
13       }
14     }
15   }
16 }
```

Repte 7: feu un programa que calculi la divisió entre dos valors reals, fent que aquests valors siguin entrats com a arguments en el mètode principal.

2.4 Solucions dels reptes proposats

Repte 1:

```
public class InverteixCadena {
  public static void main (String[] args) {
    String holaMon = "Hola, món!";
    //Es recorren les posicions de la cadena de text una per una.
    for (int i = holaMon.length() - 1; i >= 0; i--) {
      System.out.print(holaMon.charAt(i));
    }
    System.out.println();
  }
}
```

Repte 2:

```
import java.util.Scanner;
public class LecturaCaracterTresIntents {
  //El nombre d'intents es declara com una constant.
  public static final int INTENTS = 3;
  public static final char RESPOSTA_CORRECTA = 'b';
  public static void main (String[] args) {
    Scanner lector = new Scanner(System.in);
    System.out.println("Endevina la pregunta.");
    System.out.println("Quin dels següents no és un tipus primitiu?");
    System.out.println("a) Enter");
    System.out.println("b) Scanner");
    System.out.println("c) Caràcter");
    System.out.println("d) Booleà");
    //Cal un comptador d'intents.
    int intents = 0;
    //Cal saber si s'ha encertat (semàfor).
    boolean encertat = false;
    //Es pregunta mentre no s'encerti ni s'esgotin els intents.
    while ((intents < INTENTS)&&(!encertat)) {
      System.out.print("La teva resposta és l'opció: ");
      //Es llegeix la cadena de text.
      String paraula = lector.next();
      //És una paraula d'un únic caràcter?
      if (paraula.length() == 1) {
        //S'extreu el caràcter de la cadena de text.
        char caracter = paraula.charAt(0);
        //És un caràcter vàlid? (a, b, c o d).
        if ((caracter >= 'a')&&(caracter <= 'd')) {
          //La resposta final és correcta?
          if (caracter == RESPOSTA_CORRECTA) {
            System.out.println("Efectivament, la resposta era '" +
                RESPOSTA_CORRECTA + "'.");
            encertat = true;
          } else {
            System.out.println("La resposta '" + caracter + "' és incorrecta.")
                ;
            intents++;
          }
        } else {
          System.out.println("'" + caracter + "' és una opció incorrecta.");
          //Si hi ha aquest error, no s'esgotarà cap intent...
        }
      } else {
        //No ho era.
        System.out.println("\"" + paraula + "\" no és un caràcter individual.")
            ;
        //Si hi ha aquest error, no s'esgotarà cap intent...
      }
```

```
      lector.nextLine();
    }
    if (intents >= INTENTS ) {
      System.out.println("Has esgotat tots els teus intents.");
    }
  }
}
```

Repte 3:

```
public class ComptaArgumentsA {
  public static void  main(String[] args) {
    //Un comptador de paraules amb 'a'.
    int numAs = 0;
    //Cal recórrer l'array d'arguments.
    for (int i = 0; i < args.length; i++) {
      //Es mira si hi ha alguna lletra 'a' a la cadena de text.
      if (-1 != args[i].indexOf("a")) {
        numAs++;
      }
    }
    System.out.println("El nombre d'arguments amb el caràcter 'a' és " + numAs)
        ;
  }
}
```

Repte 4:

```
public class OrdenaArguments {
  public static void main (String[] args) {
    for (int i = 0; i < args.length - 1; i++) {
      for(int j = i + 1; j < args.length; j++) {
        if (args[i].compareTo(args[j]) > 0) {
          String canvi = args[i];
          args[i] = args[j];
          args[j] = canvi;
        }
      }
    }
    //El mostrem per pantalla.
    System.out.println("Les cadenes ordenades són:");
    for (int i = 0; i < args.length;i++) {
      System.out.println(args[i]);
    }
  }
}
```

Repte 5:

```java
import java.util.Scanner;
public class ObtenirParaules {
  public static void main(String[] args) {
    Scanner lector = new Scanner(System.in);
    System.out.println("Escriu una frase de text i pitja retorn:");
    String text = lector.nextLine();
    boolean fi = false;
    //Mitjançant un bucle, s'anirà escurçant el text paraula per paraula.
    do {
      //La primera paraula va de l'inici al primer espai.
      int primerEspai = text.indexOf(" ");
      //No hi ha espais. No hi ha cap paraula més per escriure.
      if (primerEspai == -1) {
        System.out.println(text);
        fi = true;
      } else {
        //Hi ha un espai. Extreure paraula i mostrar-la.
        String paraula = text.substring(0, primerEspai);
        System.out.println(paraula);
        //S'elimina la paraula del text, de manera que s'escurça.
        text = text.substring(primerEspai + 1, text.length());
      }
    } while (!fi);
  }
}
```

Repte 6:

```java
import java.util.Scanner;
public class GeneradorAcronims {
  public static void main(String[] args) {
    Scanner lector = new Scanner(System.in);
    System.out.println("Escriu una línia de text qualsevol i pitja retorn:");
    String text = lector.nextLine();
    //Dividim la frase en paraules, separades per espais.
    String[] paraules = text.split(" ");
    //Recorrem les paraules i escrivim cada inicial.
    for(int i = 0; i < paraules.length; i++) {
      System.out.print(paraules[i].charAt(0));
    }
    System.out.println();
  }
}
```

Repte 7:

```java
public class DividirArgumentsReals {
  public static void main(String[] args) {
    //Està bé comprovar si el nombre d'arguments és correcte.
    if (args.length != 2) {
      System.out.println("El nombre d'arguments és incorrecte.");
    } else {
      //Conversió de tipus.
      double dividend = Double.parseDouble(args[0]);
      double divisor = Double.parseDouble(args[1]);
      double resultat = dividend/divisor;
      System.out.println("El resultat és " + resultat);
    }
  }
}
```