El llenguatge de programació Java

Jordi Brínquez Jiménez

PID_00161659

UOC
Universitat Oberta de Catalunya
www.uoc.edu

Índex

Introducció 5

Objectius 6

1. Java com a llenguatge de programació estructurada 7
1.1. Introducció al Java 7
1.1.1. Característiques del llenguatge 8
1.1.2. Paraules reservades 8
1.1.3. Convencions en el nom de variables i funcions 9
1.1.4. El primer programa en Java 10
1.2. Tipus de dades 10
1.2.1. Nombres 11
1.2.2. Caràcters 12
1.2.3. Tipus enumerats 13
1.2.4. Constants 13
1.2.5. El tipus booleà 14
1.3. Operadors 14
1.3.1. Operador d'assignació 15
1.3.2. Operadors aritmètics 15
1.3.3. Operadors relacionals 16
1.3.4. Operadors lògics 17
1.3.5. Operadors a nivell de bit 17
1.3.6. Operadors equivalents 18
1.3.7. Operador condicional 19
1.3.8. Precedència d'operadors 20
1.4. Matrius i vectors 22
1.5. Blocs d'instruccions 23
1.5.1. Blocs condicionals 23
1.5.2. Blocs iteratius 26
1.5.3. Sentència `break` 28
1.5.4. Sentència `continue` 28
1.6. Funcions 29
1.6.1. Definició de funcions 29
1.6.2. Els paràmetres d'entrada 30
1.6.3. El valor de retorn 30
1.6.4. Un exemple de funció 31
1.6.5. La invocació de funcions 32
1.7. Visibilitat de les variables 32
1.7.1. Variables locals 33
1.7.2. Variables globals 34

2. Java com a llenguatge de programació orientada a objectes 35
2.1. Definició de classes 35
2.1.1. La classe 35
2.1.2. Els atributs d'una classe 36
2.1.3. Mètodes d'una classe 37
2.1.4. Ús de la paraula reservada `this` 39
2.1.5. Mètodes estàtics 40
2.1.6. Sobrecàrrega de mètodes 40
2.2. Les relacions entre classes 41
2.2.1. Cardinalitat 42
2.2.2. Nombre exacte 42
2.2.3. Rang de valors 42
2.2.4. Valors indefinits 43
2.2.5. Navegabilitat 43
2.2.6. Rols 44
2.3. Biblioteca de classes 44
2.3.1. La classe `String` 45
2.3.2. La classe `ArrayList` 47
2.4. Les excepcions 49
2.4.1. Creació d'una excepció 49
2.4.2. Llançament d'excepcions 50
2.4.3. Tractament d'excepcions 50

Resum 52

Activitats 53

Solucionari 54

Glossari 54

Bibliografia 55

Introducció

Aquest mòdul pretén repassar els conceptes bàsics de codificació adquirits en altres assignatures. En concret, farem un repàs de conceptes de programació descendent i prepararem l'estudiant amb vista als nous conceptes que s'exposaran en aquesta assignatura.

Aquest mòdul està organitzat en dos grans blocs:

- Una introducció al llenguatge Java com a llenguatge de programació estructurada.

- La utilització del llenguatge Java com a llenguatge orientat a objectes.

En el primer apartat, "Java com a llenguatge de programació estructurada", es pretén que els estudiants reviseu la sintaxi bàsica de Java (tipus de dades, operadors, blocs condicionals i iteratius, funcions, etc.), i altres conceptes bàsics com la visibilitat de les variables.

En el segon apartat s'introdueixen tots aquells conceptes específics de Java que ens permetran treballar amb el paradigma de la programació orientada a objectes. A part, explicarem conceptes d'interacció amb l'usuari a partir de la pantalla i el teclat a fi i efecte de poder-los aplicar als nous conceptes que s'exposaran en aquesta assignatura, amb la qual cosa podrem passar a crear els nostres primers "programes orientats a objectes".

Posteriorment completarem l'aprenentatge amb la gestió d'errors fent servir les excepcions o el tractament de fitxers.

Objectius

Els objectius d'aquest mòdul són principalment dotar l'estudiant d'uns coneixements bàsics per tal que pugui realitzar les pràctiques de l'assignatura amb certa agilitat.

Se suposaran uns coneixements mínims de programació, ja que no és l'objectiu d'aquest mòdul ensenyar a programar, sinó introduir els conceptes de la programació orientada a objectes a partir d'un llenguatge de programació concret (en aquest cas Java).

De la mateixa manera, tampoc es pretén que sigui un manual del llenguatge de programació Java; per a això existeixen molts manuals (uns en paper i d'altres disponibles al web) que cobreixen gairebé totes les necessitats d'un programador.

1. Java com a llenguatge de programació estructurada

1.1. Introducció al Java

El llenguatge de programació Java va aparèixer a principi dels anys 1990 arran d'un projecte intern de l'empresa Sun Microsystems que tenia com a objectiu permetre al programador treballar de manera més àgil, sense alguns dels problemes que presentaven alguns llenguatges de programació com C/C++. Aquest llenguatge prometia en els seus inicis la possibilitat d'escriure el codi una única vegada i poder-lo executar en moltes plataformes diferents. Inicialment es feia servir com a llenguatge per a escriure petits programes que s'incrustaven a les pàgines web (els *applets*) i de mica en mica ha anat guanyant terreny fins a la programació de grans aplicacions empresarials, així com de pàgines web senceres i dispositius mòbils.

El llenguatge Java es va crear tenint en compte cinc principis:

1) Fer servir la metodologia de l'orientació a objectes.

2) Permetre l'execució del mateix programa directament en diferents sistemes operatius.

3) Permetre la utilització de la xarxa de manera senzilla.

4) Permetre l'execució de codi remot de manera segura.

5) Ser senzill de programar.

Per a assolir el segon objectiu, els enginyers de Sun van crear un compilador que transforma el codi font a un llenguatge que s'anomena *bytecode*. Aquest és un llenguatge intermedi entre el codi font i el llenguatge màquina, que posteriorment una màquina virtual executa en la plataforma corresponent.

Màquina virtual

Una màquina virtual és un programa que és capaç d'executar altres programes i de garantir-ne la seguretat.

Aquesta màquina virtual permet una independència de la plataforma sobre la qual s'executa el codi; per tant, el programador de Java queda alliberat de la tasca de fer compatible el codi en totes les plataformes en què vol que aquest s'executi, ja que és Sun Microsystems la responsable d'aquesta feina. És el concepte de "compilar una vegada i executar-ho a tot arreu"* que Sun va utilitzar en el llançament del llenguatge Java.

* En anglès, *compile once, run everywhere.*

Des del 1996, quan va sortir la versió 1.0, fins a l'actualitat (ja hi ha la versió 6.0), han aparegut diverses versions i per a cada una d'elles diverses subversions que solucionaven possibles errades. Però no només han sorgit versions per al desenvolupament d'aplicacions d'escriptori, sinó que també existeix des de

fa temps una versió reduïda i adaptada, tant del llenguatge com de la màquina virtual, per a executar aplicacions en dispositius mòbils (com telèfons i agendes personals), fotocopiadores o d'altres aparells electrònics.

1.1.1. Característiques del llenguatge

El llenguatge de programació Java pot ser compilat gairebé en qualsevol sistema.

El llenguatge Java es considera un llenguatge d'alt nivell. Java facilita molt les tasques de programació, ja que permet programar sense tenir en compte la plataforma en què executen les aplicacions.

Recordeu

Aquest nivell indica el grau de proximitat del llenguatge amb el maquinari.

Algunes de les característiques que podem esmentar del llenguatge Java són les següents:

- Un nucli del llenguatge simple.
- Està orientat a la programació fent servir el paradigma de l'orientació a objectes.
- Donada la biblioteca d'objectes que es proporciona amb el compilador, es transforma en un llenguatge molt potent.
- Accés i gestió de la memòria senzill gràcies al reciclatge de memòria*.
- Variables locals, globals i de classe.
- Seguretat de tipus.
- Sobrecàrrega d'operadors i mètodes.

*** En anglès, *garbage collection*.**

Aquesta potència que ofereix, juntament amb la facilitat d'aprenentatge, han fet que Java sigui un dels llenguatges de programació més utilitzats avui dia, tant en entorns acadèmics com en entorns empresarials.

Tot i que el llenguatge de programació Java és un llenguatge pensat per al paradigma de la programació orientada a objectes, no el podem classificar com un llenguatge orientat a objectes "pur", atès que els seus tipus bàsics no són objectes (encara que en les noves versions els compiladors els transformen en objectes directament, sense la necessitat de fer conversions explícites).

Vegeu els llenguatges orientats a objectes purs i híbrids en el mòdul "Classes i objectes" d'aquesta assignatura.

1.1.2. Paraules reservades

En un llenguatge de programació, una **paraula reservada** és aquella que no es pot fer servir com a identificador, perquè ja es fa servir dins la gramàtica del propi llenguatge.

Això vol dir que no podem fer servir certes paraules per a donar noms ni a variables ni a funcions. Vegem quines són les paraules reservades del llenguatge Java.

```
abstract   continue   for          new         switch
assert     default    if           package     synchronized
boolean    do         goto         private     this
break      double     implements   protected   throw
byte       else       import       public      throws
case       enum       instanceof   return      transient
catch      extends    int          short       try
char       final      interface    static      void
class      finally    long         strictfp    volatile
const      float      native       super       while
```

Per tant, no podrem fer servir cap d'aquests noms per a identificar variables o funcions dintre del nostre codi. Veurem el significat de la majoria d'aquestes paraules al llarg d'aquest apartat.

1.1.3. Convencions en el nom de variables i funcions

En Java, com en qualsevol llenguatge de programació, cal seguir unes normes per a anomenar les variables i les funcions. En cas que no se segueixin, es produirà un error de compilació.

Aquestes regles són les següents:

- El primer caràcter ha de ser una lletra o el caràcter '_'.
- Els caràcters que vinguin a continuació poden ser tant lletres com nombres i el caràcter '_'.
- Les majúscules i les minúscules es consideren símbols diferents.
- Es poden fer servir caràcters com ara lletres accentuades, la 'ç' o la 'ñ', perquè Java fa servir la tipografia Unicode, però és altament desaconsellat.
- No es poden fer servir espais en blanc.
- El nom no pot ser una paraula reservada.

Enllaç d'interès

Per tal de conèixer tots els caràcters Unicode i les seves implicacions podeu visitar la pàgina:
http://www.unicode.org

Aquestes regles són les que imposa el llenguatge, però per a fer el més entenedora possible la utilització de les variables i funcions és recomanable seguir els criteris següents:

- Escriure els noms en minúscules.
- Fer servir noms descriptius.
- Evitar noms que es puguin confondre amb facilitat.
- Escriure la primera lletra de les paraules "intermèdies" en majúscules.

Vegem uns exemples d'identificadors correctes, incorrectes i no recomanats:

Nom o identificador	És correcte?	Comentari
`vendes`	Sí	
`aux2`	No recomanable	És millor fer servir noms més descriptius.
`1Article`	No	No pot començar per un nombre.
`const`	No	És una paraula reservada en Java.
`vendesGener`	Sí	
`vendesgener`	Sí	

Cal tenir en compte que el llenguatge Java diferencia les majúscules de les minúscules.* Per tant, els dos últims identificadors són diferents i no identifiquen el mateix element. Cal anar amb compte i fer servir uns criteris regulars durant tot el programa per a evitar confusions.

*** Un llenguatge que diferencia majúscules i minúscules es diu que és *case sensitive*.**

1.1.4. El primer programa en Java

Arribats a aquest punt i abans de continuar endavant amb les peculiaritats de Java, potser és convenient veure quin aspecte té un programa senzill per a anar-nos-hi acostumant.

```
public class HelloUOC {
   public static void main(String[] args) {
      System.out.println("Welcome to UOC")
   }
}
```

Observació

Encara que hi ha paraules que no entenem (com ara `public` o `static`), sí que es pot entendre aproximadament el funcionament del programa.

Aquest programa únicament escriu un missatge de benvinguda per pantalla. Com podeu veure, hi ha elements que poden resultar estranys, però n'anirem explicant el significat de mica en mica al llarg d'aquest apartat.

1.2. Tipus de dades

En els nostres programes en Java, al igual que en qualsevol altre llenguatge de programació, ens caldrà emmagatzemar les nostres dades en variables. Els tipus de dades que aquestes poden representar tindran un format lògic, numèric i caràcter (també anomenat *alfanumèric*). Dintre del format numèric podem representar nombres enters i nombres reals. Vegem-ne les característiques i com es poden representar en llenguatge Java.

1.2.1. Nombres

Com ja hem comentat, els tipus numèrics es poden dividir en dos grans grups: nombres enters i nombres en coma flotant. Vegem quins tipus de dades tenim i el rang dels valors que poden emmagatzemar.

Nombres enters

Els nombres enters són els que utilitzem normalment per a comptar objectes, per exemple: *tres pomes*.

En Java tenim diferents tipus de dades per a representar els nombres enters. La principal diferència és el rang numèric que podem representar amb els diferents tipus. En la taula següent els veiem tots:

Tipus	Mida en bytes	Rang	Exemples
`byte`	1	$-2^7 \dots 2^7-1$	`-100000, 345678`
`short`	2	$-2^{15} \dots 2^{15}-1$	`32769, 0xffea`
`int`	4	$-2^{31} \dots 2^{31}-1$	`234, -1500`
`long`	8	$-2^{63} \dots 2^{63}-1$	`-64323, 0xaffaf`

Podem representar els nombres enters fent servir diferents notacions:

- Notació decimal, base 10: `1234`, `925`, `12345678901234`
- Notació octal, base 8 (comencen amb `0`): `02`, `0123`, `0534`
- Notació hexadecimal, base 16 (comencen amb `0x`): `0xa3`, `0xfffa`

El modificador `unsigned` denota que els valors que s'emmagatzemen són únicament positius. Per tant, tenim el doble d'espai, i l'espai reservat als nombres negatius el podem aprofitar per a emmagatzemar nombres més grans.

Per a nombres més grans

Si necessitéssim nombres més grans ja hauríem de fer servir classes com per exemple `BigInteger`, que permeten emmagatzemar nombres de qualsevol mida.

De tots els tipus dels quals disposem per a representar els enters, el tipus `int` serà el que utilitzarem normalment, però cal tenir present que els altres existeixen per si hem de fer algun programa que els requereixi.

Més endavant veurem quins operadors podem fer servir per a treballar amb els nombres enters, però perquè us en feu una idea, seran totes les operacions matemàtiques més comunes (suma, resta, multiplicació, divisió, etc.).

Vegeu els operadors per a treballar amb els nombres enters en el subapartat 1.3 d'aquest mòdul.

Nombres en coma flotant

Aquests nombres, normalment anomenats *decimals*, són aquells que es fan servir per a denotar coses que no poden ser mesurades únicament fent servir unitats, per exemple: *1,5 kg de pomes*, *3,25 €*.

De la mateixa manera que ens passa amb els nombres enters, en Java tenim més d'una manera de representar els nombres reals. En la següent taula tenim un resum dels tipus bàsics disponibles.

Notació de Java

La notació que fa servir Java per a la representació de nombres decimals és la proposada per l'IEEE 754.

Tipus	Mida en bytes	Rang
`float`	4	3.4E–38..3.4E+38 i –3.4E–38..–3.4E+38
`double`	8	1.7E–308..3.4E+308 i –1.7E–308..–1.7E+308

Com podeu veure en aquesta taula, es poden representar nombres molt grans i al mateix temps nombres molt petits amb el mateix tipus de dada. En aquest cas les possibles notacions són dues:

- Notació decimal: `0.1` `12.25` `-0.002341`
- Notació científica: `2.3E+5` `0.123E-32` `4.98E+5.4`

Notació científica

La notació científica serveix per a abreviar en l'escriptura. Per exemple:
$2.3E+5 = 2.3 \cdot 10^5$
$0.123E-32 = 0.123 \cdot 10^{-32}$

De la mateixa manera que amb els nombres enters, més endavant també veurem amb més detall les operacions que podem fer amb els nombres reals.

1.2.2. Caràcters

El **caràcter** és un tipus de dades especial en Java que, tot i representar una lletra o símbol de l'alfabet (recordem que abans hem comentat que Java utilitza Unicode), acostuma a ser tractat internament com un nombre. El nom del tipus de dades és `char`.

Aquesta característica prové del fet que els alfabets en un ordinador no són més que codificacions que fan servir 2 bytes (en comptes del byte que fa servir la codificació ASCII) per a emmagatzemar les lletres. La manera perquè el compilador pugui interpretar aquests caràcters com a tals, i no com a instruccions del nostre programa, és posar-los entre cometes simples.

L'ús d'Unicode

La utilització de la codificació Unicode es deu al fet que no tots els idiomes fan servir els mateixos caràcters, i amb els 256 caràcters que conté la codificació ASCII no n'hi havia prou per a representar qualsevol alfabet (ciríl·lic, xinès, etc.).

Els valors vàlids per a un element de tipus `char` acostumen a ser lletres (majúscules i minúscules), nombres i signes d'ús comú com, per exemple, operadors matemàtics, símbols de puntuació i altres caràcters especials com ara el tabulador ('\t'), per als quals ens caldrà fer ús de combinacions de caràcters perquè no tenen una representació gràfica en la codificació ASCII. Com que Java fa servir la codificació Unicode, es poden representar qualsevol dels caràcters existents.

Les principals operacions que podem dur a terme sobre un element de tipus `char` són les comparacions, però atès que internament s'emmagatzema com un enter de 2 bytes, també es poden fer operacions aritmètiques. Tot i així, no és aconsellable, per no confondre conceptes, i a més a més, moltes vegades proporciona resultats inesperats.

Sabíeu que...

... el tipus caràcter es pot fer servir com a nombre, però tot i així, no s'aconsella per la seva poca capacitat d'emmagatzemament.

1.2.3. Tipus enumerats

Els **tipus enumerats** són un tipus especial de dades que permeten crear tipus de dades propis, sempre que sapiguem quins valors volem que aquest nou tipus accepti en temps de compilació.

Els tipus enumerats es van introduir en la versió 1.5 del llenguatge Java.

Aquests tipus de dades acostumen a fer-se servir per tal d'associar valors constants a noms per fer més entenedor el codi. Un exemple clar de tipus enumerat serien els dies de la setmana o els mesos de l'any.

```
enum days {SUN, MON, TUE, WED, THU, FRI, SAT};

enum months {JAN, FEB, MAR, APR, MAY, JUN, JUL, AUG, SEP,  OCT,
             NOV, DEC};
```

Les enumeracions en Java es tracten com les cadenes de text que es fan servir per a identificar els elements. Se'n pot obtenir la traducció a un valor de tipus enter que ens n'indiquin la posició, però per a nosaltres això resulta completament transparent. Per tant, les podem fer servir com si fossin nous tipus bàsics.

```
days d = SUN;
...

if (d == THU) {
   ...
}
```

Observació

La notació que farem servir aquí no és exactament la proposada pel fet que la majoria d'enumeracions es definiran dintre d'una classe. Reprendrem aquesta qüestió quan veiem les classes.

1.2.4. Constants

Les constants per si mateixes no són cap tipus de dades, però val la pena esmentar-les en aquest apartat.

Una **constant** és únicament un valor que podem assignar o comparar a un tipus de dades determinat, però que mai podrà variar durant l'execució del programa.

Per a la declaració de les constants hem d'utilitzar el mot `final` per tal de denotar que un cop assignat el valor ja no es podrà canviar; i posteriorment el tipus de dada que emmagatzemarem.

Si es coneix el valor de la constant abans de la compilació, s'acostuma a assignar en el moment de la declaració. Si, per contra, el valor depèn d'algun parà-

metre (recordem que un cop assignat no es pot modificar) es pot assignar en qualsevol moment.

```
final int DAYS_OF_THE_WEEK = 7;
final char ENDLINE         = '\n';
```

Els noms de les constants s'acostumen a escriure en lletres majúscules.

1.2.5. El tipus booleà

Els tipus de dades **boolean** és aquell que pot prendre els valors 'cert' o 'fals'.

Aquest tipus de dades s'utilitza en operacions booleanes com les operacions AND i OR lògica, condicions de finalització d'iteracions, etc.

Al contrari que en altres llenguatges, en Java hi ha un tipus de dades que ens ajuda a representar aquest concepte i no s'han d'utilitzar enters per a la seva representació. Els possibles valors que pot tenir són `true` (cert) i `false` (fals).

1.3. Operadors

Els **operadors** són símbols del llenguatge que ens permeten fer operacions amb els elements implicats (també anomenats **operands**). Els operadors poden ser de diversos tipus segons la seva funcionalitat i el nombre d'operands que reben.

Un concepte que cal tenir clar abans de començar a veure quins operadors té el llenguatge Java és el concepte d'*expressió*. Una **expressió** és una combinació de valors, funcions, operadors i altres expressions que, fent servir les regles de precedència dels operadors que les formen, generen un resultat (una altra expressió).

Un altre tema que aprendrem en aquest subapartat és la precedència que tenen els operadors. A mesura que les expressions siguin més complexes, caldrà saber quina és la precedència que té cada operador per tal d'aconseguir els resultats esperats.

Podem dividir els operadors aritmètics en 2 grups segons el nombre d'operands que fan servir. Així tenim:

- per als operadors que fan servir un operand:

```
operator op1
```

- i per als operadors que fan servir dos operands (la resta):

```
op1 operator op2
```

1.3.1. Operador d'assignació

Els **operadors d'assignació** es fan servir per a assignar valors a les variables.

En Java, aquest operador se simbolitza amb un símbol d'igual (=). Les assignacions poden ser entre una variable i un valor o entre dues variables:

```
int var1;
int var2;
var1 = 3;
var2 = var1;
```

En les assignacions entre dues variables cal evitar que es perdi informació en l'assignació o que puguin produir-se desbordaments:

Efectes no desitjats

Es poden donar efectes no desitjats si intentem assignar un valor major que el valor màxim representable en un tipus de dades.

```
short s;
int i = 450000;
s = i; // s cannot handle such a big number
```

1.3.2. Operadors aritmètics

Els **operadors aritmètics** són aquells que ens permeten fer operacions aritmètiques amb nombres, tant enters com reals.

En la següent taula teniu una descripció dels operadors que es poden fer servir i els tipus de dades sobre els quals es poden utilitzar:

Operador	Significat	Tipus de dada	Exemple
-	Canvi de signe	Numèric	-3, -4.2
+	Suma	Numèric	3 + 8, 3.4 + 1.5
-	Resta	Numèric	9 - 4, 2.9 - 5.3
*	Multiplicació	Numèric	3 * 4, 3.0 * 4.1
/	Divisió	Numèric	3/4, 4.2/9.3
%	Mòdul	Enter	250%23

També es poden fer operacions entre diferents tipus de dades (per exemple, podríem sumar un `float` amb un `int`). Ara bé, sempre cal tenir en compte que la divisió entre dos nombres enters pot produir un nombre no enter, de manera que caldria emmagatzemar el resultat en una variable de tipus `float` per tal de no perdre informació.

Cal puntualitzar que qualsevol operació que resulti en un valor enter (suma, resta, multiplicació i mòdul de nombres enters) sempre serà de tipus `int` si el valor resultant pot ser representat per enter; en cas contrari es farà servir el tipus `long`.

1.3.3. Operadors relacionals

Els **operadors relacionals**, també anomenats **comparatius**, són símbols que ens ajuden a expressar relacions entre dues entitats com per exemple “més gran que”, “més petit o igual que”, etc.

El resultat de les comparacions serà un valor booleà, cert o fals, segons quin sigui el resultat.

Les operacions relacionals que Java ens ofereix són:

Operador	Significat	Exemple (cert)	Exemple (fals)
>	Major que	5 > 3	-99 > 13
>=	Major o igual que	5 >= 3.2	6 >= 19
<	Menor que	-3 < -2	4 < 1
<=	Menor o igual que	-3.9 <= 9.8	9.8 <= -5
==	Igual que	3 == 3	3 == 4
!=	Diferent de	2 != 7	2 != 2

Com podeu veure, es poden comparar elements de diferents tipus numèrics. El compilador s’encarrega de fer les conversions corresponents per tal que els operands siguin del mateix tipus.

1.3.4. Operadors lògics

Els **operadors lògics** són aquells que fem servir per a operar amb valors booleans. El resultat d'aquestes operacions és sempre un valor booleà.

En la taula següent es presenten alguns exemples d'operadors lògics:

Operador	Significat	Exemple
`&&`	AND lògica	`expressio1 && expressio2`
`||`	OR lògica	`expressio1 || expressio2`
`!`	NOT lògica	`!expressio1`

Una curiositat sobre els operadors lògics `&&` i `||` és que avaluen les expressions per ordre (cosa que no succeeix amb els altres operadors) de manera que un cop podem garantir el resultat de l'operació deixem de valorar l'expressió. És a dir, en l'exemple anterior, l'operador `&&` avaluaria primer `expressio1` i, en cas que fos certa, com que encara no en podem garantir el resultat, avaluaria el segon operand (`expressio2`). En cas contrari, es deixaria d'avaluar l'expressió i es donaria directament el resultat de fals. En el cas d'una `OR`, l'expressió es deixa d'avaluar quan un dels operands dóna un resultat de cert.

L'operació XOR lògica no està prevista dintre el llenguatge Java, però es pot escriure fàcilment com a:

```
(op1 || op2) && !(op1 && op2)
```

1.3.5. Operadors a nivell de bit

En un ordinador totes les dades es transformen en zeros (0) i uns (1). Per tant, calen algunes operacions per a treballar a nivell de bit amb els valors de les dades.

Treballar a nivell de bit...

... vol dir que les operacions s'executen bit a bit i que el resultat de l'operació és el valor de concatenar tots els bits.

De les 6 operacions existents per a treballar a nivell de bit, hi ha tres formats diferents segons l'operació.

Operador	Significat	Exemple (unsigned short)
`&`	AND de bits	`5&3 = 1 , 64&63 = 0`
`|`	OR de bits	`5|3 = 7 , 170|85= 255`
`^`	XOR de bits	`5^3 = 6 ,`
`~`	complement a 1	`~5 = 65530 ,`
`<<`	desp. a l'esquerra	`5<<3 = 40 , 1<<8 = 512`
`>>`	desp. a la dreta	`5>>3 = 0 , 147>>5=4`
`>>>`	desp. a la dreta sense signe	`-2147483648 >>> 4 = 134217728`

Per a entendre una mica més el funcionament d'aquestes operacions veurem com funcionen pas a pas.

Els operands que farem servir en aquests exemples seran nombres enters de 8 bits sense signe per tal de facilitar els càlculs. Aquests són:

- $175 = 1 \cdot 2^7 + 1 \cdot 2^5 + 1 \cdot 2^3 + 1 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0$ = 10101111 (en base 2)
- $49 = 1 \cdot 25 + 1 \cdot 24 + 1 \cdot 20$ = 00110001 (en base 2)

Vegem com funcionaria l'operació `175 & 49`:

```
        1 0 1 0 1 1 1 1 = 175
&       0 0 1 1 0 0 0 1 = 49
-----------------------------
        0 0 1 0 0 0 0 1 = 33
```

Aquests mètodes fan servir les taules de veritat de les operacions lògiques, que podem resumir en:

AND	0	1
0	0	0
1	0	1

OR	0	1
0	0	1
1	1	1

XOR	0	1
0	0	1
1	1	0

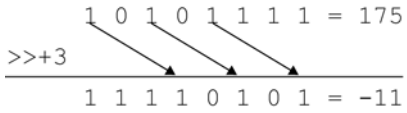
NOT	
0	1
1	0

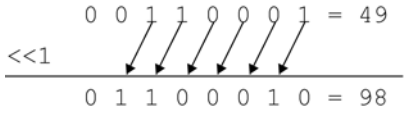
Les operacions de desplaçament de bits són també molt senzilles: es fa un desplaçament de tots els bits a la dreta o a l'esquerra.

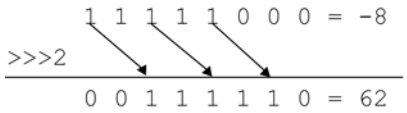
Vegem-ne un exemple per tal d'esvair qualsevol dubte. Farem 49 << 1, 175 >> 3 i –8 >>> 2:

Desplaçament del bit

Com podeu veure l'operació de desplaçar a la dreta equival a dividir per 2^n i desplaçar a l'esquerra, a multiplicar per 2^n (on n és el nombre de posicions).







1.3.6. Operadors equivalents

Els operadors que acabem de veure serveixen per a avaluar una o dues expressions (depenent de l'operador) i retornen el valor resultant.

En Java existeixen una sèrie d'operadors que ens permeten fer algunes operacions sense haver d'escriure tant, però que sovint ajuden a fer el codi més entenedor.

Operador	Exemple	Notació llarga (equivalent)	Resultat
+=	i = 3; i += 5	i = 3; i = i + 5	i = 8
-=	i = 3; i -= 5	i = 3; i = i - 5	i = -2
*=	i = 3; i *= 5	i = 3; i = i * 5	i = 15
/=	i = 3; i /= 5	i = 3; i = i / 5	i = 0
%=	i = 3; i %= 5	i = 3; i = i % 5	i = 3
&=	i = 3; i &= 5	i = 3; i = i & 5	i = 1
|=	i = 3; i |= 5	i = 3; i = i | 5	i = 7
^=	i = 3; i ^= 5	i = 3; i = i ^ 5	i = 6
<<=	i = 3; i <<= 5	i = 3; i = i << 5	i = 96
>>=	i = 3; i >>= 5	i = 3; i = i >> 5	i = 0
>>>=	i = 3; i >>>= 2	i = 3; i = i >>> 2	i = 0
++	I = 3; i++	i = 3; i = i + 1	i = 4
--	I = 3; i--	i = 3; i = i - 1	i = 2

Els operadors ++ i --

Els operadors ++ i -- s'anomenen **postincrement** i **postdecrement** quan s'escriuen a la dreta de la variable i acostumen a fer-se servir moltes vegades en els increments dels blocs iteratius.

Quan s'escriuen a l'esquerra de la variable, s'anomenen *operadors de preincrement* i *predecrement*.

No totes aquestes abreviacions solen ser d'ús comú dins els programes, però és interessant conèixer-les totes per a poder entendre el codi d'altres programadors.

1.3.7. Operador condicional

En Java existeix un operador especial, ja que rep tres operands, amb el format següent:

```
op1 ? op2 : op3
```

Aquest operador avalua `op1` (que ha de ser una expressió booleana) i depenent del resultat retorna com a valor resultant el valor de l'expressió `op2` (en cas que `op1` sigui 'cert') o el valor de l'expressió `op3` (en cas que sigui 'fals').

Pot arribar a ser complex entendre el funcionament d'aquest operador, però quedarà clar amb la seqüència d'instruccions següent:

```
int i = 0;                  // valor de i -> 0
i = (i > 0) ? -3 : 3;       // valor de i -> 3
i = (i > 0) ? -3 : 3;       // valor de i -> -3
```

Com segurament haureu deduït, l'ús d'aquest operador és equivalent al següent codi:

```
i = 0;
if (i > 0) {
   i = -3;
}
else {
   i = 3;
}
if (i > 0) {
   i = -3;
}
else {
   i = 3;
}
```

En el subapartat 1.5.1 veurem amb detall com funciona el bloc condicional `if-else`.

1.3.8. Precedència d'operadors

Fins ara les expressions que hem escrit han estat molt senzilles i no hem tingut cap problema per tal d'esbrinar el seu resultat final, però es poden escriure expressions tan complexes com es vulgui.

De vegades no és necessari escriure operacions gaire complexes per tal d'arribar a perdre el fil a l'hora de saber quin resultat donarà. Per exemple, aquesta operació:

```
i = - 3 + 4 * 3
```

Es pot interpretar de les següents formes:

```
i = -((3 + 4) * 5) → i = -35
i = -(3 + (4 * 5)) → i = -23
i = (-(3 + 4)) * 5 → i = -35
i = (-3 + 4) * 5   → i = 5
i = -3 + (4 * 5)   → i = 17
```

Com podeu veure, en un exemple senzill en què intervenen tres valors i tres operands ens han sorgit 5 interpretacions possibles de la mateixa fórmula matemàtica. Per tant, cal que el llenguatge defineixi un mètode per desfer l'ambigüitat d'aquestes fórmules (sense la necessitat de fer servir parèntesis).

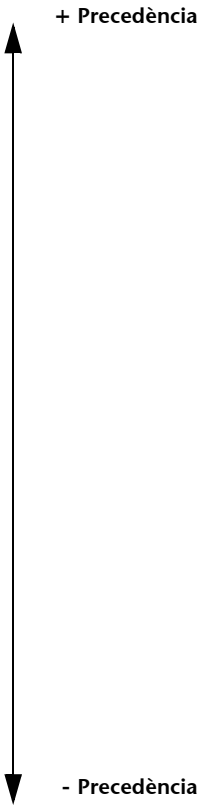
L'ús dels parèntesis

Sempre podem fer servir els parèntesis per a desfer l'ambigüitat de qualsevol expressió.

En el llenguatge de programació Java, com en altres llenguatges, aquest mètode és determinat per l'ordre de precedència dels operadors. Alguns operadors

tenen més precedència que altres i, per tant, s'apliquen primer. En la següent taula tenim un resum de les precedències:

Tipus	Operadors
Assignació	=, *=, /=, +=, -=, %=, >>=, <<=, >>>=, &=, |=, ^=
Unitaris	++, --, &, ~, -
Multiplicació	%, *, /
Additius	+, -
Desplaçaments	<<, >>, >>>
Relacionals	<, <=, >, >=
	==, !=3
Bit a bit	&
	^
	|
Lògics	&&
	||
Ternari	? :



Aquesta precedència es pot modificar sempre fent servir els parèntesis, com en l'exemple anterior, agrupant els operands amb el seu operador.

Una característica de Java, que també tenen molts altres llenguatges de programació, és que no especifica en cap moment l'ordre d'execució de les operacions dintre de la mateixa precedència. Per tant, suposant que el resultat d'una operació depengués d'una altra, és aconsellable fer dues operacions diferents per tal d'evitar obtenir resultats no esperats.

En el següent exemple no podem assegurar si s'avaluarà primer el valor del preincrement i després es calcularà la multiplicació o si es farà en ordre invers:

```
i = 3;
i = ++i + i * j;
```

Per a evitar possibles problemes és aconsellable, doncs, escriure el codi de la següent forma:

```
i = 3;
i_aux = i * j;
i = ++i + i_aux;
```

1.4. Matrius i vectors

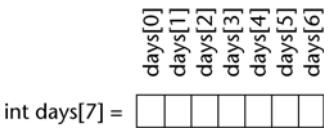
Un ***array*** és una col·lecció d'elements del mateix tipus que s'identifiquen mitjançant un índex. Els *arrays* d'un sol índex també s'anomenen **vectors**. També hi pot haver ***arrays* multidimensionals** (anomenats **matrius**), és a dir, que en comptes d'accedir a l'element amb un índex, ens cal una col·lecció de valors (**tupla**) que indexen la posició demanada.

Per a definir una matriu en Java, cal indicar el tipus de dades que s'hi emmagatzemaran, el nom amb el qual se l'identificarà i el nombre màxim d'elements que s'hi volen emmagatzemar. Si volem definir una matriu que es digui days per a emmagatzemar-hi un nombre per cada dia de la setmana, ho faríem de la següent manera:

```
int j;
int days[];
...
days = new int[7];
...
days[3] = 5;
j = days[5];
```

Cal tenir en compte que en Java, com en altres llenguatges de programació, les matrius comencen en la posició 0.

La matriu anterior tindria els següents elements:



Per tal de definir matrius (*arrays* multidimensionals), cal definir la capacitat de cada dimensió. El sistema per a accedir a cada cel·la és el mateix que en el cas anterior, només que ara ens caldrà especificar *N* índexs.

```
type name[][]...[] = new type[size_d1][size_d2]...[size_dn];
```

Per a entendre la utilitat de les matrius podem pensar en un administrador d'una pàgina web que vulgui emmagatzemar el nombre de visites que es fan a la seva pàgina durant una setmana. Per tal d'emmagatzemar les dades, l'admi-

nistrador pot definir una matriu de 3 dimensions, la primera de les quals li pot servir per a identificar el dia de la setmana, la segona per a l'hora de la visita i la tercera dimensió per als minuts. Aleshores, per a enregistrar un accés el dimarts a les 15 hores i 24 minuts, hauríem d'accedir de la següent manera:

```
int visits[][][] = new int[7][24][60];
...
visits[1][14][23]++;
```

1.5. Blocs d'instruccions

En el llenguatge Java, un **bloc d'instruccions** és un seguit d'ordres separades per un punt i coma que s'acostumen a escriure en línies diferents per tal de millorar-ne la llegibilitat.

Els blocs d'instruccions es poden agrupar fent servir les claus { i } de manera que les instruccions que estan situades dintre de les claus es comporten com si fossin una única sentència.

Els blocs d'instruccions es poden classificar principalment en dos grups, blocs condicionals i blocs iteratius, que passem a veure a continuació.

1.5.1. Blocs condicionals

Els **blocs condicionals** són aquells que serveixen per a prendre decisions i segons el resultat d'aquestes realitzar un seguit d'accions o un altre.

En el llenguatge Java, hi ha tres estructures sintàctiques que ens permeten escriure sentències condicionals: els blocs `if-else`, `else-if` i `switch`. Les tres estructures són equivalents, però el que les diferencia és que les dues últimes són una manera més còmoda d'expressar una composició de sentències `if-else`, és a dir, ofereixen més claredat al codi.

1) Bloc `if-else`

El bloc `if-else` serveix per a denotar dos blocs d'instruccions que s'han d'executar de manera excloent, segons l'avaluació d'una expressió lògica. Per exemple podem expressar termes com "si és dilluns fer... en cas contrari ...".

La **sintaxi** d'aquest bloc d'instruccions és:

```
if (expression) {
   // true expression
}
else {
   // false expression
}
```

Les expressions que utilitzem han de generar un valor booleà.

És obligatori que l'expressió estigui delimitada per parèntesis, ja que forma part de la sintaxi del llenguatge Java. En canvi, les claus són elements opcionals que només serveixen per a delimitar un bloc d'instruccions que s'han d'executar segons el resultat de l'expressió.

Recomanem...

... fer servir sempre claus per tal de fer més aclaridor el codi.

Depenent de quin sigui l'algorisme, aquest bloc d'instruccions potser no és necessari. Per tant, podem escriure només el codi que pertany al cas que l'expressió és certa i aleshores la sintaxi de construcció resulta d'aquesta manera:

```
if (expression) {
 // true expression
}
```

2) Bloc `else-if`

El bloc anterior ens serveix si volem discriminar entre el valor cert i fals d'una expressió, però resulta incòmode si volem executar un bloc d'accions diferent segons el valor de diferents condicions, com per exemple, expressar condicions de l'estil "si és dilluns fer... si és dimarts... si és dimecres...".

La **sintaxi** d'aquesta construcció és la següent:

```
if (expression1) {
   // true expression1
}
else if (expression2) {
   // true expression2
}
else if (expression3) {
   // true expression3
}
...
else {
   // false all expressions
}
```

Les expressions que utilitzem han de generar un valor booleà.

Si ens hi fixem, veurem que aquesta construcció no deixa de ser una construcció `if-else` en què els segons blocs d'instruccions (els del `else`) no tenen claus.

3) Bloc `switch`

El bloc `switch` ens ajuda a escriure condicions en què volem avaluar una expressió i fer un seguit d'operacions o unes altres segons el resultat. En realitat, aquestes condicions es poden escriure també amb una successió de blocs `if-else`, però resulta molt més còmode fer-ho amb la construcció `switch`.

Vegem la **sintaxi** d'aquesta construcció.

```
switch (expression) {
   case (value1) :
      // Code for value
      break;
   case (value2) :
      // Code for value2
      break;
   ...
   default :
      // Code for other values
}
```

L'expressió ha de retornar un valor de tipus char o qualsevol tipus de dades que es pugui convertir en un enter sense perdre precisió (`byte`, `short`, `int`), que es compara amb els valors de les clàusules `case` i s'executa el codi de la clàusula que correspongui. Tos els valors de les clàusules case han de ser diferents. En cas que no es compleixi cap de les clàusules, s'executaria el codi de la clàusula `default`, que es considera l'acció per defecte, en cas que n'hi hagi. Si aquesta clàusula no existeix, no s'executarà cap bloc i es continuarà amb la instrucció següent.

`if` i `switch`

Al contrari que la instrucció `if`, en el bloc `switch` l'expressió no s'avalua a 'cert' o 'fals'.

Un cop s'hagi trobat un valor que sigui igual al resultat de l'expressió, s'executarà el codi fins a trobar la sentència `break`. Llavors se surt de l'execució del bloc condicional i es continua per la instrucció següent. Si volem executar un mateix bloc d'instruccions per a diferents valors, caldrà escriure les instruccions `case` una darrera de l'altra sense la paraula clau `break`.

```
switch (expression) {
   case (value1) :
      // Code for value
   case (value2) :
      // Code for value & value2
      break;
   ...
   default :
      // Code for other values
}
```

1.5.2. Blocs iteratius

Els **blocs iteratius** serveixen per a executar un conjunt d'accions més d'una vegada.

En el llenguatge Java tenim tres blocs iteratius diferents: `while`, `do-while` i `for`.

Les tres estructures serveixen per al mateix, però cadascuna té unes característiques diferents. Les veurem amb més deteniment a continuació.

1) Bloc `while`

El bloc `while` és el bloc iteratiu més genèric. Serveix per a expressar qualsevol seqüència d'accions que s'hagin de repetir.

La **sintaxi** d'aquest bloc és la següent:

```
while (condition) {
   // actions
}
```

Les condicions que utilitzem han de generar un valor booleà.

Per a entendre'l millor, vegem quin seria el diagrama d'execució d'un bloc `while`.

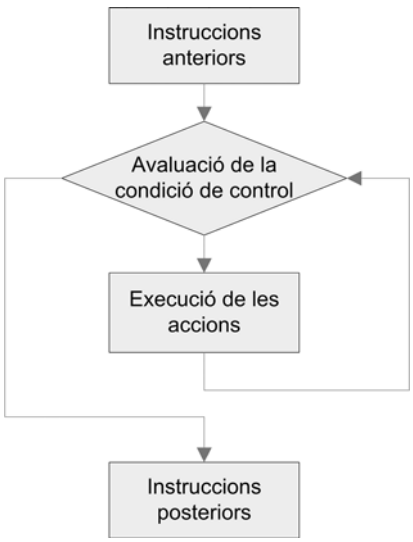


Figura 1. Diagrama de la sintaxi d'un bloc `while`

El bloc `while` funciona de manera que les instruccions dins d'aquest bloc s'executen de manera continuada mentre la condició sigui certa. Quan l'avaluació de la condició doni fals, llavors es passarà a executar les instruccions que hi ha a continuació del bloc iteratiu. La condició s'avalua abans de cada iteració.

Sabíeu que...

... per a fer un bucle infinit s'ha d'escriure només això?:

`while(true)`

Cal anar amb molt de compte per tal de no codificar bucles infinits, és a dir, trossos de codi que no deixen mai d'executar-se. Els bucles infinits són produïts principalment per funcions de fita mal escollides.

Funcions de fita

Una funció de fita és aquella que fem servir per a assegurarnos que un bloc iteratiu s'executarà sempre un nombre fitat de vegades.

2) Bloc `do-while`

Un altre tipus de bloc iteratiu és el bloc `do-while`, que funciona de manera similar al bloc `while`, però amb la diferència que la condició s'avalua després de l'execució del bloc d'instruccions.

La **sintaxi** d'aquest bloc d'instruccions és la següent:

```
do {
   // actions
} while (condition);
```

En aquest cas també poden aparèixer els bucles infinits. Per tant, també hem d'anar amb compte en definir les funcions de fita.

3) Bloc `for`

Finalment, per a acabar els blocs iteratius, tenim el bloc `for`. Aquesta construcció la farem servir principalment quan vulguem repetir certes instruccions un nombre determinat de vegades, per exemple, "recórrer els elements d'una matriu".

La sintaxi que segueixen els blocs iteratius que es construeixen amb la sentència `for` és la següent:

```
for (initialization; control condition; increment) {
 // actions
}
```

Aquesta construcció (figura 2) té un apartat en què s'inicialitza l'execució, normalment donant valors a les variables que ens serviran per a controlar el nombre d'iteracions; un segon apartat on posem la condició d'execució del bucle; i finalment el tercer apartat, que es fa servir per a incrementar les variables.

Sabíeu que...

... per a construir un bucle infinit només cal escriure això?:

```
for(;;)
```

A continuació (figura 2) teniu un diagrama que us ajudarà a entendre millor el funcionament d'aquesta construcció.

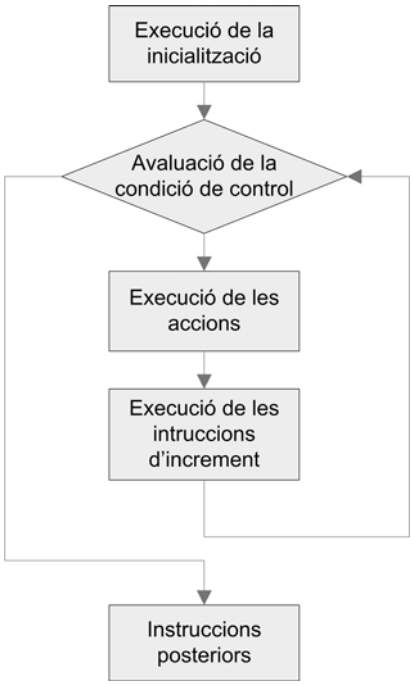


Figura 2. Diagrama de la sintaxi d'un bloc `for`

Com hem vist, el bloc `for` està compost per tres parts. Aquestes parts poden estar buides, però recomanem que sempre es posin les sentències d'inicialització, condició de control i increment per tal de fer més entenedor el codi i evitar possibles problemes en la posterior execució.

1.5.3. Sentència `break`

La **sentència `break`** serveix per a interrompre l'execució d'un bloc iteratiu en qualsevol moment.

Recordeu

La sentència `break` també es fa servir en el bloc `switch`.

Encara que el llenguatge Java permet aquesta possibilitat, no és gens recomanable, ja que fa difícil la comprensió del codi i sempre es pot escriure el codi de manera que aquesta sentència no sigui necessària.

1.5.4. Sentència `continue`

La sentència **`continue`** també serveix per a alterar el funcionament dels blocs iteratius, però en comptes d'interrompre l'execució del bloc iteratiu, s'inicia una iteració nova.

Amb la sentència `continue`, doncs, es torna a avaluar la condició i en el cas de la construcció `for` s'executen també les instruccions d'increment de les variables.

De la mateixa manera que desaconsellem l'ús de la sentència `break`, desaconsellem també l'ús de la sentència `continue`, ja que es pot aconseguir la mateixa funcionalitat sense penalitzar la llegibilitat del codi.

1.6. Funcions

Tots els conceptes que hem vist fins ara ens han permès tenir una visió profunda del llenguatge de programació Java per a poder programar aplicacions senzilles que fan servir blocs iteratius, blocs condicionals i operacions matemàtiques.

Un concepte molt important en la programació estructurada és el concepte de *funció*.

Sabíeu que...

... normalment es denominen *funcions*, però també es poden anomenar *subrutines*, *procediments*, *subprogrames*, i en programació orientada a objectes, *mètodes*?

Una **funció** és un tros de codi que executa una tasca determinada. Aquesta tasca està constituïda per un seguit d'instruccions.

Les funcions eviten al programador haver de repetir el mateix codi diverses vegades i fan més entenedor el funcionament d'un programa. Per exemple, podríem tenir una funció que calculés la mitjana aritmètica dels valors d'un vector d'enters, o el valor màxim, etc. (qualsevol tasca que se'ns acudeixi que haguéssim d'usar diverses vegades dins el nostre programa).

1.6.1. Definició de funcions

Per a poder emprar una cosa tan útil per al programador com una funció, abans cal definir-la. La instrucció que defineix una funció també s'anomena *capçalera* de la funció.

La **sintaxi** que correspon a la capçalera d'una funció és la següent:

En aquest cas no li hem afegit el concepte de *visibilitat*. Podeu repassar aquest concepte en el mòdul "Abstracció i classificació" d'aquesta assignatura.

```
ReturnDataType FunctionName(Parameters) {
 // Code to be executed
}
```

El primer que haurem de decidir per a una funció n'és el nom. Les funcions, igual que les variables, segueixen les mateixes convencions de nomenclatura. Així doncs, intentarem posar-los un nom aclaridor que ens permeti intuir quina tasca executa la funció. Per exemple, per a anomenar una funció que calculi el màxim de dos nombres enters podem fer servir `max`, `int_max`, o qualsevol altre nom, però és aconsellable fer servir un nom prou clar (no seria adient fer servir com a nom `min`).

Vegeu les mateixes convencions de nomenclatura en Java en el subapartat 1.1.4 d'aquest mòdul.

Uns altres punts que haurem de decidir per a definir una funció són el valor que aquesta retorna i els paràmetres que rep, cosa que veurem en els subapartats següents.

1.6.2. Els paràmetres d'entrada

En la majoria de casos caldrà facilitar a la funció un conjunt de valors sobre els que volem que executi les operacions que li han de permetre dur a terme la seva tasca.

En la definició de les funcions hem d'indicar quins paràmetres rebrà la funció, el tipus de paràmetres i l'ordre. La manera de fer-ho és mitjançant una llista. Per exemple, la definició de la funció que ens calcula el màxim de dos nombres enters seria:

```
int max(int a, int b) {
```

Com podeu observar, no només es posa el tipus del paràmetre, sinó també un nom identificatiu del paràmetre. Aquests noms són els que haurem de fer servir dintre del codi d'aquesta funció per tal de referir-nos als paràmetres, com si es tractessin de variables declarades normalment. Aquests noms únicament ens serviran dintre de la funció.

1.6.3. El valor de retorn

Una funció acostuma a executar un càlcul o qualsevol tasca que retorna un resultat. Cal indicar d'alguna manera quin és el tipus de dades que retorna.

Com hem vist en la definició de funcions, el tipus de dades que serà retornat s'indica abans d'especificar el nom que tindrà i aquest pot ser qualsevol tipus de dades d'entre els que hem explicat, `int`, `double`, `char`, etc., o un objecte d'una classe.

En l'exemple del punt anterior la funció `max` retorna un valor de tipus `int`.

Per a retornar un valor cal que fem servir la sentència `return`, que té el format següent:

```
return expression;
```

Aquí, `expression` pot ser qualsevol expressió que generi un valor del tipus definit en la capçalera de la funció. Tan bon punt s'executa una sentència `return`, es retorna al punt del codi on la funció ha estat cridada.

Hi ha funcions que no retornen cap valor, sia perquè modifiquen variables globals, sia perquè fan tasques com l'escriptura en un arxiu i/o per pantalla. Aquestes funcions es poden declarar de la mateixa manera que les altres, únicament cal indicar que retornen un element de tipus `void`.

1.6.4. Un exemple de funció

Per a aclarir les idees sobre les funcions, vegem com seria una funció que calcula el màxim de dos enters passats com a paràmetres.

```
public int max (int a, int b) {
 int aux;
 if (a < b) {
 aux = b;
 }
 else {
 aux = a;
 }
 return aux;
}
```

Aquesta funció rep dos paràmetres d'entrada, `a` i `b`, tots dos enters, i en retorna el valor màxim.

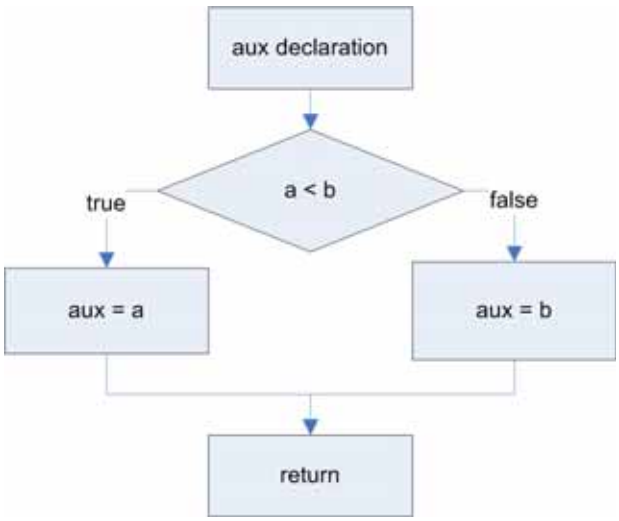


Figura 3. Diagrama de flux de la funció `max`

La funció determina quin és el valor màxim mitjançant un bloc condicional `if-else`. Fa servir una variable auxiliar anomenada `aux` per a emmagatzemar el resultat, i finalment retorna el valor d'aquesta variable auxiliar.

1.6.5. La invocació de funcions

Ja hem après a definir i omplir de contingut una funció. Ara ens cal saber com fer-ne la crida, o invocació, per a obtenir-ne el resultat. De fet, no cal fer res especial, únicament escriure el nom de la funció juntament amb els paràmetres d'aquesta en l'instant en què aquesta s'ha d'executar.

Podríem dir que una **crida a una funció** és exactament igual que avaluar una expressió que donés com a resultat el valor retornat per la funció.

Assignem el resultat de la invocació a una variable:

```
int i;
i = max(3, 4);
```

Fem servir el resultat com a operand d'una expressió:

```
if (8 < max(9,3)) {
 ...
}
```

O simplement no recollim el valor retornat:

```
System.out.println("I'm writting this on the screen\n");
```

El mètode `println` escriu per la sortida estàndard (normalment la pantalla) la cadena donada.

1.7. Visibilitat de les variables

Fins ara no ens hem preocupat de veure si podíem accedir o no al valor de les variables, sempre hem fet servir variables que declaràvem just abans d'usar-les i, per tant, sempre eren accessibles. Però això no sempre és així.

Per a entendre la visibilitat de les variables cal recordar un concepte ja explicat, el bloc d'instruccions. Recordem que un bloc d'instruccions és un conjunt de sentències que estan delimitades per una `{` i una `}`.

Definim la **visibilitat d'una variable** (també anomenat **àmbit d'una variable**) com la regió d'un programa des d'on es pot fer referència a aquesta variable pel seu nom i accedir al seu valor.

Segons el tipus de variable i, principalment, segons la seva visibilitat, podem tenir variables locals i variables globals, que passem a explicar tot seguit.

1.7.1. Variables locals

Les **variables locals** són aquelles que només són accessibles des de dins d'un bloc d'instruccions.

A continuació tenim un exemple d'una possible implementació de la funció `max`.

```
int max(int a, int b) {
   if (a > b) {
      int aux = a;
   }
   else {
      int aux = b;
   }
   return aux;          // Compilation error
}
```

En l'exemple anterior, el compilador ens donarà un error perquè en la sentència `return` no és visible cap variable amb nom `aux`. Les variables `aux` deixen de ser visibles amb la `}`. Per a solucionar aquest problema cal escriure la funció d'aquesta manera:

```
int max(int a, int b) {
   int aux;
   if (a > b) {
      aux = a;
   }
   else {
      aux = b;
   }
   return aux;
}
```

D'aquesta forma s'amplia la visibilitat de la variable `aux`, i ja no ens donarà cap error de compilació.

De la mateixa manera que no podem accedir a variables que estan fora del nostre àmbit de visibilitat, tampoc podem accedir a variables que es troben en altres funcions.

```
int max(int a, int b) {
   int aux;
   ...
   return aux;
}

int min(int a, int b) {
   if (a > b) {
    aux = b; // Compilation error. Variable res not defined
   }
   ...
   return aux;
}
```

1.7.2. Variables globals

Les **variables globals** són aquelles que són accessibles des de qualsevol punt de l'execució del programa, fins i tot des de dintre de funcions diferents.

Normalment, la presència de variables globals en un programa acostuma a denotar un mal disseny de les aplicacions.

2. Java com a llenguatge de programació orientada a objectes

Com hem vist, Java es pot utilitzar per a crear programes fent servir el paradigma de la programació estructurada, però en realitat on s'aconsegueix aprofitar al màxim la potència del llenguatge és amb la programació orientada a objectes. En aquest apartat veurem les possibilitats que ens ofereix Java per al treball emprant classes, concepte cabdal de la programació orientada a objectes. En primer lloc, veurem com es defineix una classe, els seus mètodes i atributs. A continuació, com aquesta classe es relaciona amb altres classes per tal de modelitzar la realitat necessària i resoldre el problema plantejat. Després, tractarem alguns temes avançats per no quedar-nos en la superfície, però atès que això no és un manual de Java, sinó només una introducció per a poder aprendre els conceptes de la programació orientada a objectes, altres temes com ara l'explicació exhaustiva de l'API de Java, o el disseny d'interfícies gràfiques queden exclosos d'aquest manual.

2.1. Definició de classes

Per començar a treballar amb programació orientada a objectes cal que aprenguem a definir una classe amb els seus atributs i mètodes. Més endavant veurem com representem en Java les relacions entre classes i la invocació de mètodes d'altres classes.

2.1.1. La classe

Per a escriure una classe en Java cal definir i codificar tots els mètodes i atributs que la componen. Haurem de crear un arxiu amb extensió `.java` en què escriurem tant la definició de la classe com el codi d'aquesta (en Java no hi ha altra manera de fer-ho). Vegem, primer de tot, quina és l'estructura de l'arxiu d'una classe, i després ja veurem com l'omplim de contingut.

Recordeu

En Java, la classe `Person` ha d'anar a l'arxiu `Person.java`; si no ho fem així, el compilador donarà un error.

`MyFirstClass.java`

```
public class MyFirstClass {

   private int myVar;

   public int myPublicMethod(int oneVar) {
      // some code here
   }

   private int myPrivateMethod(int anotherVar) {
      // some more code
   }
}
```

La classe es defineix com a `public` perquè es pugui fer servir des d'altres programes.

Per una banda, es denoten els atributs i, posteriorment, les signatures dels mètodes tenint en compte la seva visibilitat. Tot i que aquesta estructura no està imposada pel llenguatge, és una bona pràctica per tal de guanyar en claredat al codi i permetre una millor lectura a terceres persones.

Per a recordar el concepte de visibilitat, repasseu el mòdul "Abstracció i classificació" d'aquesta assignatura.

Per tal de millorar el nostre codi es recomana que tots els atributs siguin privats i que només s'exposin aquells que siguin imprescindibles mitjançant mètodes consultors i/o modificadors; d'aquesta manera utilitzem el concepte d'ocultació d'informació i l'encapsulament.

Per tal de declarar un atribut, únicament hem d'escriure quina serà la seva visibilitat, seguit del tipus (pot ser tant un tipus de dada bàsic com qualsevol tipus d'objecte) i posteriorment el seu nom.

Declararem els mètodes escrivint inicialment la visibilitat, seguida de la seva signatura, és a dir, el valor que retorna, el nom del mètode i els seus paràmetres. Un cop definida la signatura escriurem entre claus el codi corresponent al mètode.

2.1.2. Els atributs d'una classe

Els atributs d'una classe es defineixen utilitzant la següent construcció:

```
    visibilityMode type name;
```

On, com ja hem comentat abans, cal indicar inicialment la visibilitat que tindrà l'atribut, posteriorment el tipus i posteriorment el nom d'aquest (seguint les convencions d'aquest mòdul).

Vegeu les convencions de nomenclatura en el subapartat 1.1.3 d'aquest mòdul.

Veiem com definiríem un atribut de la classe `Person`, en aquest cas, el nom de la persona.

`Person.Java`

```
public class Person {
   private String name;
}
```

Es poden definir atributs i mètodes com a `public`, `private` i `protected`. Aquests modificadors ens permeten indicar la visibilitat tant dels mètodes com dels atributs.

Els atributs de classe

Un atribut de classe pot servir-nos, per exemple, per a poder comptabilitzar el nombre d'instàncies d'una classe que hem creat, etc.

De la mateixa manera, si volem definir un atribut de classe, és a dir, un atribut que tingui el mateix valor per a qualsevol de les instàncies de la classe, cal indicar el modificador `static` davant del seu tipus.

2.1.3. Mètodes d'una classe

Arribats a aquest punt, ja som capaços de crear una classe amb els seus atributs. El nostre objectiu és tenir una classe plenament funcional amb la qual poguem interaccionar invocant mètodes tant per a consultar-ne dades com per a modificar-ne l'estat.

Vegem com s'ha de fer per tal de poder definir mètodes en una classe.

1) El mètode constructor

El **mètode constructor** és el que s'utilitza per a crear noves instàncies d'aquella classe.

Es pot definir més d'un mètode constructor. Vegeu la sobrecàrrega de mètodes en el subapartat 2.1.9 d'aquest mòdul.

El mètode constructor ha de tenir el mateix nom que la classe que es defineix, però pot tenir qualsevol nombre de paràmetres, segons es requereixi.

A continuació, tenim un exemple de la classe `Person` amb un mètode constructor que únicament assigna a un atribut la cadena de text passada com a paràmetre.

`Person.java`

```
public class Person {
   private String name;

   public Person(String pName) {
   name = pName;
   }
}
```

Com podem veure, el mètode constructor no retorna cap valor. Això es deu al fet que, en realitat, retorna una instància de la classe.

Dins del mètode constructor es duen a terme les principals tasques d'inicialització dels elements de la instància de la classe. Encara que es poden realitzar mitjançant crides a altres mètodes, és altament recomanable fer-les totes en el moment de la creació de l'objecte per a prevenir errors.

El mètode constructor, com qualsevol altre mètode, permet la sobrecàrrega; això vol dir que podem tenir definits diversos mètodes constructors amb diferents paràmetres.

Vegeu la sobrecàrrega de mètodes en el subapartat 2.1.9 d'aquest mòdul.

A més a més, el llenguatge de programació Java, en cas que definíssim una classe sense cap mètode constructor, en genera un de manera automàtica sense cap paràmetre i que no executa cap tasca.

2) El mètode destructor

De la mateixa manera que tenim un mètode per a crear una instància d'una classe, hi ha un mètode per a destruir una instància d'una classe.

Aquest mètode és cridat de manera automàtica en perdre's la visibilitat d'una instància de la classe. Per tant, haurem de realitzar les tasques necessàries per tal d'alliberar tots els possibles recursos reservats (tancar arxius, alliberar memòria, connexions a bases de dades, etc.).

Pèrdua de visibilitat d'un objecte

Un objecte perd la seva visibilitat quan ja no és accessible fent servir el nom de la variable que l'identificava (vegeu el subapartat 1.7 d'aquest mòdul).

El compilador de Java, per defecte, ens crea un mètode destructor. Tot i així, ens pot interessar crear-ne un a nosaltres mateixos. El mètode destructor s'anomena `finalize` i no rep cap paràmetre.

La classe `Person` que hem definit abans quedaria de la següent manera:

`Person.java`

```
public class Person {
   private String name;
   public Person(String pName) {
      name = pName;
   }
   public void finalize() {
      // do something to erase}
   }
}
```

En aquest cas no hem efectuat cap tasca dins el mètode destructor, atès que en el mètode constructor no hem demanat recursos, però és una cosa que hem de tenir en compte per tal de crear aplicacions correctes.

3) La resta de mètodes

Els dos tipus de mètodes anteriors ens han servit per a inicialitzar una nova instància de la classe i per a alliberar recursos un cop aquesta instància es deixa de fer servir. Ara ens cal veure com hem de definir la resta de mètodes de la classe: els mètodes consultors i els mètodes modificadors.

De la mateixa manera que hem fet amb el mètode constructor i el destructor, hem d'indicar la signatura de les operacions: el nom, el tipus del valor de retorn (en cas que no retorni cap valor, indiqueu-ho mitjançant la paraula `void`), i el nom i tipus de cada paràmetre.

Si seguim amb l'exemple anterior de la classe `Person`, ara haurem d'implementar un parell de mètodes per a recuperar i modificar el nom de la classe. Aquests mètodes es definirien de la següent manera:

`Person.java`

```
public class Person {
   private String name;

   public Person(String pName) {
      name = pName;
   }

   public void finalize() {
      // do something to erase
   }

   public String getName() {
      return name;
   }

   public void setName(String pName) {
      name = pName;
   }
}
```

En aquest cas, el mètode `getName` no rep cap paràmetre i retorna una cadena de text i el mètode `setName` no retorna cap valor i rep com a paràmetre una cadena de text.

2.1.4. Ús de la paraula reservada `this`

En l'exemple anterior hem definit i implementat el mètode `setName` de la següent manera:

`Person.java`

```
...
   public void setName(String pName) {
      name = pName;
   }
...
```

Però res no ens impedeix de fer-ho d'aquesta manera:

`Person.java`

```
...
   public void setName(String name) {
      name = name;
   }
...
```

Per tant, tenim un problema dintre del mètode `setName`, ja que no podem diferenciar de manera clara l'atribut `name` del paràmetre `name`. Aquesta diferenciació es pot assolir mitjançant la paraula reservada `this`.

Per a evitar possibles ambigüitats, en la implementació d'aquests mètodes cal escriure la paraula `this` davant dels atributs de la classe. La paraula clau `this` ens permet referir-nos a l'objecte mateix. El codi resultant quedaria d'aquesta manera:

`Person.java`

```
...
   public void setName(String name) {
      this.name = name;
   }
...
```

La paraula reservada `this` també serveix per a denotar que el mètode que invoquem està definit dins la mateixa classe.

2.1.5. Mètodes estàtics

Els mètodes que hem definit fins ara retornaven o modificaven informació referent a la instància de la classe sobre la qual s'invocava el mètode. Ara bé, ens pot interessar tenir un mètode que no depengui de l'estat de la instància sobre la qual s'executa, és a dir, que retorni un mateix resultat per a uns mateixos paràmetres, sense importar sobre quina instància concreta s'invoqui. Aquests mètodes s'anomenen mètodes `static`.

Un exemple podria ser un mètode que, donada una data, hi sumés o en restés una certa quantitat de dies. El normal seria implementar-lo en una classe `Date` i definir-lo d'aquesta manera:

Accessibilitat dels mètodes

Un mètode estàtic només pot accedir a atributs estàtics; en canvi, un mètode no estàtic pot accedir a atributs tant estàtics com no estàtics.

`Date.java`

```
class Date {
...
   public static Date addDates(Date pDate, int days) {
      // some code here
   }
}
```

2.1.6. Sobrecàrrega de mètodes

En aquest moment ja tenim una classe creada, amb atributs i mètodes. Però, amb el que hem explicat fins ara, tenim la limitació que només podem tenir

un mètode amb el mateix nom. Això representaria haver de definir diferents mètodes per a la mateixa tasca amb noms diferents; per exemple, tenir una classe que s'encarrega de calcular operacions matemàtiques i voler definir una operació per a fer sumes de dos nombres.

Per tant, tindríem les següents definicions de mètodes:

MathOperation.java

```
class MathOperation {
   public static int add_int_int(int a, int b) {...}
   public static int add_int_long(int a, long b) {...}
   public static int add_int_float(int a, float b) {...}
   public static int add_int_int_int(int a, int b, int c) {...}
   ...
}
```

Encara que aquesta manera de treballar no és incorrecta, tampoc no és la millor, ja que la nomenclatura es torna feixuga. És més aconsellable fer servir la sobrecàrrega de mètodes.

La **sobrecàrrega de mètodes** ens permet definir diferents mètodes amb el mateix nom dins d'una mateixa classe, sempre que el nombre i l'ordre dels tipus dels paràmetres sigui diferent.

Amb aquest concepte, la manera més correcta de definir l'anterior classe seria aquesta:

MathOperation.java

```
class MathOperation {
   public static int add(int a, int b) {...}
   public static int add(int a, long b) {...}
   public static int add(int a, float b) {...}
   public static int add(int a, int b, int c) {...}
   ...
}
```

D'aquesta manera, l'usuari sap que aquesta classe li permet calcular sumes de nombres i que el mètode utilitzat sempre es diu `add`.

2.2. Les relacions entre classes

Vegeu les relacions entre classes en el mòdul "Abstracció i classificació" d'aquesta assignatura.

En aquest apartat comentarem els principis bàsics per a implementar les relacions entre classes, que ja hem vist pel que fa a disseny.

2.2.1. Cardinalitat

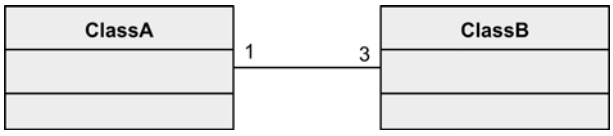
Com ja hem dit, una relació sempre té una cardinalitat associada que ens indica el nombre d'instàncies amb les quals pot estar relacionada.

Per tal de representar la cardinalitat de les relacions de la programació orientada a objectes en Java, hem de tenir una referència a objectes a la nostra classe en forma d'un atribut que representarà aquesta relació.

Atès que tenim diferents tipus de cardinalitat (nombre exacte, rang de valors i cardinalitats indefinides), cal una manera diferent per a cada tipus. Vegem com s'implementen cadascuna d'aquestes tipologies.

2.2.2. Nombre exacte

Considerem el diagrama de classes següent:



Observem que una instància de `ClassB` sempre està relacionada amb una (1) única instància de `ClassA`. Per tant, definiríem a `ClassB` el següent atribut del tipus `ClassA` per tal de representar aquesta relació.

`ClassB.java`

```
class ClassB {
   ClassA a1;
   ...
}
```

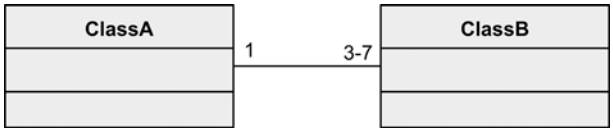
Per a implementar completament la relació caldria definir tres atributs del tipus `ClassA` que representarien l'altre costat de la relació. Aquesta manera de treballar ens planteja un greu problema: què fem si tenim una cardinalitat molt gran? No resulta pràctic definir 100 atributs amb noms similars, ja que aquests noms resulten poc intuïtius per al programador. En aquest cas caldria crear un vector amb capacitat per al nombre d'instàncies amb les quals estarà relacionat i inserir cadascuna de les instàncies en una posició d'aquest.

Una altra possibilitat

Una altra manera de solucionar aquest problema és fent servir la solució que proposarem per als valors indefinits (subapartat 2.2.4).

2.2.3. Rang de valors

En el cas que tinguem un rang de valors com en el següent diagrama:



Ens trobem amb el mateix problema que teníem en el cas anterior, pel fet de tenir una cardinalitat superior a 1. Definir un atribut per cada instància és factible només per a pocs valors. És més comú definir un vector que ens permeti emmagatzemar totes les instàncies.

Així, en l'exemple anterior definiríem un vector de 7 posicions a `ClassA` d'elements a `ClassB`. Atès que tenim un mínim de 3 instàncies relacionades i un màxim de 7, si definim el vector de mida inferior, ens podríem trobar sense prou espai per a emmagatzemar les dades corresponents. Sempre definirem el valor màxim dins del rang.

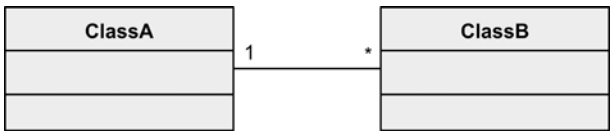
2.2.4. Valors indefinits

En el cas que tinguéssim un nombre indefinit de valors (una relació tipus `*`) en comptes d'un rang predefinit, la solució més òptima (tot i que possible) no és un vector, ja que cada vegada que hi inseríssim un element i n'excedíssim la capacitat, hauríem de tornar a crear un vector nou de més capacitat i copiar-hi els elements. Hem de buscar una altra manera d'emmagatzemar aquestes instàncies.

Per a fer-ho, caldrà usar classes auxiliars que permetin un creixement dinàmic del nombre d'elements del vector. Amb aquesta intenció, podem fer servir qualsevol de les classes contenidores que ens ofereix l'API de Java. En el nostre cas proposem emprar `ArrayList`, però en altres casos i segons les necessitats que tinguem en la cerca d'instàncies de la relació podrem fer servir altres classes.

Teniu informació sobre llistes i vectors en el mòdul "Estructures d'objectes" d'aquesta assignatura.

Un exemple de com quedaria `ClassA` del següent diagrama seria:



`ClassA.java`

```
import java.util.ArrayList;

class ClassA {
   private ArrayList<ClassB> v;
   ...
}
```

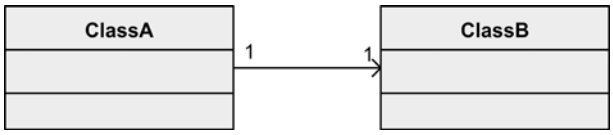
La classe `ArrayList`

La classe `ArrayList` és una classe parametritzada, la qual cosa ens permet indicar en temps de compilació el tipus d'objectes que emmagatzemarà.

2.2.5. Navegabilitat

La **navegabilitat**, com ja sabem, denota la coneixença per part d'una classe de la pròpia relació. Parlant en termes d'implementació, aquest concepte ens indica en quina de les classes hem de definir l'atribut, matriu o vector, atesa la multiplicitat de la relació.

En navegabilitats unidireccionals hem de declarar l'atribut en la classe de la qual surt la fletxa indicativa. En aquest cas, l'atribut ha d'estar declarat a `ClassA`.



En navegabilitats bidireccionals, l'atribut s'ha de declarar en totes dues classes.

2.2.6. Rols

El rol de les classes en les associacions només ens indica el nom de l'atribut que farem servir.

Encara que els atributs poden tenir el nom que ens sembli més adient, és recomanable fer servir el nom del rol per tal de mantenir una certa coherència amb el diagrama UML.

2.3. Biblioteca de classes

En Java, com en altres llenguatges de programació orientats a objectes, es disposa d'un seguit de classes ja implementades. Aquest conjunt de classes s'anomena *biblioteca de classes*.

Hi ha moltes biblioteques de classes diferents que permeten realitzar des de tasques senzilles fins a tasques molt complexes com el xifratge, el tractament d'imatges, el treball amb interfícies gràfiques, etc.

En Java, tenim un conjunt de classes que conformen l'**API de Java** i que ens ofereixen totes aquestes utilitats. Com que aquest conjunt de classes és molt gran i està en evolució constant, ateses les millores que es van afegint de mica en mica, únicament comentarem, i molt per sobre (ja que si voleu més informació sempre teniu l'ajuda oficial), un parell de classes que utilitzarem al llarg de tots els materials.

***API* és la sigla del terme anglès *application programming interface*.**

Per a completar la informació sobre qualsevol classe de l'API de Java, visiteu la següent web:
http://java.sun.com/reference/api/

En primer lloc, veurem com utilitzar la classe `String`, que ens serveix per a tractar cadenes de caràcters, i a continuació farem també una ullada a la classe `ArrayList`, que ens permet emmagatzemar objectes de qualsevol tipus.

2.3.1. La classe `String`

La **classe `String`** representa una seqüència de caràcters i ofereix un seguit d'operacions que permeten treballar-hi de manera senzilla i còmoda.

Per a poder treballar amb objectes de la classe `String`, no cal fer res d'especial, ja que aquesta classe es troba dintre de la biblioteca `java.lang`, i aquesta biblioteca s'utilitza per defecte en tots els programes Java. Per tant, el primer que hem de fer és declarar un objecte de tipus `String` i, a continuació, crear l'objecte. Això es pot fer de diverses maneres, tantes com mètodes constructors tenim (13 en aquests moments), més l'assignació d'una cadena de caràcters delimitats entre cometes dobles que té la mateixa funcionalitat.

Alguns exemples de crear objectes de tipus `String` són:

```
String s0 = "abc";
String s1 = new String();
char data[] = {'a', 'b', 'c'};
String str = new String(data);
String s3 = new String(s0);
```

Hi ha altres maneres de crear i inicialitzar objectes `string`. Les trobareu totes a l'API.

Un cop ja tenim un objecte de tipus `String` creat, podem executar-hi moltes operacions. Entre d'altres, hi ha operacions d'assignació, o operacions que ens mostren la longitud de la cadena, que ens permeten fer cerques, etc.

Vegem algunes de les opcions en forma d'operacions que aquesta classe ens ofereix:

Observació

No és l'objectiu d'aquest apartat veure totes les possibilitats, però cal saber que hi ha diverses sobrecàrregues dels mètodes aquí presentats.

- `charAt`: ens dóna el caràcter d'una posició determinada.
- `compareTo`: compara dues cadenes lexicogràficament.
- `compareToIgnoreCase`: compara dues cadenes lexicogràficament ignorant les majúscules i minúscules.
- `concat`: concatena la cadena que representa objectes amb la donada.
- `contains`: indica si existeix una cadena de caràcters donada a objectes.
- `endsWith`: comprova si la cadena acaba amb una seqüència donada.
- `indexOf`: retorna la posició d'un caràcter dintre de l'objecte.
- `lastIndexOf`: retorna l'última posició d'un caràcter dintre de l'objecte.
- `length`: retorna la longitud de la cadena emmagatzemada.
- `matches`: comprova si la cadena compleix una expressió regular.
- `replace`: reemplaça una part d'una cadena per una altra.
- `split`: parteix una cadena fent servir una expressió regular.
- `startsWith`: comprova si la cadena comença d'una determinada manera.
- `substring`: retorna una subcadena de la cadena original.
- `toLowerCase`: transforma la cadena en minúscules.
- `toUpperCase`: transforma la cadena en majúscules.
- `trim`: elimina els espais sobrants davant i darrere de la cadena.
- `valueOf`: retorna la cadena que representa l'objecte/variable donat.

Per a veure com funciona la classe `String`, a continuació tenim un tros de codi que fa servir moltes de les operacions esmentades.

`TestString.java`

```
public class TestString {

   public static void main(String[] args) {

      String s0 = "Something inside";
      String s1 = "GUAU!";
      String s2 = "The dog says ... ";
      String s3 = new String("The cat says... ");
      String s4 = new String("MIAU!");

      System.out.println("The lowercase value of s0 is : " + s0.toLowerCase());
      System.out.println("The uppercase value of s0 is : " + s0.toUpperCase());
      System.out.println("The value of s1 is : " + s1 + " and has " + s1.length() +
            " characters");
      System.out.println("The value of s2 is : " + s2 + " and has it's firts 's' on " +
            s2.indexOf('s') + " position");
      System.out.println("The trimed value of s3 is : \"" + s3.trim() + "\"");
      s2 = s2.concat(s1);
      s3 = s3.concat(s4);
      System.out.println("The concatenation of s2 and s4 is : " + s2);
      System.out.println("The first 5 characters of s3 are : \"" + s3.substring(0, 5) + "\"");
      if (s3.startsWith(s4)) {
      System.out.println("s3 starts with \"" + s4 + "\"");
      }
      else {
      System.out.println("s3 doesn't start with \"" + s4 + "\"");
      }
      if (s3.endsWith(s4)) {
         System.out.println("s3 ends with \"" + s4 + "\"");
      }
      else {
         System.out.println("s3 doesn't end with \"" + s4 + "\"");
      }
      if (s2.contains("dog")) {
         System.out.println("s3 contains the \"dog\" substring");
      }
      else {
         System.out.println("s3 doesn't contains the \"dog\" substring");
      }
      System.out.println("The value of s3 after a replacement : " + s3.replace(s4, s1));
      System.out.println("The value of a boolean false is : " + String.valueOf(false));
      System.out.println("The value of the max int value is : " +
            String.valueOf(Integer.MAX_VALUE));
      String splits[] = s2.split(" ");
      System.out.println("The content of splits is :");
      for(int i = 0; i < splits.length; i++) {
         System.out.println("[" + i + "] : " + splits[i]);
      }
   }
}
```

Les operacions que hem fet servir tenen més d'una sobrecàrrega; per tant, cal fer un cop d'ull a la documentació de la classe String per tal de conèixer totes les opcions que aquesta classe ens ofereix.

2.3.2. La classe `ArrayList`

Com ja hem vist, la classe `ArrayList` serveix per a emmagatzemar instàncies d'altres objectes i recuperar-les posteriorment per a fer el que calgui. De la mateixa manera, quan hem comentat les relacions entre classes en Java, també hem introduït els `ArrayList`; per tant, anirem directament a com podem fer per a treballar-hi.

Vegeu la classe `ArrayList` en el mòdul "Un exemple pràctic" d'aquesta assignatura. Vegeu les relacions entre classes en el subapartat 2.2 d'aquest mòdul.

Primer hem de declarar una variable de tipus `ArrayList`, com ho faríem amb qualsevol altra variable, i a continuació haurem de cridar a un dels seus mètodes constructors, com en el cas de la classe `String`. Per exemple, la declaració i inicialització posterior d'un `ArrayList` de persones tindria aquesta forma:

```
ArrayList<Person> persons;
...
persons = new ArrayList<Person>();
```

Ara cal veure quines són les operacions que podem realitzar sobre un `ArrayList`. Atès que aquest implementa la interfície `List`, podrem realitzar les mateixes operacions que aquesta defineix, més les pròpies de la classe.

De la mateixa manera que hem fet amb la classe `String`, veurem també un subconjunt de les operacions que la classe `ArrayList` té definides, ja que si volem veure-les totes, sempre podem utilitzar l'API de Java.

Per a conèixer més mètodes de la classe `ArrayList`, podeu consultar l'API.

- `add`: afegeix un element a la llista.
- `addAll`: afegeix una llista a la llista (equivalent a concatenar).
- `clear`: elimina tots els elements de la llista.
- `clone`: clona la llista, els objectes interns no són clonats.
- `contains`: comprova si un objecte existeix a la llista.
- `ensureCapacity`: incrementa la capacitat màxima de la llista.
- `get`: retorna l'element demanat de la llista.
- `indexOf`: busca la primera posició d'un element dintre de la llista.
- `lastIndexOf`: busca l'última posició d'un element dintre de la llista.
- `remove`: elimina l'element de la posició indicada.
- `removeRange`: elimina els elements del rang donat.
- `set`: substitueix l'element de la posició indicada amb l'objecte donat.
- `size`: indica la quantitat d'elements emmagatzemats.
- `toArray`: retorna un *array* amb tots els elements de la llista.
- `iterator`: ofereix un iterador per tal de poder recórrer la llista en recorreguts.

`iterator`

Un `iterator` és un objecte que ens ajuda a recórrer els elements d'un conjunt de manera més simple.

En el tros de codi següent teniu unes quantes de les operacions que es poden fer amb `ArrayList`.Com que aquesta classe emmagatzema classes, utilitzarem una classe `Person` que tenim implementada a l'arxiu `Person.java`. Aquesta classe únicament té un atribut de tipus `String` (el nom de la persona), un mètode consultor i un altre de modificador.

Aquí no tenim aquesta classe `Person` implementada, ja que no és l'objectiu d'aquest subapartat, però arribats en aquest punt no hauríeu de tenir cap problema per a implementar-la seguint els passos dels subapartats anteriors.

`TestArrayList.java`

```
import java.util.Iterator;
import java.util.List;
import java.util.ArrayList;

public class TestArrayList {

   public static void main(String[] args) {

      Person p1 = new Person("Luke");
      Person p2 = new Person("Leia");
      Person p3 = new Person("Anakin");
      Person p4 = new Person("Amidala");

      List<Person> persons = new ArrayList<Person>();
      List<Person> parents = new ArrayList<Person>();

      parents.add(p3);
      parents.add(p4);

      persons.add(p1);
      persons.add(p2);
      persons.addAll(parents);
      parents.clear();

      System.out.println("Persons list size : " + persons.size());
      System.out.println("Parents list size : " + parents.size());

      Person p = persons.get(2);
      System.out.println("Person's name : " + p.getName());

      if (parents.remove(p4)) {
      System.out.println("On the parents list there's only the dark side");
      }
      else {
         System.out.println("Something went wrong!!");
      }
      Iterator it1 = persons.iterator();
      while (it1.hasNext()) {
         System.out.println("The force is strong on : " + ((Person)it1.next()).getName());
      }
   }
}
```

Algunes de les operacions que hem fet servir tenen més d'una sobrecàrrega. Per tant, seria recomanable que, abans d'utilitzar-ne alguna, miréssiu si alguna de les sobrecàrregues existents és prou adient per a poder-se utilitzar.

2.4. Les excepcions

Una **excepció** es pot definir com l'ocurrència d'un esdeveniment inesperat durant l'execució normal d'un programa (per exemple, un error).

Una excepció ens serveix per a comunicar l'ocurrència d'esdeveniments sense haver de definir un protocol concret. Nosaltres ens centrarem en l'ús d'excepcions per al control d'errors.

Atès que en Java tot són classes, necessitem una classe que ens representi l'excepció i un mecanisme del llenguatge que ens permeti capturar els errors i tractar-los en cas de necessitat.

Pel fet de ser una classe, les excepcions ens ofereixen un nou ventall de possibilitats per al tractament d'errors, ja que ens permet encapsular totes les dades referents a l'error dintre d'una estructura i treballar-hi de manera més senzilla. Si una excepció no és tractada, l'execució del programa s'avorta immediatament. Per això és important tractar-les totes. 

2.4.1. Creació d'una excepció

En Java, qualsevol excepció ha d'heretar de la classe `Exception`, que ja tenim definida i implementada en l'API.

Aquesta nova classe ha de tenir com a mínim la següent estructura:

`MyException.java`

```
public class MyException extends Exception {

}
```

Com podem observar, no cal incloure-hi cap atribut o mètode, però llavors tenim una excepció que no transmet cap informació addicional a la que ens ofereix la pròpia classe `Exception`.

Com que la pròpia classe `Exception` ja té un atribut de tipus `String` per a emmagatzemar el missatge que donarà, un bon costum és definir missatges personalitzats que informin de la mateixa manera sobre el mateix error, i en cas que fos necessari, algun altre atribut per a poder passar alguna informació que sigui útil després per a tractar els errors.

La nostra excepció quedaria de la següent manera:

MyException.java

```
public class MyException extends Exception {

   public static String MyMessage = "Write here what you need.";

   public MyException(String message) {
      super(message);
   }
}
```

El mètode `super` invoca el mètode constructor de la classe pare.

Fixeu-vos que podem afegir qualsevol atribut i mètode que ens vingui al cap, però normalment amb aquesta definició ja en tindrem prou per a les nostres aplicacions.

En cas que vulguem declarar diversos mètodes constructors, o d'altres mètodes per a consultar els atributs que tingui la nostra excepció, simplement els declararem de la mateixa manera que en una classe qualsevol.

2.4.2. Llançament d'excepcions

Un cop tenim definida una classe que representa les nostres excepcions, caldrà poder-les utilitzar per a indicar les situacions anòmales que es produeixen durant l'execució del codi.

S'indica que s'ha produït una excepció amb la instrucció següent:

```
throw new MyException("Error description");
```

En aquest cas estem llençant una excepció del tipus `MyException` amb el text donat. Per a llençar una excepció hem de crear una instància de la classe `Exception` amb els paràmetres necessaris i posteriorment, amb l'ús de la paraula reservada `throw`, indiquem que s'ha produït l'error indicat.

Cal tenir en compte que l'execució del mètode en què s'ha llençat l'excepció queda interromput, i immediatament retornem al lloc on es va realitzar la crida, de la mateixa manera que si haguéssim realitzat un `return` en el nostre codi.

2.4.3. Tractament d'excepcions

Fins ara hem vist com podem crear excepcions i llençar-les. Ara ens cal aprendre com podem tractar-les en cas que s'hagin produït en mètodes que nosaltres fem servir en el nostre codi.

Per tal de poder tractar les excepcions correctament, cal tenir constància del lloc en què es poden produir, és a dir, cal que tinguem documentats tots els mètodes de les classes que fem servir i que aquesta documentació informi de les excepcions que pot generar.

Donat això, només hem d'incloure dintre d'un bloc de codi especial el conjunt d'instruccions que poden generar una excepció, de manera que l'ocurrència de les excepcions es detectarà sense que el programa avorti.

Aquest bloc especial d'instruccions s'anomena bloc `try...catch`, i té l'estructura següent:

```
try {
   // Code that could throw an exception
}
catch (ExceptionType exceptionVar) {
   // Code that handles an exception
}
```

Escriurem el codi que pot generar una excepció entre les claus que segueixen la paraula reservada `try`. Dins la sentència `catch` indicarem quin tipus d'excepció es pot generar; li donarem un nom de variable (per tal de poder-ne recuperar dades més tard) i hi escriurem el codi que tractarà aquesta excepció.

Quan qualsevol dels mètodes dins el bloc `try` llenci una excepció del tipus declarat al bloc `catch`, el programa passarà a executar immediatament el codi dins d'aquest bloc.

```
private static void createException() throws MyException {
   throw new MyException(MyException.MyPersonalMessage);
}

try {
   createException();
}
catch (MyException m) {
   // Code that handles an exception
   System.out.println(m.getMessage());
}
```

Per al tractament d'excepcions podem posar qualsevol codi que necessitem, tant pot ser l'escriptura en un fitxer, la pantalla, com simplement actualitzar un comptador, tot dependrà de la nostra aplicació i les seves necessitats.

Resum

En el primer apartat d'aquest mòdul hem vist una petita introducció dels orígens del llenguatge de programació Java. A part, hem anomenat algunes de les característiques principals del llenguatge, les paraules clau, i hem introduït les convencions necessàries per a anomenar les variables i les funcions.

A continuació, hem fet un repàs dels tipus de dades disponibles en el llenguatge Java per poder-les fer servir de manera correcta. Hem donat una àmplia visió dels operadors que el llenguatge Java suporta i n'hem posat diferents exemples d'utilització. En concret, hem posat un èmfasi especial en l'ordre de precedència dels operadors, atès que aquest ordre condiciona el resultat de les expressions.

A continuació, hem introduït l'*array* (vector o matriu), un tipus de dada estructurat que ens serveix per a emmagatzemar un seguit d'elements de la mateixa classe, i hem vist la manera d'emmagatzemar i recuperar les dades.

Un cop hem tingut una visió general dels tipus de dades i les operacions que podem fer amb aquests, ens hem endinsat en el món de les instruccions i les seqüències d'instruccions (els blocs condicionals i iteratius). En aquest punt hem vist les diferents estructures que hi ha per a escriure aquests blocs i les peculiaritats de cadascuna.

Per acabar el primer apartat, hem tractat les funcions, amb els paràmetres d'entrada i sortida i la visibilitat de les variables, dos conceptes que van molt lligats.

El segon apartat presenta les característiques de Java com a llenguatge de programació orientada a objectes. Primer es presenten tots els recursos relacionats amb la creació de classes, i els seus atributs i mètodes. A continuació, s'apuja de nivell d'abstracció per a definir la representació en Java de les possibles relacions que es poden donar entre classes. En un tercer subapartat, es presenten un parell de biblioteques per a veure com es poden reutilitzar certes característiques del llenguatge. I, finalment, es mostra com es fa el tractament d'excepcions en Java.

Activitats

1. Digueu quin tipus de dades faríeu servir per a...
a) comptar persones.
b) mesurar el pes de la fruita.
c) treballar amb els dies de la setmana.
d) emmagatzemar les hores d'un dia.
e) emmagatzemar el saldo d'un compte bancari.

2. Ja heu vist que els operadors en el llenguatge de programació Java tenen precedències diferents. Digueu quin seria el resultat d'avaluar les expressions següents:
a) $3 + 5 \cdot 4 / 2 - 1$.
b) $21 - 15 > 3 \cdot 9$.
c) $0 / 3 + 2 \cdot 4 - 3 \cdot 7$.

3. Hem vist com definir un *array* d'un tipus de dades bàsic (hem fet servir els `int`). Com es definiria aquest *array* si s'emprés un tipus enumerat?

Vegeu els *arrays* en el subapartat 1.4 i els tipus enumerats en el subapartat 1.2.3 d'aquest mòdul.

Solucionari

1.
a) Per a comptar persones farem servir el tipus de dades `int`, ja que per a comptar unitats s'ha de fer servir un tipus enter. Depenent de la quantitat de persones que s'haguessin de comptar es podria fer servir un tipus de dada que permetés emmagatzemar nombres més petits (`short`).

b) Per a mesurar el pes de la fruita utilitzarem un tipus de dada real; com que és fruita, no cal gaire precisió, per tant: un `float`.

c) Per a treballar amb els dies de la setmana podem fer dues coses: emprem un tipus enumerat o prenem la convenció d'enumerar els dies, per exemple, 0 = dilluns, 1 = dimarts, etc. És millor el tipus enumerat.

d) Per a emmagatzemar les hores d'un dia, com en el cas anterior, també es pot fer servir un tipus enumerat o un nombre enter, i continuem preferint el tipus enumerat per la claredat que ofereix.

e) Per a emmagatzemar el saldo d'un compte bancari necessitem un nombre real, ja que els comptes bancaris fan servir fraccions (el cèntims) d'unitats (els euros, en el nostre cas); per tant, faríem servir un tipus `float` (o un tipus `double` en un cas molt exagerat).

2.
a) $3 + 5 \cdot 4 / 2 - 1 = 3 + (5 \cdot 4 / 2) - 1 = 3 + 10 - 1 =$ **12**
b) $21 - 15 > 3 \cdot 9 = (21 - 15) > (3 \cdot 9) = 6 > 27 =$ **fals**
c) $0 / 3 + 2 \cdot 4 - 3 \cdot 7 = (0 / 3) + (2 \cdot 4) - (3 \cdot 7) = 0 + 8 - 21 =$ **–13**

3. Primer hem de declarar un tipus enumerat com hem fet abans:

```
enum days {SUN, MON, TUE, WED, THU, FRI, SAT};
```

Després hem de declarar un *array* on el tipus de dades és el tipus enumerat:

```
int days[] = new int[7];
```

A continuació, hem d'accedir als elements d'aquest tipus enumerat:

```
d[0] = MON;
d[1] = TUE;
...
```

Glossari

array *m* Agrupació fitada d'elements d'un mateix tipus.

atribut *m* Representació d'una qualitat dels elements del món real dintre d'una classe.

biblioteca de classes *f* Conjunt de classes implementades per altri que permeten la reutilització del codi.

bloc condicional *m* Conjunt d'instruccions que s'executen o no depenent d'una certa condició.

bloc iteratiu *m* Conjunt d'instruccions que s'executen reiterativament fins que es compleix una certa condició.

booleà -ana *adj* Dit del tipus de dada que representa els valors de certesa o falsedat.

caràcter *m* Tipus de dada que representa un caràcter.

classe *f* Estructura que representa un element del món real.

excepció *f* Tipus de classe que serveix per a millorar el tractament d'errors.

funció *f* Agrupació d'instruccions que permet escriure-les una única vegada.

mètode *m* Acció que es pot realitzar sobre un element del món real que té un símil en la programació orientada a objectes.

nombre enter *m* Tipus de dada que representa els valors numèrics enters.

nombre en coma flotant *m* Tipus de dada que representa els valors numèrics en coma flotant.

operador *m* Element del llenguatge que permet definir una operació sobre les dades a les quals afecta.

Bibliografia

Sierra, K. *Java 2 SUN Certified Programmer & Developer*. McGraw-Hill.

Eckel, B. *Thinking in Java* (4a. ed.). Upper saddle River: Prentice Hall.
<http://java.sun.com/j2se/1.5.0/docs/api/ (API de Java)>