Universitat de Girona
Escola Politècnica Superior

Projecte/Treball Final de Carrera

Estudi: Eng. Tèc. Informàtica de Sistemes

Títol:

Traductor de pseudocodi a java

Document: Annex: Manual d'usuari

Alumne: Carles Royán Salvatella

Director/Tutor: Josep Suy
Departament: Informàtica i Matemàtica Aplicada
Àrea: Llenguatges i Sistemes Informàtics

Convocatòria: Juny 2003

Índex

1. Introducció....2

2. Instal·lació....4
2.1 Instal·lació de les variables d'entorn en Linux....4
2.2 Instal·lació de les variables d'entorn en Windows 95/98/ME....5
2.3 Instal·lació de les variables d'entorn en Windows 2000/XP....5

3. Ús del traductor CPC....7
3.1 Flux d'informació....7
3.2 Opcions de la línia de comandes....8

4. Pseudocodi 03....9
4.1 Conceptes bàsics....9
4.1.1 Comentaris....10
4.1.2 Sentències....11
4.1.3 Blocs de codi....11
4.1.4 Paraules reservades....11
4.1.5 Identificadors....12
4.1.6 Literals i tipus de dades....12
4.1.7 Expressions i operadors....13
4.1.7 Expressions i operadors....13
4.2 Tipus de dades: declaració de variables i instanciació....14
4.3 Instruccions acceptades....16
4.3.1 Instrucció d'assignació....16
4.3.2 Instrucció res....16
4.3.3 Instruccions alternatives: si i cas....17
4.3.4 Instruccions iteratives: mentre i per....18
4.3.5 Instrucció crear....19
4.3.6 Instrucció destruir....20
4.3.7 Instrucció retorna....20
4.3.8 Instruccions d'E/S....20
4.4 Esquema d'algorisme....22
4.5 Classes....23
4.5.1 Encapsulament....24
4.5.2 Atributs i mètodes....25
4.5.3 Atributs de classe i mètodes de classe....25
4.5.4 Atributs i funcions privades....27
4.5.5 Constructors i destructors....28
4.5.6 Sobrecàrrega....29
4.5.7 Herència....29
4.5.8 Les referències jo i super....30
4.6 Taules....31
4.7 Ús de les classes....32
4.8 Format del fitxer font....33
4.9 Restriccions del traductor....34

1. Introducció

El CPC és un traductor de pseudocodi a **Java**. Permet escriure programes en pseudocodi i executar-los, prèvia traducció i compilació del fitxer **Java**. Podem pensar que el CPC és com una interfície entre el llenguatge pseudocodi i el propi llenguatge **Java**.

El CPC és una eina que permet implementar ràpidament i fàcilment algorismes, sense tenir que preocupar-nos de la rigidesa dels llenguatges de programació actuals. El llenguatge que accepta és el pseudocodi 03, una modificació del pseudocodi original per permetre la programació orientada a objectes. Amb el traductor es pot escriure programes que solucionin problemes de mida mitjana, tal com es poden veure en els exemples proporcionats. Per tant, és una eina bastant potent i versàtil per que es tingui en compte a l'hora d'implementar algorismes i estructures de dades experimentals o amb fins docents.

Aquest petit manual d'usuari proporcionarà tots els coneixements per tal de poder extreure tota la utilitat a l'eina, així com servir de manuals de referència si es vol utilitzar el CPC de forma continuada.

2. Instal·lació

El traductor necessita les següents dependències:

- Per executar el CPC i generar els fitxers **Java**:

    Màquina virtual de Java 2 instal·lada (JRE 1.2 o superior).

- Per executar el CPC, generar i compilar els fitxers **Java**:

    Java Development Kit 1.2 o superior (JDK 1.2 o superior).

És interessant considerar satisfer la segona opció, ja que amb la primera els fitxers generats per el traductor no es podran compilar, per la qual cosa la utilitat del CPC queda molt reduïda.

Per instal·lar el programa només cal descomprimir el fitxer **zip** en un directori qualsevol del disc dur creat expressament per contenir el traductor. Per a poder executar el CPC des de qualsevol directori del disc dur, només cal afegir la ruta a les variables d'entorn PATH i CLASSPATH del sistema. A continuació es detallarà com es fa en els sistemes operatius més estesos, en altres sistemes només s'haurà d'operar de manera similar.

2.1 Instal·lació de les variables d'entorn en Linux

Les variables d'entorn dels sistemes Linux solen estar en el fitxer “/etc/profile”. Només cal afegir la ruta on hagem descomprimit el CPC a la variable PATH i a la variable CLASSPATH, i després fer un *export*. El separador entre directoris és el dos punts “:”. Per exemple si suposem que hem descomprimit el CPC en el directori “/usr/local/cpc”, hauríem de modificar el fitxer tal com es mostra (NOTA: s'ha de tenir permisos de **root**):

```
export PATH=$PATH:/usr/local/cpc
export CLASSPATH=$CLASSPATH:/usr/local/cpc
```

Si volem només afegir-li a un usuari, o simplement no tenim permisos de **root** a la màquina on ho estem instal·lant, només cal afegir les dues línies anteriors al fitxer que conté la configuració d'inici del *shell* de l'usuari. En el cas del *bash* (el més comú) és el ".bash_profile" que es troba en el directori de inici (o directori *home*) de cada usuari.

2.2 Instal·lació de les variables d'entorn en Windows 95/98/ME

El Windows 98 utilitza el mateix sistema que el MS-DOS per tal de definir les variables d'entorn. Aquestes estan en el fitxer "autoexec.bat" al directori arrel del disc dur.

Normalment el llistat del fitxer donaria una sortida semblant a aquesta:

```
SET BLASTER=A220 I7 D1 H7 P320 T6
SET SBPCI=C:\ARCHIV~1\CREATIVE\AUDIO\DOSDRV
mode con codepage prepare=((850) C:\WINDOWS\COMMAND\ega.cpi)
mode con codepage select=850
keyb sp,,C:\WINDOWS\COMMAND\keyboard.sys
SET PATH=C:\j2sdk1.4.0_02\bin;C:\MinGW\bin
SET CLASSPATH=.
```

La instrucció *SET* especifica que és una variable d'entorn i que s'ha de emmagatzemar en memòria durant tota la sessió. Ara, només cal afegir la ruta del directori on tinguem instal·lat el CPC, tot tenint en compte que el separador de directoris és el punt i coma ";". Si haguéssim instal·lat el traductor en el directori "C:\cpc", les línies modificades tindrien un aspecte semblant a aquest:

```
SET PATH=C:\j2sdk1.4.0_02\bin;C:\MinGW\bin;C:\cpc
SET CLASSPATH=.;C:\cpc
```

2.3 Instal·lació de les variables d'entorn en Windows 2000/XP

Per el Windows 2000 es poden modificar les variables d'entorn d'un usuari, o si es té privilegis d'**Administrador**, les de tot el sistema. Per fer això, cal anar al *panell de control*, i

seleccionar la icona *sistema*. Acte seguit s'ha d'anar a la pestanya *Avançat*. Un cop aquí, seleccionant el botó *Variables d'entorn* s'accedirà a la finestra de configuració.

Si es té privilegis d'**Administrador**, és convenient canviar el PATH i el CLASSPATH de les variables del sistema, de manera que tots els usuaris les tinguessin canviades automàticament. Però en cas que no es tingui privilegis, es poden canviar les del propi usuari. De qualsevol de les maneres, els passos a seguir són els mateixos.

Es selecciona la variable corresponent, per exemple PATH, i es prem el botó *editar*. Tot recordant que el separador de directoris és el punt i coma ";", inserim a la llista de directoris la ruta on tinguem instal·lat el CPC. Si suposem que l'hem instal·lat a "E:\cpc", una possible variable PATH ens podria quedar semblant a:

```
%SystemRoot%\system32;%SystemRoot%;%SystemRoot%\System32\Wbem;E:\cpc
```

Després s'actua igual amb la variable CLASSPATH, i ja tindríem el sistema preparat.

3. Ús del traductor CPC

El traductor CPC està generat en **Java** i per tant necessita la màquina virtual **Java** per funcionar. Per poder ser executat cal ser cridat des de la línia de comandes com qualsevol classe **Java**:

```
java cpc
```

Si no li escrivim res més, mostra una petita ajuda en pantalla del format d'entrada que rep i de les opcions que té. Per a poder traduir un fitxer de pseudocodi, se li ha de passar el seu nom com a paràmetre, com per exemple:

```
java cpc holamon.pse
```

Ara el traductor tractaria el fitxer "holamon.pse" i en cas que aquest sigués correcte, generaria un fitxer de sortida anomenat "holamon.java".

3.1 Flux d'informació

Com ja s'ha explicat, el CPC rep com entrada un fitxer en text pla que contingui un algorisme o classe en pseudocodi i treu com a sortida un fitxer **Java**. Aquest fitxer **Java** pot ésser compilat directament amb qualsevol compilador estàndard de **Java**, com el de *Sun Microsystems*. Finalment, s'obté un fitxer de binari en *bytecode* amb l'extensió ".class" que s'executa amb la màquina virtual de **Java** instal·lada. En la següent figura es pot observar el flux d'informació:

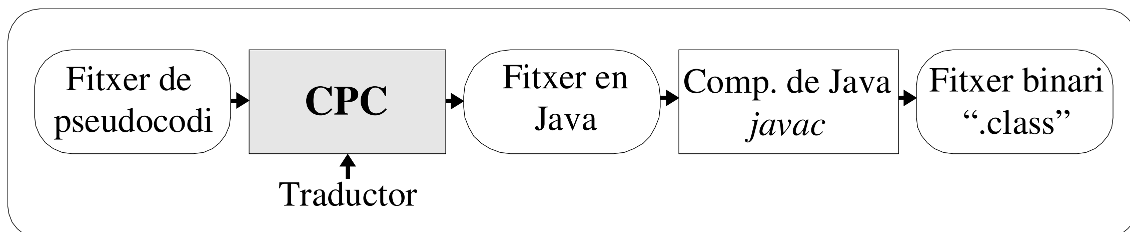

Figura 3.1 Flux d'informació del traductor

Com es pot observar, el traductor és pot considerar com una interfície entre el fitxer de pseudocodi i el **Java**, però té molta més potència i versatilitat, ja que fa totes les comprovacions que realitza qualsevol compilador estàndard.

3.2 Opcions de la línia de comandes

El CPC pot rebre opcions quan es crida des de la línia de comandes. El format complet de la crida del programa és:

```
java cpc [opcions] fitxer
```

On opcions poden ser:

-j Compila el fitxer traduït amb el compilador *javac*.
-v Imprimeix missatges sobre el que fa el traductor.

La primera opció ajuda a automatitzar el procés de manera que, si el fitxer de pseudocodi és correcte, compila automàticament el fitxer **Java** generat amb el compilador *javac*, el compilador estàndard de **Java** proporcionat per *Sun Microsystems*. Així, si per exemple executem la comanda:

```
java cpc -j holamon.pse
```

Obtindrem com a sortida el fitxer "holamon.java" i també el fitxer "holamon.class", aquest últim executable directament. D'aquesta manera ens estalvia de fer la compilació del fitxer **Java** manualment.

L'opció **-v** és la opció *verbose*, que imprimeix per pantalla missatges sobre les tasques que duu a terme el traductor. És molt útil quan es tradueix un fitxer de pseudocodi que fa referències a altres fitxers, ja que es pot observar en quin ordre són tractats per el traductor.

4. Pseudocodi 03

El traductor accepta el llenguatge pseudocodi 03. Aquest està basat en el pseudocodi habitual amb algunes modificacions i afegits, per tal d'explotar el model d'orientació a objectes. Existeixen les instruccions repetitives, condicionals i l'assignació. A més s'afegeix el concepte de classe, polimorfisme, herència i encapsulació, que són els pilars del paradigma de programació amb objectes. La programació s'esdevé més centrada en el disseny de classes i estructures de dades, i no pas en els algorismes, que és on estava centrat el pseudocodi anterior.

Aquest capítol és una referència al llenguatge per tal de tenir una ajuda a l'hora de programar en pseudocodi i treure tot el partit a l'eina desenvolupada. Cal conèixer bé el llenguatge que tracta el traductor per entendre com està definit aquest i com actua. Per tant, es definirà la gramàtica del pseudocodi d'una manera simple, directa i entenedora.

Es farà una visió bastant ràpida i no s'entrarà en gaires de detalls, ja que se suposa uns mínims de coneixement de programació, i no s'explicaran conceptes bàsics com per exemple definir que és una variable. En canvi, s'explicarà com es declara i s'utilitza en el pseudocodi.

Finalment, cal fer notar que el pseudocodi no és sensible a majúscules i minúscules (no és *case sensitive*). Això significa que una variable tota en majúscules i una altra variable tota en minúscules són interpretades com la mateixa variable. Això fa més fàcil la programació per els novells, que solen patir molt per la característica de distincions de majúscules i minúscules del **C++** i del **Java**.

4.1 Conceptes bàsics

A continuació s'enumeraren una sèrie de conceptes bàsics en la definició d'un llenguatge de programació.

- **Comentaris**: línies escrites per el programador que no són processades per el compilador i tenen efecte de documentació.

- **Sentències**: és una línia de programa, normalment anomenada també instrucció.
- **Blocs de codi**: és un grup de sentències que formen una unitat.
- **Paraules reservades**: Paraules que s'han predefinit en el llenguatge pseudocodi 03 i no poden ser utilitzades com a identificadors.
- **Identificadors**: són els noms que es donen a les classes, variables, funcions,... Els identificadors poden ser qualsevol cadena alfanumèrica formada per lletres i números, amb la restricció que el primer caràcter ha d'ésser una lletra o el caràcter guió-baix: "_".
- **Literals**: valors constants que s'escriuen diferent depenen del tipus de dades que representin. Per exemple, és diferent **123** de "123", ja que el primer simbolitza un número enter, mentre que el segon correspon a una cadena.
- **Expressions**: combinació de termes que s'avaluen a un únic valor, com per exemple una suma, o una comparació lògica.
- **Operadors**: serveixen d'unió entre factors i expressions. Poden ser matemàtics com la suma, resta,... o també lògics, com igual, major que,...

Passem a explicar-los més detalladament, i com són tractats en el pseudocodi 03.

4.1.1 Comentaris

Els comentaris són tractats d'igual forma que el **C++**. El primer tipus de comentari és el multilínia, comença amb "/*" i acaba amb "*/".

Per exemple:

```
/* Això és un comentari
de més d'una línia de text */
```

El segon tipus de comentari és aquell que comença amb la marca de comentari "//" i finalitza al final de la línia del fitxer (retorn de carro).

Per exemple:

```
// Aquest comentari arriba només fins a final de línia
```

4.1.2 Sentències

Una sentència és una línia de codi. S'utilitza el retorn de carro com element de puntuació per indicar el final de la instrucció. Es pot utilitzar el punt i coma per escriure més d'una sentència en la mateixa línia.

Per exemple:

```
a <- 2+3; a <- 56-2
```

És el mateix que:

```
a <- 2+3
a <- 56-2
```

Els espais entre les parts d'una sentència poden consistir en qualsevol número d'espais en blanc o tabulacions.

4.1.3 Blocs de codi

Les sentències es solen agrupar dintre de blocs de codi. Els blocs de codi solen estar determinats per un nom que l'identifica al principi, i al final amb una “f” seguida del mateix nom.

Per exemple, el bloc de declaració de variables seria:

```
var
      // Aquí anirien les declaracions de variables
fvar
```

4.1.4 Paraules reservades

Una paraula reservada és una paraula que posseeix un significat especial per el traductor, tal com el nom d'un tipus de dades, una instrucció, entre d'altres.

En la pàgina següent hi ha la taula 3.1 en la que es mostra la llista de paraules reservades.

Paraules reservades			
algorisme	escriure	fmetode	metode_classe
altrament	escriureln	fmetode_classe	metodes
atributs	falgorisme	fmetodes	metodes_classe
atributs_classe	fals	fmetodes_classe	mod
boolea	fatributs	fper	no
caracter	fatributs_classe	fsi	null
cas	fcas	funcio	o
cert	fclasse	fvar	pas
classe	fconst	hereta	per
const	fconstructor	i	real
constructor	fdestructor	implementacio	res
crear	fer	interficie	retorna
de	ffuncio	jo	si
destructor	fimplementacio	llavors	super
destruir	fins	llegir	taula
div	finterficie	mentre	var
enter	fmentre	metode	

Taula 3.1 Paraules reservades del pseudocodi 03

4.1.5 Identificadors

Un identificador és el nom que se li dóna a una variable, classe o funció. Es pot escollir l'identificador que es desitgi, mentre que comenci per una lletra o un guió baix "_" seguit d'una combinació de lletres i números, i han de ser diferents a qualsevol paraula reservada. Existeix un conveni en la nomenclatura dels identificadors, que es segueix en la memòria i en el codi font del projecte, tot i que no és obligatori (vegeu 1.4 Convencions).

4.1.6 Literals i tipus de dades

Si els identificadors són el símbol al que fem referència a un valor, un literal és un valor concret com 39 ó "numeret". Cada literal correspon a un tipus de dades, excepte el literal cadena, que només existeix com a literal, però no es pot declarar cap variable de tipus cadena.

Tipus de dades	Literal	Exemples
enter	Dígits.	10 239
real	Dígits amb un punt decimal.	3012.0 9.382
boolea	**cert** o **fals**	cert fals
caracter	Qualsevol caràcter ASCII entre cometes simples.	'a' 'B'
cadena	Una seqüència de caràcter o seqüència d'escapament entre cometes.	“Hola món\n” “Més proves”

Taula 3.2 Format per els literals de cada tipus de dades

Com ja hem dit, el tipus cadena en sí no existeix, només existeix el literal. Ha sigut introduir per simplificar a l'usuari l'escriptura de missatges per pantalla, ja que és evident que escriure caràcter a caràcter una frase per poder ésser impresa és massa laboriós. No obstant, el tipus cadena o *string* es pot implementar fàcilment en una classe.

4.1.7 Expressions i operadors

Les expressions són combinacions de variables, literals i crides a funcions que són avaluades amb un determinat tipus de dades, que pot ser un número, una cadena o qualsevol altre classe o tipus de dades. Normalment les expressions les trobem en el costat dret en una assignació. Les expressions més simples són variables aïllades o literals, i es poden complicar tant com vulguem.

Exemples d'expressions:

```
33
'a'
734*(97-(82+1)/2)
```

Els operadors són símbols que combinen les expressions simples per formar expressions més complexes o apliquen una transformació a una variable. Els operadors poden ser binaris, si són aplicats a dues expressions, i unaris si només són aplicats a una. Per exemple, la multiplicació és un

operador binari, ja que sempre es multipliquen dos números, mentre que l'operador lògic **no** és un operador unari, aplicat només a una expressió de tipus lògica. Quan s'utilitza varis operadors en una expressió, és important saber en quin ordre s'avaluaran. Com en matemàtiques, existeix una *prioritat d'operadors*, i si els operadors tenen la mateixa prioritat, s'avaluen d'esquerre a dreta. Cal recordar que es pot fer ús dels parèntesis per agrupar expressions i definir d'aquesta manera en quin ordre s'han d'avaluar. D'operadors hi ha de tres tipus: operadors aritmètics, operadors lògics i l'operador assignació.

Els operadors matemàtics són els utilitzats en operacions elementals, a la següent taula es pot comprovar els operadors i el seu ordre de preferència.

Operador	Descripció	Preferència
+,-	Més unari,menys unari	1 (major preferència)
*,/	Multiplicació,divisió real	2
mod	Resta (mòdul de la divisió)	2
div	Divisió entera	2
+,-	Suma,resta	3

Taula 3.3 Operadors aritmètics

Els operadors lògics comparen dues expressions per determinar si són iguals, si una és més gran que l'altre,... És aconsellable l'ús de parèntesis per tal de no equivocar-se a fer ús de l'expressió.

Operador	Descripció	Preferència
<,<=,>,>=	Verifica magnituds relatives	4
=,<>	Verifica la igualtat i la desigualtat.	5
i	I condicional	6
o	O condicional	6

Taula 3.4 Operadors lògics/relacionals

4.2 Tipus de dades: declaració de variables i instanciació

En pseudocodi 03 existeixen dos grans tipus de dades: elementals, o també anomenats bàsics, i els de referència. Els elementals són aquells integrats en el llenguatge, i que ja hem vist

anteriorment: **enter**, **real**, **caràcter**, **booleà**. Els tipus de referència són les variables declarades com a qualsevol tipus definit per l'usuari en una classe. Són variables que referencien la instància de la classe (que ha de ser creada posteriorment). Dintre del tipus referència també es troben les taules, que ja s'explicaran més endavant.

Per la declarar una variables s'utilitza la notació del nom de la variable, seguida per dos punts i el tipus. El nom de la variable ha d'ésser un identificador vàlid. En una mateixa línia es poden declarar més d'una variable del mateix tipus, tot separant-les per comes.

Per exemple:

```
variable1: enter                    // Tipus elemental (enter)
var1:enter                          // Tipus elemental (enter)
variable2,variable3: Classe1        // Referència a una classe (Classe1)
```

Les variables objecte, les que són referència a una classe, tenen un tracte una mica especial. A diferència dels tipus bàsics, no poden ser utilitzades directament un cop declarades, sinó que cal *instanciar-les*. Instanciar una classe significa crear-la: es reserva l'espai de memòria que necessita, i pot ser utilitzada per el programa. No es pot utilitzar una classe sense haver-la instanciat prèviament, ja que la referència és nul·la i es produirà un error en temps d'execució. La instanciació d'una variable es realitza amb la paraula reservada **crear** seguida del tipus que es vol instanciar i de parèntesis i opcionalment paràmetres, com si fos una crida a una funció.

Per exemple:

```
variable2 <- crear Classe1()
```

Les constants són dades que no varien en el temps, són dades constants (valgui la redundància). Per tant, una vegada que s'ha declarat la constant i el seu valor, aquest persisteix immutable fins al final del programa. Per definit una constant, es fa servir el següent esquema:

NOM_CONSTANT : tipus_elemental = valor

Les constant només es poden declarar de tipus bàsic (o elemental), i, lògicament, el valor que prenen ha de concordar amb el tipus de constant declarada.

Per exemple:

```
PI: real = 3.1416      // Correcte.
JULIOL: enter = 6.0  // Això provocaria un error, ja que la constant és de
                       tipus entera, i el valor és de tipus real.
```

4.3 Instruccions acceptades

A continuació s'exposaran les instruccions que formen part de la definició del pseudocodi 03, fent notar detingudament quin és el seu format d'ús. Com anteriorment s'havia comentat, el salt de línia és el separador entre les instruccions, per tant, s'ometrà en aquelles instruccions que el necessitin en el seu format, ja que es podrà apreciar visualment en la seva definició.

4.3.1 Instrucció d'assignació

L'operador d'assignació és la fletxa dirigida a l'esquerra (“<-”). La instrucció d'assignació es composa de la part dreta, que és el valor que es vol assignar, i de la part esquerra, que és el destí final del que s'assigna. A la part esquerra hi ha d'haver una variable, que permeti poder emmagatzemar un valor, mentre que la part dreta pot ser qualsevol expressió, això si, del mateix tipus que la variable a la qual es vol assignar.

Per exemple:

```
variable1 <- 3 + 527
variable2 <- variable3
```

L'assignació múltiple present en el pseudocodi 03 no s'ha pogut implementar, a causa de les limitacions de temps i del propi llenguatge **Java**.

4.3.2 Instrucció *res*

No fa res, serveix per simbolitzar que no cal executar cap instrucció arribat a aquell punt.

4.3.3 Instruccions alternatives: *si* i *cas*

Les instruccions alternatives són molt importants ja que permeten desviar el flux d'execució del programa, podent obtenir algorismes molt més interessants, ja que poden tenir diferent resposta a diferents entrades.

El **si** permet avaluar una condició, i executar un bloc d'instruccions si aquesta es compleix, i opcionalment executar un altre bloc d'instruccions diferents si no es compleix. Es pot escriure en dos variants: una línia o multi-línia.

- Una línia:

  si condició **llavors** instrucció **[altrament** instrucció**] fsi**

- Multi-línia:

  si condició **llavors**
  instruccions
  [**altrament**
  instruccions]
  fsi

En el **si** només es pot executar un dels dos codis, ja que o bé es compleix la condició, o bé no es compleix (condició booleana). Com es pot comprovar, la línia de l'**altrament** és opcional, i si no hi és i no és compleix la condició, s'executarà **res**.

Posem un exemple per clarificar-ho:

```
si (a1 < 2) llavors
      b1 <- cert     // Això només passa si a1 < 2
altrament
      si (a2=3) llavors b1<-cert fsi
      // El programa sempre arriba aquí si a1>=2
      // Posarà b1 a cert si es compleix també que a2=3
fsi
```

La instrucció **cas** és semblant a la **si**, ja que ambdues són alternatives, però amb la diferència que la primera permet avaluar múltiples condicions alhora. El format és el següent:

cas
[] condició_1 -> instruccions

[] condició_2 -> instruccions
...
[] condició_n -> instruccions
fcas

Al començament de cada condició hi ha un obrir (**[**) i tancar (**]**) corxet, i són obligatòries, ja que es necessita un delimitador per la condició (sinó la gramàtica que defineix el llenguatge esdevindria ambigua). En el **cas** s'ha d'especificar com a mínim una condició.

Per exemple:

```
// Traducció a cas de l'exemple posat per la instrucció si
cas
[] (a1 < 2) -> b1<-cert
[] (a1 >= 2) -> cas
                [] (a2=3) -> b1<-cert
                fcas
fcas
```

4.3.4 Instruccions iteratives: *mentre* i *per*

Les instruccions iteratives permeten escriure blocs de codi que seran repetits una sèrie de vegades. Això dóna molta potència a l'hora de crear algorismes molt complexos.

La instrucció **mentre** conté una condició, la qual s'avalua la primera vegada abans d'executar les instruccions del bloc, i a cada pas de la iteració es torna a avaluar. Això es va repetint fins que la condició esdevé falsa. Acte seguit el programa salta a la instrucció següent després del **mentre**. La instrucció **mentre** també es presenta en dos variants: una línia o multi-línia.

- Una línia:
  mentre condició **fer** instrucció **fmentre**
- Multi-línia:
  mentre condició **fer**
  instruccions
  fmentre

Per exemple:

```
// Iteració cap endavant
var<-1
mentre (var1 < 10) fer
```

```
      variable1 <- var1
      var1 <- var1 + 1
fmentre
```

En canvi, la instrucció **fer** presenta una iteració no condicional, en la qual el nombre de iteracions que es realitzaran sobre el codi ve definit en la pròpia instrucció. La instrucció **fer** també es presenta en dos variants: una línia o multi-línia, vegem-ne el format:

- Una línia:
  per variable **<-** expressió **fins** expressió **[pas** expressió**] fer** instrucció **fper**
- Multi-línia:
  per variable **<-** expressió **fins** expressió **[pas** expressió**] fer**
  instruccions
  fper

La variable és la que controla el bucle. S'inicialitza amb el valor de la primera expressió, que ha de ser entera, i el bucle itera mentre el valor de la variable sigui més petit que el de la segona expressió, que també ha de ser entera. Com es pot observar, el **pas** és opcional, que significa els increments que es sumen a la variable en cada iteració. L'expressió del **pas** pot ser positiva o negativa, però sempre un nombre enter. Si és negativa, a cada volta de la iteració disminueix el valor de la variable controladora. Si no s'especifica cap **pas**, es pren com a defecte *+1*, i per tant, a cada iteració, la variable de control s'incrementaria en una unitat.

Vegem un exemple:

```
// Iteració cap enrere
per var1<-10 fins 1 pas -1 fer
      variable1 <- var1
fper
// Iteració cap endavant
per var1<-1 fins 10 fer
      variable1 <- var1
fper
```

4.3.5 Instrucció *crear*

La instrucció **crear**, retorna una instància d'un objecte (classe), de tal manera que assigna la memòria que necessita l'objecte.

El seu ús és el següent:

nomVariable **<- crear** tipus(paràmetres)

paràmetres: són paràmetres que pot necessitar-se per instanciar-se el tipus, pot no necessitar-ne cap. La instrucció **crear** crida al constructor de la classe que es vol instanciar, i per tant, els paràmetres han de coincidir amb la definició d'algun dels constructors de la classe (per a més informació referent als constructors consultar 3.6.5 Constructors i destructors). Qualsevol variable de tipus objecte cal que sigui instanciada abans d'ésser utilitzada. Una variable no instanciada té una referència nul·la, que es representa amb la paraula reservada **null**.

Exemple:

```
// crear sense cap paràmetre
variable2 <- crear Classe1()
// crear amb paràmetres
variable2 <- crear Classe1(23,'a')
```

4.3.6 Instrucció *destruir*

Aquesta instrucció destrueix l'objecte, és a dir, allibera la memòria reservada per l'objecte en la instrucció **crear**. Explicat més detingudament, aquesta instrucció crida al destructor de la classe, que s'encarregarà d'alliberar la memòria reservada per el constructor. Es pot dir que és justament la inversa de la instrucció **crear**.

4.3.7 Instrucció *retorna*

Aquesta instrucció fa que es finalitzi l'execució d'una funció o mètode i es retorni el valor.

4.3.8 Instruccions d'E/S

Ja que el pseudocodi ha d'ésser el més senzill possible, s'ha optat de proporcionar al propi llenguatge d'instruccions bàsiques d'entrada i sortides de dades: **escriure**, **escriureln** i **llegir**.

- **escriure**(*expressió*)

Escriu per la sortida estàndard (normalment la pantalla), l'expressió o variable. Si la variable és de tipus elemental, escriu directament el seu valor, si és un objecte, escriu el que retorna directament la funció: **toString**(). Nota: tota classe creada per l'usuari té una funció **toString()** per defecte que pot ser sobreescrita per l'usuari.

- **escriureln**(*expressió*)

Es comporta d'igual manera que la funció **escriure**, però afegeix automàticament un salt de línia al final del que s'escriu, d'aquesta manera, després d'haver escrit l'expressió, es salta a la següent línia.

A **escriure** i **escriureln** se li poden passar literals de tipus cadena, de tal forma que es pot escriure una frase per pantalla. A més, es poden passar diversos caràcters d'escapament. Les seqüències d'escapament són les mateixes que les que suporta el **Java**, a la taula 3.5 es presenten les més importants.

Literal	Descripció
'\b'	Retrocés
'\t'	Tabulador
'\n'	Avanç de línia
'\f'	Avanç de pàgina
'\r'	Retorn de carro
'\\'	Barra invertida

Taula 3.5 Seqüències d'escapament especials

- **llegir**()

Espera que l'usuari entri una dada i la retorna, de tal manera que pot ésser assignat a una variable. Si l'usuari entra un valor d'un tipus que no correspon amb el que pertoca a l'assignació, es llançarà una *excepció* i el programa finalitzarà. La instrucció només retorna dades de tipus elemental, si es fa una assignació a un altre tipus generarà un error en temps de compilació.

4.4 Esquema d'algorisme

Els **algorismes** simbolitzen el punt d'entrada al programa, la funció principal. És com la funció *main* del **C/C++** o el mètode estàtic *main* del **Java**. Degut això, dintre d'un algorisme no es permet la declaració de funcions i/o mètodes de cap tipus. Això és degut a que en les tècniques de programació orientada a objectes, l'algoritme principal serveix només per instanciar els objectes bàsics i eventualment d'entrada/sortida d'algun valor de l'usuari. Per tant, tota la feina corre a càrrec de les classes.

L'estructura d'algorisme és la següent:

algorisme nomAlgorisme
const
NOM_CONSTANT : tipus_elemental = valor
fconst
var
nomVariable: tipus
fvar

instruccions
falgorisme

Les parts de declaració de constants i de variables són opcionals. S'ha mostrat també com s'introdueixen la declaració de constants i variables, explicades en el capítol anterior, dintre d'un algorisme, i amb el que s'ha vist fins ara, es poden fer algorismes senzills.

Com ja s'ha vist, dintre els operadors aritmètics no n'hi ha cap que simbolitzi la potència. Mostrarem un programa exemple, que demanarà la base i l'exponent a l'usuari, i calcularà la potència, és a dir, elevarà la base a l'exponent especificat.

Programa potència:

```
algorisme potencia
var
      base,exp,result,num: enter
fvar
```

```
        escriure("Escriu la base: ")
        base <- llegir()
        escriure("Escriu l'exponent: ")
        exp <- llegir()

        si (exp=0) llavors
                  // Qualsevol nombre elevat a zero és 1
                  result <- 1
        altrament
                  // Inicialitzarem el resultat amb la base
                  result <- base
                  // Farem tantes multiplicacions com indiqui l'exponent
                  per num<-2 fins exp fer
                            result <- result * base
                  fper
        fsi

        escriure("El resultat: ")
        escriureln(result)

falgorisme
```

Com s'ha pogut observar, en el programa s'ha utilitzat instruccions de molts tipus: iteratives, condicionals, d'E/S,... variables i constants. I a més, és plenament funcional. Amb aquest esquema es poden resoldre problemes petits, però no n'hi ha prou. Per resoldre problemes una mica “seriosos” ens cal alguna cosa més, que ens ajudi a estructurar i desenvolupar programes molt llargs. Necessitem fer ús de les classes.

4.5 Classes

El pseudocodi nou ha d'afavorir la bona programació orientada a objectes i ha de proporcionar un llenguatge senzill per poder crear aquests objectes. Per designar-los farem servir les classes, que és el concepte més estès gràcies al **C++** i al **Java**. Una classe és la definició d'un objecte, que conté unes dades que el defineixen, anomenades atributs, i unes accions que permeten que actuï i es comuniqui amb els altres objectes, anomenades mètodes. Com que la classe és només una definició, quan es declara una variable de tipus objecte, cal instanciar-la per poder ésser utilitzada, com ja s'ha vist anteriorment.

Per definir una classe s'utilitza la paraula reservada **classe** seguida d'un identificador, i al final del bloc: **fclasse**.

Per tant, una classe restaria d'aquesta forma:

```
classe PrimeraClasse
      ...
fclasse
```

4.5.1 Encapsulament

L'encapsulament és una propietat intrínseca de l'orientació a objectes que consisteix en establir unes variables, funcions i atributs privats, que només poden ésser accedits des de dintre la classe, i uns mètodes i atributs públics que poden ser accedits des de fora de la classe. Aquests últims permeten alterar-ne les propietats i comportament. Per a poder separar les dades públiques de les privades, hi ha el bloc **interficie** i el bloc **implementacio**.

A la interfície es declara tot allò que és públic, i per tant pot ser accedit des de fora la classe, és el que defineix com es relaciona amb les altres classes. En aquesta secció només hi ha declaracions, però no hi ha codi de cap tipus. En canvi en el bloc d'implementació hi ha el codi, o implementació, dels mètodes definits en la secció interfície, i a més també hi ha altres funcions privades, així com variables internes a la classe, declarades dintre un bloc **var-fvar**.

Així, un primer esquema d'una classe seria:

```
classe PrimeraClasse
      interficie
                 // Aquí va tot allò que sigui públic i es vulgui
                 // que sigui visible a altres classes externes.
      finterficie
      implementacio
                 // Aquí va tot el codi de la classe, així com dades que
                 // no volem mostrar o que siguin accedides des de fora.
      fimplementacio
fclasse
```

4.5.2 Atributs i mètodes

Els atributs són variables públiques, i per tant, visibles des de fora la classe. Podem definir com a atribut qualsevol tipus de variable. Els atributs sempre estan definits en el bloc **atributs** dintre la **interficie**. Al ser variables, els atributs es declaren dintre un bloc **var**.

Els mètodes són el que s'anomenava accions i funcions en el pseudocodi habitual. Els mètodes poden retornar o no un valor. A la secció **interficie** trobem declarades les capçaleres dels mètodes dintre el bloc **metodes**. La capçalera d'un mètode és:

metode identificador **(***paràmetres_formals)* [**retorna** identificador: tipus]

Paràmetres formals indica els paràmetres que s'han de passar al cridar al mètode. Han d'estar definits com una llista de: identificador **:** tipus. Si n'hi ha més d'un, han d'estar separats per comes.

El **retorna**, és opcional. Si hi és, especifica que el mètode es comporta com una funció, i retorna un valor del tipus definit. L'identificador declarat en el **retorna** és el paràmetre de retorn.

El codi del mètode està dintre el bloc **implementacio**: només cal tornar a escriure la capçalera del mètode, escriure el codi, i acabar amb un **fmetode**. L'identificador de retorn declarat en el **retorna** es declara automàticament en la implementació del mètode. Es pot utilitzar directament, no cal declarar-ho explícitament.

Els atributs i mètodes sempre estan lligats a una instància, i per tant, cada instància tindrà valors diferents per els atributs i els seus mètodes retornaran valors diferents, depenent de les dades internes que tingui emmagatzemades.

4.5.3 Atributs de classe i mètodes de classe

Els atributs i mètodes pertanyen a una instància, però ens pot interessar que hi hagi dades que pertanyin a la pròpia classe, com per exemple, el número d'instàncies creades, diferents constants (com ara PI, nombre E,... per una classe matemàtica), entre d'altres. Les variables, constants i

mètodes que vulguem que siguin de la classe, es declaren en el bloc **atributs_classe** per les constants i variables, i en el bloc **metodes_classe** per els mètodes. La declaració d'un mètode de classe es fa de la mateixa manera que la d'un mètode d'instància, amb la diferència que s'ha d'utilitzar la paraula reservada **metode_classe**, enlloc de **metode**. La implementació del mètode de classe és també igual a la del mètode, canviant les paraules reservades a **metode_classe** i **fmetode_classe**, per començar i acabar el bloc respectivament.

Com es pot comprovar, no s'ha comentat que es puguin declarar constants en un bloc **atributs** d'una instància. El propi llenguatge no ho admet, ja que una constant que pertanyi a una instància no té sentit, les constants són pròpies d'una classe.

En els mètodes de classe, només poden accedir als atributs de classe, i no pas als atributs d'instància, ja que el mètode de la classe no sabria quin valor d'instància agafar per l'atribut.

Resumint, fins ara, la declaració completa d'una classe podria ser:

```
classe PrimeraClasse
      interficie
                // Aquí va tot allò que sigui públic i es vulgui
                // que sigui visible a altres classes externes.

      atributs_classe
                const
                          // Constants de la classe.
                fconst
                var
                          // Variables que representen els atributs
                          // d'una classe.
                fvar
      fatributs_classe
      atributs
                var
                          // Aquí estan definits els atributs
                          // que pertanyen a cada instància.
                fvar
      fatributs
      metodes_classe
                          // Els mètodes pertanyents a la classe.
                metode_classe sumar_instancia() retorna e: enter
      fmetodes_classe
```

```
	metodes
		// Els mètodes pertanyents a cada instància
		metode fer_alguna_cosa(num: enter)
	fmetodes
	finterficie
	implementacio
		// Aquí va tot el codi de la classe, així com dades que
		// no volem mostrar o que siguin accedides des de fora.
		metode_classe sumar_instancia() retorna e: enter
			// Implementació del mètode de classe
		fmetode_classe
		metode fer_alguna_cosa(num: enter)
			// Implementació del mètode d'intància
		fmetode
	fimplementacio
fclasse
```

L'ordre de la declaració de la interfície: atributs de classe, atributs, mètodes de classe i mètodes no es pot alterar, tot i que tots són opcionals.

4.5.4 Atributs i funcions privades

Per poder permetre l'encapsulament en els objectes, hem de proporcionar un mitjà per tal de definir atributs i funcions que no siguin accessibles des d'objectes externs. D'aquesta manera proporcionem coherència a les dades, ja que sabem que cap element extern hi té accés, i per tant no poden ésser modificades per ningú més que la pròpia classe.

Els atributs o variables privades s'han de declarar en el seu corresponent bloc **var** dintre el bloc d'implementació, just abans de la implementació del codi dels mètodes i funcions. Les funcions privades estan declarades amb la paraula **funcio** i només les trobem en el bloc implementació i no en a la interfície (sinó no serien privades).

Per exemple, declararem un atribut privat de tipus enter, i una funció privada que retorna el número multiplicat pel paràmetre passat:

```
implementacio
	var
		num: enter
	fvar
```

```
        funcio mult2(factor: enter) retorna e: enter
                  e <- num * factor
                  retorna e
        ffuncio
    fimplementacio
```

4.5.5 Constructors i destructors

En les classes existeixen dos mètodes especials anomenats: **constructor** i **destructor**. El **constructor** es executat automàticament quan es crea una instància de la classe, és a dir, quan s'utilitza la instrucció **crear**. Serveix normalment per inicialitzar algunes variables o reservar memòria. El **destructor** es executat automàticament quan es crida a **destruir**, i normalment s'executa codi per alliberar la memòria reservada en el constructor, o en altres mètodes executats per la classe. Els constructors estan declarats en el bloc d'interfície encara que no pot ser cridat directament des de fora la classe, ja que, com ja s'ha dit, es cridat automàticament per la instrucció **crear**. En canvi, els destructors només apareixen en el bloc d'implementació.

Cal dir que és millor no escriure codi que sigui essencial executar-se al destruir la instància de la classe, com un alliberament dels recursos. Això és així per que el llenguatge destí al que es tradueix, el **Java**, no posseeix estrictament un destructor, sinó que defineix la funció membre **finalize** que actua de manera semblant. El control de memòria del **Java** recau en el *Garbage collector* (en endavant GC), a diferència del **C++** que s'ha d'implementar tot per el programador. Això significa que quan un objecte no s'utilitza, la memòria d'aquest és alliberada per el GC, i per tant, perd importància el destructor com alliberament de la memòria reservada. El mètode **finalize** s'executa just abans que el GC destrueixi l'objecte, però no es pot confiar plenament en l'execució del mètode. No sabem amb exactitud quan el GC tractarà l'objecte, i quins objectes seran tractats abans que els altres; és més, pot ser que el mètode no s'executi mai per què el programa ha finalitzat abans. Per tant, si s'ha d'alliberar recursos (fitxers oberts, sockets,...) al destruir la instància de la classe, és millor programar un mètode a part, per exemple *tancar()*, que s'encarregui de l'alliberament i cridar-lo explícitament des del programa.

Per declarar els constructors i els destructors, es segueix la mateixa pauta que amb els mètodes, però canviant la paraula reservada **metode** per **constructor** i **destructor**, respectivament. Els destructors no poden portar cap paràmetre, mentre que els constructors si accepten paràmetres, que es passen quan es fa la crida a **crear**.

Els constructors apareixen sempre en la interfície, per què són els mètodes que es criden implícitament amb la instrucció **crear**, i poden acceptar diferents paràmetres que s'han d'especificar. Tot i que els destructors es criden també implícitament amb la instrucció **destruir**, al no poder passar cap paràmetre, no es declaren en la part d'interfície.

4.5.6 Sobrecàrrega

També coneguda com polimorfisme o funcions polimòrfiques, significa la possibilitat de declarar un mètode una o més vegades, sempre que s'utilitzin paràmetres diferents per cada una. El conjunt de paràmetres d'un mètode o funció s'anomena signatura, i per tant, la sobrecàrrega permet tenir dos mètodes amb el mateix nom i diferent signatura. Si dos funcions tenen els paràmetres iguals, però diferent paràmetre de retorn, no es pot sobrecarregar, ja que la signatura només la formen els paràmetres formals, i es donarà un error en temps de compilació.

Per exemple, es podria sobrecarregar la funció suma d'una classe, per que es pogués sumar tant enters com reals:

```
metode suma(e: enter)
      // Suma l'enter a la classe
fmetode
metode suma(r:real)
      // Suma el real a la classe
fmetode
```

4.5.7 Herència

La herència és un dels mecanismes més potents del paradigma de programació orientada a objectes. Permet que una classe *hereti* automàticament els mètodes i atributs de la seva classe pare, anomenada també superclasse. Adicionalment, fa que qualsevol funció que tingui un paràmetre de

tipus de la classe pare, accepti també qualsevol classe filla, anomenada també subclasse.

Per definir que una classe hereta d'una altra, o que simplement és una subclasse, s'utilitza la paraula reservada **hereta** seguida d'un identificador que determina la classe pare, just al costat de la definició de la classe:

classe nomClasse **hereta** nomClassePare
...
fclasse

Els mètodes i funcions heretades, es poden sobreescriure en la classe filla, ja que com que les dades que emmagatzema la classe filla poden ser diferents que la del pare, també pot variar la implementació dels mètodes.

L'herència permet definir una estructura de classes jeràrquica, i ajuda a la reutilització de codi, ja que hi ha funcions que seran programades un sol cop en la classe pare, i les classes filles l'heretaren i en podran fer ús sense haver de tornar a implementar-les.

4.5.8 Les referències *jo* i *super*

Aquestes dues paraules reservades són referències a la pròpia classe i a la classe pare respectivament. Dintre una classe, et pots referir a un atribut o mètode posant a davant **jo**, seguit d'un punt:

jo.nomAtribut

La paraula **super** serveix per referir a un mètode o atribut de la classe pare. Això és molt útil, ja que permet accedir a mètodes de la classe pare, quan existeixen mètodes sobreescrits en la subclasse. Per accedir a un mètode de la superclasse, només cal escriure **super**, seguit de punt i l'atribut o mètode:

super.nomMetode(paràmetres_actuals)

La crida de **super** simplement, significa la crida al constructor de la classe pare:

super()

o bé amb paràmetres:

super(paràmetre1,paràmetre2,...)

4.6 Taules

Una taula és un grup de variables del mateix tipus, al qual ens podem referir mitjançant un nom en comú. El tipus pot ser tan un tipus bàsic, com l'enter, com qualsevol classe definida per l'usuari. Una taula és considerada com una classe, però degut al seu habitual ús, està dotada d'una sintaxi especial.

La seva declaració és:

nomTaula: **taula**[] de tipus

nomTaula es comporta com una referència a un objecte, i per tant, abans de poder ésser utilitzat, s'ha de instanciar, de la següent manera:

nomTaula <- **crear** tipus[*número*]

on número és un enter que indica la grandària de la taula.

Per accedir a cada element de la taula, s'indexa d'igual manera que es fa amb **C++** o **Java**:

nomTaula[índex]

on índex ha d'ésser una expressió de tipus entera.

Les taules poden ser també multidimensionals, i es tracta també d'igual manera que el **Java** i el **C++**:

nomTaula: **taula**[][] de tipus // Dos dimensions

nomTaula: **taula**[][][] de tipus // Tres dimensions

I a l'hora d'instanciar la taula, cal indicar la grandària de cada dimensió. Cal notar, que les dimensions poden tenir diferent grandària, per exemple:

```
var
      taula1: taula[][] de enter
fvar

taula1 <- crear enter[5][2]
```

Per poder saber el límit de la taula, podem fer servir el mètode **longitud()** que retorna el límit superior, és a dir, la llargària de la taula. Per cridar-lo, cal posar el nom de la taula, seguit d'un punt i la crida:

nomTaula.**longitud()**

Els dos mètodes retornen un enter. Cal notar, que el límit inferior és sempre zero, ja que s'utilitza el mateix format que el **Java**, mentre que **longitud()** retorna la grandària de la taula amb la qual s'ha instanciat. Si té més d'una dimensió, haurem d'accedir a cada una de les dimensions per saber la seva grandària.

Finalment, cal apuntar un error freqüent entre els programadors novells: si la taula està formada per objectes, enlloc de elements de tipus elemental, una vegada que s'ha instanciat la taula, no es pot accedir directament als elements. Cal primer, crear l'element, ja que com s'ha explicat, els objectes cal instanciar-los abans d'utilitzar-los. Vegem un exemple per clarificar la qüestió:

```
var
      taula1: taula[] de Classe1
fvar

taula1 <- crear Classe1[10]     // Instancia la taula amb 10 elements
taula1[0] <- crear Classe1()    // Instancia el primer element de la taula
taula1[0].metode1()            // Accedeix a un mètode del primer element
taula1[1] <- crear Classe1()    // Instancia el segon element de la taula
      ...
```

4.7 Ús de les classes

Per utilitzar una classe externa, només cal declarar-la com si fos un tipus elemental. El compilador, si no troba la definició en la taula de símbols, intenta carregar el fitxer i llegeix la

interfície d'aquest.

S'ha pogut veure anteriorment que l'operador punt és el que ens permet accedir als atributs i mètodes d'una classe. Per tant, si volem accedir a un mètode només cal posar el nom de la variable, seguit d'un punt i la crida al mètode pertinent. Lògicament només es pot accedir als atributs i mètodes públics, aquells que han estat declarats en el bloc d'interfície.

Recordem que els atributs i mètodes poden ésser d'instància o de la classe. Si són d'instància, els més comuns, fan referència a les dades de la pròpia instància, i són cridats externament:

nomInstanciaDeClasse.metode1(*paràmetres_actuals*)
nomInstanciaDeClasse.atribut1

En canvi si són de classe, no estan lligats a una instància en concret, sinó que són més generals. Podem pensar, per exemple, en la contant PI, o un mètode de comparació de dos enters. En aquest cas, per cridar-los ens referirem al nom de la classe, enlloc del nom de la instància:

NomClasse.metodeClasse1(*paràmetres_actuals*)

NomClasse.atributClasse1

NomClasse.CONSTANT1

4.8 Format del fitxer font

Els fitxers font de pseudocodi 03 tenen per defecte l'extensió ".pse". En un fitxer hi pot haver-hi un algorisme o una classe. Les definicions de classe i algorisme s'han vist en els capítols anteriors. Quan s'assigna un nom a la classe o algorisme, és convenient (però no necessari) que sigui igual al nom del fitxer. Ja que, el fitxer de sortida que genera el compilador té el mateix nom que el fitxer origen, canviant l'extensió per ".java". I quan es passa el compilador del **Java** es generen les classe binàries en *bytecode*, i aquests fitxers si tenen el mateix nom que la classe declarada en el fitxer font. Per tant, poden diferir del nom del fitxer **Java** o del pseudocodi original. Per això, s'aconsella que la classe/algorisme tinguin el mateix nom que el fitxer, ja que així el fitxer **Java** i el fitxer de classe final tindran tots tres el mateix nom (naturalment diferent extensió). D'altra manera, sorgirien fitxers amb noms diferents i la situació podria esdevenir en un petit caos de noms.

4.9 Restriccions del traductor

El traductor posseeix unes restriccions que no el fan cent per cent compatible amb el pseudocodi.

Primerament, la visibilitat dels atributs i mètodes és tractada com el **C++**, ha de ser primer declarats per poder ésser utilitzats/cridats. El **Java** permet referències endavant, és a dir, cridar un mètode que està declarat més endavant, mentre que el traductor no.

Finalment, no s'accepta l'assignació múltiple, ja que no ha estat implementada. Això no és un gran problema, per què les instruccions amb assignacions múltiples es poden desglossar ràpida i fàcilment en instruccions amb assignacions simples.