

# **Squeak: Aprèn a Programar amb Robots**

**Stéphane Ducasse**

**Traducció de Jordi Delgado  
Octubre 2008**

Aquest llibre està disponible per a ser descarregat a [citilab.eu](http://citilab.eu), hostatjat per Citilab-Cornellà.

**Copyright © 2005, Stéphane Ducasse.**

**Copyright © 2008 de la traducció, Jordi Delgado.**

Els continguts d'aquest llibre estan protegits sota la llicència Creative Commons Reconeixement-Compartir amb la mateixa llicència 3.0 No adaptada.

*Sou lliure de:*

**Compartir:** copiar, distribuir i comunicar públicament l'obra

**Modificar:** fer-ne obres derivades

*Amb les condicions següents:*

**Reconeixement.** Heu de reconèixer els crèdits de l'obra de la manera especificada per l'autor o el llicenciat (però no d'una manera que suggerexi que us donen suport o rebeu suport per l'ús que feu l'obra).

**Compartir amb la mateixa llicència.** Si transformeu o modifiqueu aquesta obra per generar-ne una obra derivada, només podreu distribuir l'obra resultant amb la mateixa llicència, una de similar o una de compatible.

- Quan reutilitzeu o distribuïu l'obra, heu de deixar ben clar els termes de la llicència de l'obra. Podeu posar un link a aquesta web: [creativecommons.org/licenses/by-sa/3.0/deed.ca](http://creativecommons.org/licenses/by-sa/3.0/deed.ca)
- Alguna d'aquestes condicions pot no aplicar-se si obtenuï el permís del titular dels drets d'autor.
- No hi ha res en aquesta llicència que menyscabi o restringeixi els drets morals de l'autor.

Els drets derivats d'usos legítims o altres limitacions reconegudes per llei no queden afectats per l'anterior.

Això és un resum fàcilment llegible del text legal, la llicència completa la podeu trobar a [creativecommons.org/licenses/by-sa/3.0/legalcode](http://creativecommons.org/licenses/by-sa/3.0/legalcode)



**Traducció realitzada amb la col·laboració del Citilab-Cornellà.**

[citilab.eu](http://citilab.eu)

Publicat per ??

ISBN xxx-x-xxxxxxx

Primera edició, Tardor 2008.

# Índex

Prefaci	xiii
Sobre l'autor	xix
Agraïments	xxi
Nota del traductor	xxiii
Pròleg	xxv
<b>I Començar</b>	<b>1</b>
<b>1 Instal·lació i creació de robots</b>	<b>3</b>
Instal·lar l'entorn . . . . .	3
Instal·lació en un Macintosh . . . . .	4
Instal·lació en Windows . . . . .	4
Obrir l'entorn . . . . .	5
Ajuts a la instal·lació . . . . .	7
Primeres interaccions amb un robot . . . . .	7
Enviar missatges al robot . . . . .	7
Crear un nou robot . . . . .	10
Sortir i guardar . . . . .	10
Problemes amb la instal·lació . . . . .	12
Resum . . . . .	14
<b>2 Un primer <i>script</i> i les seves implicacions</b>	<b>15</b>
Utilitzar una cascada per enviar múltiples missatges . . . . .	15
Primer <i>script</i> . . . . .	17

Squeak i Smalltalk . . . . .	19
Llenguatges de programació . . . . .	19
Smalltalk i Squeak . . . . .	19
Programes, expressions i missatges . . . . .	20
Escriure i executar programes . . . . .	20
Anatomia d'un <i>script</i> . . . . .	21
Sobre els píxels . . . . .	22
Expressions, missatges i mètodes . . . . .	23
Separació de missatges . . . . .	24
Mètode . . . . .	25
Cascada . . . . .	25
Crear nous robots . . . . .	26
Errors als programes . . . . .	27
Escriure malament un selector de missatge . . . . .	28
Escriure malament el nom d'una variable . . . . .	28
Variables no utilitzades . . . . .	29
Majúscules o minúscules? . . . . .	29
Oblidar un punt . . . . .	31
Paraules que canvien de color . . . . .	31
Resum . . . . .	33
<b>3 Homes i robots</b> . . . . .	<b>35</b>
Crear robots . . . . .	35
Dibuixar segments de línia . . . . .	37
Canviar de direcció . . . . .	38
L'ABC del dibuix . . . . .	41
Controlar la visibilitat del robot . . . . .	42
Resum . . . . .	43
<b>4 Direccions i angles</b> . . . . .	<b>45</b>
Dreta o esquerra? . . . . .	45
Una convenció direccional . . . . .	47
Orientació absoluta vs. orientació relativa . . . . .	48
L'enfocament adequat . . . . .	50
Un rellotge robot . . . . .	54
Dibuixos senzills . . . . .	55
Polígons regulars . . . . .	56
Resum . . . . .	58

<b>5 L'entorn de Pica</b>	<b>61</b>
El menú principal . . . . .	61
Obtenir un Espai de Treball Bot . . . . .	62
Interaccionar amb Squeak . . . . .	63
Utilitzar l'Espai de Treball Bot per guardar un <i>script</i> . . . . .	65
Carregar un <i>script</i> . . . . .	65
Capturar un dibuix . . . . .	66
Resultat dels missatges . . . . .	67
Executar un <i>script</i> . . . . .	69
Consells . . . . .	69
Dos exemples . . . . .	70
Resum . . . . .	72
<b>6 Divertim-nos amb els robots</b>	<b>73</b>
Nances del robot . . . . .	73
Mida del llapis i color . . . . .	74
Més sobre els colors . . . . .	76
Canviar la forma i la mida dels robots . . . . .	77
Dibuixar el vostre propi robot . . . . .	79
Guardar i restaurar dibuixos . . . . .	81
La nansa “Guardar Gràfic” . . . . .	81
Re-equipar la fàbrica de robots . . . . .	82
Operacions gràfiques utilitzant <i>scripts</i> . . . . .	83
Resum . . . . .	88
<b>II Conceptes elementals de programació</b>	<b>89</b>
<b>7 Repetir</b>	<b>91</b>
Ha nascut una estrella . . . . .	91
Bucles al rescat . . . . .	93
Repeticions en marxa . . . . .	94
Sagnat de codi . . . . .	95
Dibuixar figures geomètriques regulars . . . . .	96
Redescobrir les piràmides . . . . .	97
Més experiments amb repeticions . . . . .	99
Resum . . . . .	101

<b>8 Variables</b>	<b>103</b>
Per cortesia de la lletra A . . . . .	104
Variacions sobre el tema de la A . . . . .	104
Variables al rescat . . . . .	106
Declarar una variable . . . . .	107
Assignar un valor a una variable . . . . .	107
Referir-nos a les variables . . . . .	108
I què passa amb Pica? . . . . .	108
Utilitzar variables . . . . .	108
La potència de les variables . . . . .	110
Expressar les relacions entre variables . . . . .	110
Experimentar amb variables . . . . .	111
Les piràmides redescobertes . . . . .	113
Polígons automatitzats utilitzant variables . . . . .	114
Polígons regulars de mida fixa . . . . .	115
Resum . . . . .	116
<b>9 Aprofundir en les variables</b>	<b>117</b>
Anomenar les variables . . . . .	117
Variables com a contenidors . . . . .	118
Assignació: L'esquerra i la dreta de := . . . . .	119
Analitzar alguns <i>scripts</i> senzills . . . . .	120
Resum . . . . .	125
<b>10 Repeticions i variables</b>	<b>127</b>
Una escala estranya . . . . .	128
Pràctiques amb repeticions i variables: laberints, espirals i altres . . . . .	131
Alguns aspectes importants d'utilitzar variables i repeticions . . . . .	134
Inicialització de variables . . . . .	134
Utilitzar i canviar el valor d'una variable . . . . .	134
Experiments avançats . . . . .	136
Resum . . . . .	137
<b>11 Compondre missatges</b>	<b>139</b>
Els tres tipus de missatges . . . . .	140
Identificar missatges . . . . .	141
Els tres tipus de missatges en detall . . . . .	143
Missatges unaris . . . . .	143
Missatges binaris . . . . .	144

Missatges de paraula-clau . . . . .	145
Ordre d'execució . . . . .	146
Regla 1: Unari > Binari > Paraula-Clau . . . . .	147
Regla 2: Primer els parèntesis . . . . .	150
Regla 3: D'esquerra a dreta . . . . .	151
Resum . . . . .	155
<b>III Posar en joc l'abstracció</b>	<b>157</b>
<b>12 Mètodes: seqüències de missatges amb nom</b>	<b>159</b>
<i>Scripts</i> versus mètodes . . . . .	160
Com definir un mètode? . . . . .	161
Explorador de la classe Bot . . . . .	162
Crear una nova categoria de mètodes . . . . .	163
Definir el vostre primer mètode . . . . .	165
Què hi ha en un mètode? . . . . .	166
<i>Scripts</i> versus mètodes: Anàlisi . . . . .	168
Retornar un valor . . . . .	169
Dibuixar figures . . . . .	170
Resum . . . . .	172
Glossari . . . . .	173
<b>13 Combinar mètodes</b>	<b>175</b>
Res de nou: Revisitar el mètode quadrat . . . . .	176
Altres patrons gràfics . . . . .	176
Què us diuen aquests experiments? . . . . .	178
Quadrats a tot arreu . . . . .	179
Resum . . . . .	181
<b>14 Paràmetres i arguments</b>	<b>183</b>
Què és un paràmetre? . . . . .	184
Un mètode per dibuixar quadrats . . . . .	184
Practiqueu amb els paràmetres . . . . .	186
Variables en mètodes . . . . .	187
Experimentar amb múltiples arguments . . . . .	189
Paràmetres i variables . . . . .	191
Arguments i paràmetres . . . . .	193
Sobre l'execució dels mètodes . . . . .	195

Resum . . . . .	196
<b>15 Errors i depuració</b>	<b>197</b>
El valor per defecte d'una variable . . . . .	197
Examinar l'execució d'un missatge . . . . .	199
Una primera ullada al depurador . . . . .	202
Anar pas a pas per la pila . . . . .	206
Corregir els errors . . . . .	209
Exemple 1 . . . . .	209
Exemple 2 . . . . .	210
Resum . . . . .	212
<b>16 Descompondre per recompondre</b>	<b>213</b>
Laberints i espirals . . . . .	214
Quadrats centrats . . . . .	214
Espirals . . . . .	216
Rectangles d'or . . . . .	219
Una solució d'un-rectangle-per-línia . . . . .	220
Enrajolar . . . . .	224
Resum . . . . .	227
<b>17 Cadenes, i eines per entendre programes</b>	<b>229</b>
Cadenes . . . . .	229
La comunicació amb l'usuari . . . . .	230
Cadenes i caràcters . . . . .	231
Cadenes i nombres . . . . .	233
Utilitzar el <i>Transcript</i> . . . . .	234
Generar i comprendre una traça . . . . .	235
Resum . . . . .	239
<b>IV Condicionals</b>	<b>241</b>
<b>18 Condicions</b>	<b>243</b>
Els autèntics colors d'un robot . . . . .	243
Afegir una traça per veure què va passant . . . . .	245
El valor retornat per un mètode . . . . .	246
Expressions condicionals amb una sola bifurcació . . . . .	247
Tria el mètode condicional adequat . . . . .	248
Imbricar expressions condicionals . . . . .	248

Acolorir robots amb tres colors . . . . .	248
Aprendre dels errors . . . . .	250
Interpretar un petit llenguatge . . . . .	252
Més experiments . . . . .	254
Resum . . . . .	255
<b>19 Repeticions condicionals</b> . . . . .	<b>257</b>
Repeticions condicionals . . . . .	257
Un exemple . . . . .	258
Experiències amb traces . . . . .	260
Aturar un bucle infinit . . . . .	263
Aprofundir en els bucles condicionals . . . . .	264
Una aplicació interactiva senzilla . . . . .	265
Quan hem d'utilitzar claudàtors . . . . .	267
Resum . . . . .	267
<b>20 Booleans i expressions booleans</b> . . . . .	<b>269</b>
Valors booleans i expressions booleans . . . . .	269
Valors booleans . . . . .	269
Expressions booleans . . . . .	270
Combinar expressions booleans bàsiques . . . . .	271
Negació (no) . . . . .	272
Conjunció (i) . . . . .	273
Disjunció (o) . . . . .	273
Tot junt . . . . .	274
Alguns aspectes d'Smalltalk . . . . .	274
Oblidar parèntesis (un error freqüent) . . . . .	274
Cas d'estudi . . . . .	275
Utilitzar el depurador . . . . .	275
Entendre el problema . . . . .	276
Problemes semblants i solucions . . . . .	277
Resum . . . . .	278
<b>21 Coordenades, punts i moviments absoluts</b> . . . . .	<b>279</b>
Punts . . . . .	280
Utilitzar quadriculares . . . . .	282
Una font d'errors amb els punts . . . . .	283
Descompondre $50@60 + 200@400$ . . . . .	284
Descompondre $(50@60) + (200@400)$ . . . . .	284

Moviments absoluts . . . . .	285
Moviment relatiu versus moviment absolut . . . . .	285
Alguns experiments . . . . .	288
Translació . . . . .	288
Traslladar triangles . . . . .	290
Oques volant . . . . .	291
Utilitzar moviments absoluts . . . . .	292
Bucles i translació . . . . .	295
Més experiments . . . . .	296
Resum . . . . .	297
<b>22 Comportament avançat dels robots</b>	<b>299</b>
Obtenir la direcció d'un robot . . . . .	299
Apuntar en una direcció . . . . .	300
Distància des d'un punt . . . . .	301
Tornar al centre de la pantalla . . . . .	301
Posició si es moguéss . . . . .	301
En una capsa . . . . .	302
Apuntar cap a un punt . . . . .	303
Centre versus posició . . . . .	305
Resum . . . . .	305
<b>23 Simular comportament animal</b>	<b>307</b>
Vagar . . . . .	307
Separar influències . . . . .	309
Estudiar la influència de la longitud . . . . .	309
Estudiar la influència del costat cap a on gira l'animal . . . . .	310
Atrapat dins una capsa . . . . .	312
Resseguir els costats . . . . .	313
Volar al costat oposat . . . . .	314
Direcció aleatòria . . . . .	314
Afegir una sortida a la capsa . . . . .	315
Romandre en un entorn segur . . . . .	316
Més experiments . . . . .	319
Trobar menjar . . . . .	319
Comparar la distància . . . . .	319
Més experiments . . . . .	321
Mantenir l'orientació . . . . .	322
Simular la visió . . . . .	324

Resum . . . . .	326
-----------------	-----

## V Altres mons Squeak 327

<b>24 Un recorregut per eToy</b> <span style="float: right;">329</span>	
Pilotar un avió . . . . .	330
Pas 1: Dibuixar un avió . . . . .	330
Pas 2: Jugar amb l'halo . . . . .	330
Pas 3: Arrosseggar i deixar un mètode per crear nous <i>scripts</i> . . . . .	335
Pas 4: Afegir mètodes . . . . .	337
Joysticks en acció . . . . .	339
Pas 1: Crear un <i>joystick</i> . . . . .	339
Pas 2: Experimentar amb un <i>joystick</i> . . . . .	339
Pas 3: Vincular el <i>joystick</i> i l' <i>script</i> . . . . .	339
Crear una animació . . . . .	341
Pas 1: Crear el contenidor . . . . .	341
Pas 2: Dibuixar els elements de l'animació . . . . .	342
Pas 3: Deixar els dibuixos dins del contenidor . . . . .	343
Pas 4: Crear un dibuix senzill com a base de l'animació . . . . .	343
Pas 5: Crear un <i>script</i> amb lookLike . . . . .	344
Pas 6: Mostrar l'element seleccionat de l'animació . . . . .	344
Pas 7: Canviar l'element seleccionat d'un contenidor . . . . .	345
Una altra manera . . . . .	346
Cotxes i conductors . . . . .	347
Pas 1: Dibuixar un cotxe i un volant . . . . .	347
Pas 2: Girar el cotxe en un cercle . . . . .	347
Pas 3: Utilitzar la direcció del volant . . . . .	348
Pas 1: Sensors . . . . .	350
Pas 2: La carretera . . . . .	350
Pas 3: Condicions i tests a eToy . . . . .	350
Pas 4: Personalitzar els tests basats en colors . . . . .	351
Pas 5: Afegir accions . . . . .	352
Alguns trucs . . . . .	353
Executar diversos <i>scripts</i> . . . . .	354
Eliminar . . . . .	355
Crear una icona . . . . .	355
Internacionalització . . . . .	356
Resum . . . . .	356

<b>25 Un Recorregut per Alice</b>	<b>357</b>
Començar amb Alice . . . . .	357
Interactuar directament amb els actors . . . . .	359
L'entorn . . . . .	361
Scripts . . . . .	362
Analitzar el primer <i>script</i> . . . . .	364
Moure, girar i rodar . . . . .	365
Parts dels actors . . . . .	366
Altres operacions . . . . .	368
Fer-se gran . . . . .	368
Moviments quantificats . . . . .	368
Mantenir-se dret . . . . .	368
Acolorir . . . . .	368
Destrucció . . . . .	369
Visibilitat . . . . .	369
Moviments absoluts i rotacions . . . . .	369
Apuntar . . . . .	369
Posicionament relatiu dels actors. . . . .	370
Accions relacionades amb el temps . . . . .	370
Animació . . . . .	371
El vostre propi <i>Wonderland</i> . . . . .	372
Múltiples càmeres i altres efectes especials . . . . .	373
Alarms . . . . .	374
Introduir la interacció amb l'usuari . . . . .	375
Aspectes ocults d'Alice i Pooh . . . . .	376
Projectar <i>morphs</i> 2D en 3D . . . . .	377
Pooh: generar formes 3D a partir de 2D . . . . .	378
Resum . . . . .	380
<b>Índex Alfabètic</b>	<b>381</b>

# Prefaci

**Alan Kay**

President, *Viewpoints Research Institute, Inc.* &

Sr. Fellow, *Hewlett-Packard Company*

## El futur de la programació, vist des dels anys 60

Vaig començar el doctorat (a la Universitat de Utah, dins el projecte ARPA) el novembre de 1966, i val a dir que és interessant mirar enrera al món de la programació tal com jo el veia aleshores.

L'extraordinària Jean Sammit (que va ser inventora de diversos llenguatges de programació i la seva primera historiadora, així com la primera dona presidenta de l'ACM) va comptar uns 3.000 llenguatges de programació utilitzats activament a finals dels anys 60. S'estava treballant molt en aquest camp, i part d'aquesta feina era molt important i molt interessant.

Algol 60, tal com Tony Hoare va assenyalar, “va ser una enorme millora, especialment respecte als seus successors!”. Tenia moltes virtuts, incloent més èmfasi en els contextos i els entorns dins la seva semàntica i una característica remarcable pel seu moment, la crida per nom (*call by name*) que permetia als programadors una capacitat d'expressió similar a la dels mateixos dissenyadors del llenguatge. Per exemple, hom podia escriure accions que tinguessin el mateix significat i la mateixa execució que les instruccions de control pròpies del llenguatge:

```
for (i, 1, 10, print(a[i]))
```

on el primer i el quart paràmetres es marcarien com a name i així s'integrarien en una expressió que seria capaç de recordar correctament l'espai de noms jeràrquic al que pertanyen les seves variables, però que a més podia ser manipulat i executat des de dins el cos de l'acció for. Ni el LISP original va implementar correctament això al principi!

I hi havia una variant sintàctica poc coneguda en la sintaxi oficial d'Algol 60 que promovia una forma més lleigible per a les accions definides per l'usuari. Això permetia que una coma en una crida a una acció pogués ser substituïda per la següent construcció:

): comentari (

la qual cosa hagués permès que la crida precedent fos escrita de la següent manera:

```
for (i): desde (1): fins (10): fer (print(a[i]))
```

Si féssiu això en una pantalla o en una IBM *Executive typewriter* convertida en terminal (tal com podíeu fer amb JOSS) obtindríeu:

```
for (i): desde (1): fins (10): fer (print(a[i]))
```

que és ben bé com el llenguatge Algol de base, però amb una meta-extensió afegida pel programador en benefici d'altres programadors.

És possible que el conjunt d'idees i representacions més profundes al voltant dels llenguatges de programació succeís abans d'Algol, però van trigar força a ser enteses pels informàtics (i molts mai no les van entendre), en part per una notació diferent i difícil de llegir (per gent de fora, si més no), i perquè moltes de les millors aportacions de LISP eren "realment meta". Una de les grans contribucions de LISP va ser el seu avaluador escrit en LISP mateix, en mitja pàgina de codi. Això va ser una mena d'equacions de Maxwell per a la programació, i va permetre pensar moltes coses que eren impensables amb els enfocaments habituals.

LISP mateix va ser utilitzat per ser el sistema en el que es va programar l'*Advice Taker*, un agent interactiu amb sentit comú, que podia entendre els desitjos humans expressats en llenguatge natural i els transformava en processos informàtics que duien a terme aquells mateixos desitjos. A mitjans dels anys 60 es van crear alguns llenguatges intermedis interessants, com FLIP, i intents de reproduir part de les propietats de l'*Advice Taker*, com PILOT.

Sketchpad potser va ser el més radical d'aquests sistemes desenvolupats inicialment, ja que va intentar proposar directament un marc raonablement interactiu per usuaris que volien utilitzar l'ordinador per fer allò pel que està millor preparat: tota mena de simulacions interactives. Les tres grans contribucions d'Sketchpad van ser:

- La primera aproximació utilitzable als gràfics interactius per ordinador.
- Una véritable estructura d'objectes per a totes les seves entitats.
- Una manera declarativa de programar en termes dels resultats finals desitjats, on el sistema podia utilitzar diversos mètodes automàtics de resolució de problemes per aconseguir els resultats buscats.

Això va ser impulsat tremendament per una "aproximació tolerant" a resoldre restriccions, on, en lloc de procurar trobar solucions lògico simbòliques perfectes pels conjunts de restriccions, les restriccions s'intentaven resoldre amb toleràncies de tipus global. Aquesta manera de fer va

permetre aproximar la solució de problemes importants que avui dia són difícils de resoldre o fins i tot intractables.

JOSS era tota una altra cosa: no feia “pràcticament res” (bàsicament càcul numèric utilitzant nombres i taules), però el que feia ho feia perfectament; fins i tot ara podem considerar que tenia una de les millors interfícies d’usuari de la història.

*A Programming Language* era el títol d’un llibre de Kenneth Iverson que va adoptar un enfoc fortament matemàtic a la programació, via funcions i meta-funcions expressades en una mena d’àlgebra. El llenguatge definit al llibre va rebre el nom d’”Iverson”. Un sistema real amb què es pogué programar un ordinador existia només com a rumor dins d’IBM en aquells temps, però sobre el paper es van escriure molts programes utilitzant aquestes idees. El millor d’Iverson era que realment sortia a compte si s’hi pensava en termes de transformacions i relacions matemàtiques, i no es considerava gaire el cost de les operacions. No preocupar-se pel cost era quasi impensable en aquells dies de rellotges d’1Mhz en ordinadors de la mida d’edificis i preus de millions de dòlars, per tant Iverson i LISP eren vehicles alliberadors a l’hora de pensar en el futur, un temps en el qual les màquines serien més petites i més ràpides.

Els dissenyadors de Simula volien modelitzar estructures dinàmiques grans i complexes i van adonar-se que els blocs d’Algol podien fer el fet si s’aconseguia alliberar-los de l’estricta estructura de control jeràrquica d’Algol. Quan van crear Simula I a mitjans dels anys 60 van ser capaços de veure que les seves idees eren força importants per al llenguatge i la seva programació, i quan van fer Simula 67 van poder reemplaçar molts tipus de dades que venien incorporats, com l’string, amb classes de Simula 67.

La idea d’extendre la sintaxi, la semàntica i la pragmàtica dels llenguatges de programació era tot un camp de recerca a mitjans i finals dels anys 60. Una de les raons d’això és que estava molt clar que la programació és una tasca on és difícil canviar d’escala, i que l’escalabilitat en moltes dimensions seria crítica per a la salut de la informàtica. On la complexitat és un problema central, l’arquitectura preval sobre els materials. La comprensió d’aquesta idea va fer que es comencés a veure que la programació és diferent de les matemàtiques, tot entenent-la com una nova forma d’enginyeria. Hi va haver propostes per a la formació d’una nova disciplina anomenada “Enginyeria del Software”, i per a conferències, amb l’únic propòsit de treure l’igua clara sobre el significat d’aquest nou nom (què fer quan no pots fer només matemàtiques).

L’Information Processing Techniques Office (IPTO) de l’ARPA estava en plena activitat quan vaig començar el doctorat l’any 1966, i ja tenia en marxa uns quants projectes cap al somni col·lectiu de tenir computació interactiva per a tothom, connectats via una “xarxa intergalàctica”. Intentar construir aquesta xarxa (amb importants requeriments d’escala) va generar part de les millors idees en sistemes informàtics de l’època, i va ser part important de les meves reflexions sobre el futur de la programació.

Els proveïdors de finançament d’ARPA van ser llestos i no van transformar la visió i el somni en objectius financers, en lloc d’això, van intentar trobar i finançar individus amb talent que tenien les seves pròpies idees sobre què significava el somni i com podia ser implementat. Això

va donar lloc a uns 17 grups dins d'universitats i empreses, la majoria dels quals van proposar diferents dissenys i demostracions molt interessants. Així es va constituir una comunitat tant de discussió com de col-laboració que va fer a tothom que hi pertanyia més illes del que era abans d'afegir-s'hi.

Naturalment, donats els 3.000 llenguatges de Jean Sammit, hi ha molt que no he explicat, igual que molts dissenys interessants que s'han dut a terme des de 1967 fins al final de la dècada dels 70. Per triar només cinc desenvolupaments de particular importància per als lectors d'aquest llibre, jo em quedaria amb la meva pròpia concepció dels objectes i com se suposa que havien de ser útils als usuaris d'ordinadors personals; PLANNER, de Carl Hewitt, que era el sistema més cohesiu per practicar la "programació com a raonament"; IMP, de Ned Iron, que representa potser el primer llenguatge útil completament extensible; i el *Control Definition Language* de Dave Fisher, que va il·luminar les propietats d'extensibilitat en general i les de les estructures de control en particular.

Em vaig formar en matemàtiques, en biologia molecular (vaig pagar-me els estudis com a programador) i en art. Diverses circumstàncies van portar-me a haver d'entendre Sketchpad, Simula i la xarxa intergal·làctica proposada per ARPA en la meva primera setmana dins l'escola de doctorat, i la reacció que això em va provocar va ser cataclísmica. Eren similars en alguns aspectes i diferents en d'altres, però eren diverses espècies del mateix gènere, si hom prenia un punt de vista tant biològic com matemàtic. Biològicament, eren "quasi cèl·lules" demanant ser cèl·lules. Matemàticament, eren "quasi àlgebres" demanant ser àlgebres. Així, la meva fusió inicial d'aquestes metàfores amb la computació va donar lloc a la idea que es podia fer qualsevol cosa a partir d'entitats que hom podia considerar ordinadors virtuals amb la capacitat d'enviar missatges (els quals havien de ser també ordinadors virtuals). Els ordinadors virtuals actuarien com a cèl·lules i els protocols inventats podien ser força algebraics -el que avui dia anomenem (incorrectament) *polimorfisme*. Això donaria lloc a una simplicitat i escalabilitat més gran a "nivell dels materials", i obriria la porta a progrés en simplicitat i escalabilitat al "nivell de les expressions", que és on viu el programador.

Alguns anys després vaig trobar PLANNER, de C. Hewitt, i em vaig adonar que aquella era la manera de fer programes més expressius i escalables (moltes de les idees de PLANNER van aparèixer després en el llenguatge Prolog). Estava força clar que intentar enviar missatges orientats als objectius podia incrementar l'escalabilitat, en part perquè hi ha moltes més maneres d'intentar satisfer objectius que objectius (penseu a ordenar com a objectiu i en totes les maneres que hi ha d'ordenar), i aquesta separació podia proporcionar beneficis alhora mantenint els programes més expressius i menys barrejats amb altres problemes com l'optimització.

Mentrestant, va aparèixer el llenguatge extensible IMP, amb algunes bones idees que en van fer un llenguatge pràctic, i no satisfet únicament amb les seves capacitats "meta".

I, en paral·lel amb la tesi en què jo estava treballant sobre ordinadors personals i sistemes orientats a objectes per a tota mena d'usuaris, Dave Fisher estava treballant en un conjunt complementari d'idees molt maques sobre com fer les estructures de control extensibles via la capacitat

d'afegir nova semàntica dinàmicament a un meta-interpret de l'estil de LISP.

LOGO, el primer gran llenguatge de programació per a nens, era una combinació encertada de JOSS i LISP, feta per Papert, Feurzig, Bobrow i altres a BBN. Això va introduir la idea dels nens com a usuaris finals d'idees poderoses en computació, i va transformar la meva idea de la computació com a eina o vehicle en una idea de la computació com a *mitjà* d'expressió amb un destí còsmic similar al de la impremta.

Aquest cinc sistemes i la invitació per ajudar a arrencar Xerox PARC van ser l'empenta per a Smalltalk, la qual cosa es deixa veure més en les seves primeres versions.

Mirant enrera, és sorprendent que:

- L'expressivitat dels llenguatges de programació actuals sigui tan baixa (a l'alçada del que hi havia cap el 1965) i que poquíssims programadors treballin al nivell que LISP o Smalltalk ja feien possible en els anys 70.
- Smalltalk no ha canviat apreciablement des que va sortir la versió avui coneguda com Smalltalk-80, fins i tot considerant que conté el seu propi meta-sistema i que per tant és molt senzill de millorar.
- La llei de Moore de 1965 ha resultat ser força correcte, i ara podem construir maquinari i programari molt grans, tot i que també molt fràgils, ja que no han estat considerats els conceptes escalables més enllà de la visió simple dels objectes (potser tenim "cèl·lules", però no conceptes equivalents als teixits, o com fer créixer o construir organismes multi-cel·lulars).
- Internet ha acabat sent l'expressió amb èxit d'una aproximació radical a l'arquitectura i a l'escalabilitat, tot i així, cap programari o sistema de programació ha estat preparat per a que els programadors puguin expressar sistemes similars a Internet.

Què ha passat amb el progrés en els darrers 25 anys? I per què Squeak és essencialment només un Smalltalk gratuït, si necessitem progressar desesperadament?

Al 1995 Internet era prou madura per intentar alguns experiments amb *media* que feia temps que volíem fer. I el Java (i altres llenguatges) de l'època (i el d'ara) era força defectuós en aspectes com la flexibilitat, les capacitats "meta" i la portabilitat com per ser-nos un vehicle útil. Ja havíem fet Smalltalk abans, i havíem escrit un llibre sobre com construir-ne un sistema complet, era per tant raonable prendre's un any per fer un Smalltalk gratuït i controlable i distribuir-lo per Internet (de fet, ens va portar uns nou mesos). La idea era que Squeak no hauria de ser un vehicle sinó una fàbrica per a un llenguatge del segle XXI que fos molt millor.

Malgrat tot, els sistemes de programació amb què els programadors treballen sovint tenen vida pròpia, i gran part del moviment *open source* al voltant d'Squeak va mostrar-se més interessat en un Smalltalk gratuït amb un sistema multimèdia portable. Crec que és raonable afirmar que la

majoria de la comunitat Squeak està dedicada a fer aquest Smalltalk més útil i accessible, i no pas a fer alguna altra opció que deixi Smalltalk obsolet (un destí que m'encantaria veure realitzat).

Per això, m'agradaria encoratjar els lectors d'aquest nou i excel·lent llibre a no pensar Smalltalk com un conjunt de característiques imposades pels venedors a les que ens hem de resignar, sinó com un sistema que és capaç de ser ampliat en totes les seves dimensions i que recompensarà tots aquells que inventin noves i millors maneres de programar. Al PARC nosaltres canviàvem Smalltalk cada poques setmanes, i d'una manera significativa cada dos anys. Encara que amb prou feines ha canviat des d'aleshores, si us plau proveu de canviar-lo, i poseu aquestes modificacions a Internet perquè tots en puguem aprendre i gaudir!

# Sobre l'autor

**STÉPHANE DUCASSE** va obtenir el seu doctorat a la Universitat de Nice-Sophia Antipolis i la seva habilitació a la Universitat de París 6. Va rebre el premi SNF 2002 *Professeur Boursier*. Actualment és professor a la Universitat de Berna i a la Universitat de Savoie<sup>1</sup>.

Les arees d'interès de Stéphane són el disseny de sistemes reflexius, el disseny de llenguatges orientats a objectes, els components de programari, el disseny i la implementació d'aplicacions, la re-enginyeria d'aplicacions orientades a objectes i l'ensenyament. És el desenvolupador principal del *Moose Reengineering Environment*. Adora programar en Smalltalk i és president de l'ESUG (*European Smalltalk Users Group*).

Stéphane ha escrit diversos llibres en francès i anglès: *La programmation: une approche fonctionnelle et recursive en Scheme* (Eyrolles, 1996), *Squeak* (Eyrolles, 2001), i *Object-Oriented Reengineering Patterns* (MKP, 2001).

Si vols descobrir perquè Stéphane es diverteix amb Squeak i participa activament en el seu desenvolupament, visita <http://www.squeak.org>.

Visita <http://smallwiki.unibe.ch/botsinc> pel lloc web d'aquest llibre<sup>2</sup>.

---

<sup>1</sup>Nota del Traductor: Actualment (gener 2008) és director de recerca a l'INRIA (Lille, França)

<sup>2</sup>Nota del Traductor: En anglès



# Agraïments

És un plaer donar-vos les gràcies a tots aquells de vosaltres que heu llegit parts i esborrany s d'aquest llibre i m'heu donat *feedback*. No és senzill llegir un treball inacabat, i us agraeixo que hagieu fet l'esforç. No intentaré llistar els vostres noms aquí, ja que segurament oblidaria algú. Tot i així, hauria de mencionar l'Orla Greevy, l'Ian Prince i en Daniel Knierim, que van llegir el manuscrit sencer. Gràcies pel vostre *feedback* i recolzament. També m'agradaria mencionar en particular el Daniel Villain, que va llegir un esborrany de la versió francesa.

Voldria agrair a la comunitat Squeak per l'ajut proporcionat mentre desenvolupava els entorns que utilitzo en aquest llibre, i per desenvolupar aquest entorn tan magnífic que és Squeak, principalment. En particular, vull agrair al Nathanel Schärli i al Ned Konz el seu ajut. Agraïments especials a tots aquells desenvolupadors que han ajudat Smalltalk a deixar de ser un somni i esdevenir una realitat. També m'agradaria donar les gràcies a tots els "Smalltalkers" que han fet d'aquest llenguatge i aquesta comunitat quelcom tan emocionant. Que continueu fent realitat els vostres somnis.

Escriure aquest llibre ha estat un procés llarg i complicat, ja que ensenyar novells és difícil. És més, no sóc una persona amb qui sigui fàcil conviure, i com a investigador, m'emocionen massa temes. Vull agrair particularment Didier Basset les discussions molt fructíferes al començament d'aquest projecte.

També voldria donar les gràcies a la meva esposa, la Florence, i els meus fills el Quentin i el Thibaut, dos noiets a qui encantava córrer sorollosament al voltant de la taula mentre intentava concentrar-me en la meva feina. Gràcies per acceptar un marit i un pare que no era sempre present, entusiasta ni accessible. Aviat estarem programant junts.



# Nota del traductor

Aquest llibre que teniu entre mans és la traducció al català del llibre del professor Stéphane Ducasse:

Squeak: Learn Programming with Robots  
Stéphane Ducasse, APress 2005  
ISBN:1-59059-491-6

El programari BotsInc que acompaña al llibre en anglès el teniu a  
<http://smallwiki.unibe.ch/botsinc/download/>

Teniu una traducció parcial al català d'aquest programari a <http://citilab.eu>, que és la que he utilitzat en aquesta traducció del llibre. S'han traduït: Les classes associades a l'entorn BotsInc que són accessibles per a l'usuari, els noms dels colors (constructors de la classe Color), l'entorn (menús, halos, botons, etc), el mètode negated (classe Number, traduït com negat), i les següents estructures de control d'Smalltalk: timesRepeat: (classe Integer, traduït com vegadesRepetir:), whileTrue (classe BlockContext, traduït com mentreCert), whileTrue: (classe BlockContext, traduït com mentreCert:), whileFalse (classe BlockContext, traduït com mentreFals), whileFalse: (classe BlockContext, traduït com mentreFals:), repeat (classe BlockContext, traduït com repetir), ifTrue: (classes True i False, traduït com siCert:), ifTrue:ifFalse: (classes True i False, traduït com siCert:siFals:), ifFalse: (classes True i False, traduït com siFals:) i ifFalse:ifTrue: (classes True i False, traduït com siFals:siCert:).

Amb això ja podeu practicar, en català, la major part del contingut del llibre.

Ara bé, l'entorn BotsInc està implementat en Smalltalk dins de l'entorn Squeak (que és una implementació del llenguatge de programació Smalltalk). Per tant, el llibre fa servir moltes eines que són pròpies de l'entorn Squeak general i no estan especialment vinculades a BotsInc. Aquestes eines d'Squeak utilitzades en el llibre no han estat traduïdes. En concret, les eines són el depurador (*debugger*), *Paint* (per fer dibuixos) i el *Transcript*. A més, ocasionalment s'utilitzen

classes que pertanyen a Squeak (i no específicament a BotSInC) que tampoc han estat traduïdes: World, Character, Magnitude, PopUpMenu, FillInTheBlank, String, Time, Date, Rectangle, Point i UndefinedObject.

**Agraïments:** Voldria agrair el suport que he tingut per fer aquesta traducció per part de:

- Stéphane Ducasse, per haver escrit un llibre tan adequat i pels seus esforços en fer accessible lliurement tant l'original com la traducció.
- Citilab: Va ser un projecte que vaig començar amb ells que va motivar aquesta traducció. M'agradaria mencionar especialment en Ramón Sangüesa, per enredar-me, i en Joan Güell i en Josep Garcia, per seguir-me la veta en això de l'Smalltalk.
- Softcatalà i el TERMCAT, per la seva valuosa tasca, que entre d'altres coses m'ha estalviat la feina d'introduir neologismes.
- Pau Fernández, per fer-me descobrir l'Inkscape i per disculpar-me el temps que no li he dedicat com a director de tesi.
- Lali Forcades, per fer que aquest llibre sembli escrit per algú que sap escriure en català.
- La meva família, per tolerar el temps robat.
- Finalment, els estudiants que han volgut venir al curs inspirat per aquest llibre, per deixar-me experimentar i perllongar la seva paciència més enllà del que molts haguessin considerat raonable.

Voldria dedicar aquesta traducció a la Montse, la Marta i el Jaume.

**Jordi Delgado**

Citilab, i Dept. Llenguatges i Sistemes Informàtics (UPC)

jdelgado@lsi.upc.edu

# Pròleg

*El coneixement és només una part de la comprensió.*

*L'autèntica comprensió ve de l'experiència.*

S. Papert

## Objectius i públic lector

L'objectiu d'aquest llibre és explicar conceptes elementals de programació (com l'abstracció, la composició, les repeticions o els condicionals) a novells de totes les edats. Crec que aprendre experimentant i resolent problemes és fonamental en l'adquisició de coneixements. Per tant, introduiré conceptes de programació a través de problemes senzills com ara dibuixar rectangles d'or o simular el comportament animal.

El meu objectiu final és ensenyar-vos programació orientada a objectes, ja que aquest paradigma particular proporciona una metàfora excel·lent per ensenyar a programar. Tot i així, la programació orientada a objectes requereix algunes nocions addicionals de programació i abstracció. Per tant, he escrit aquest llibre per introduir aquests conceptes de programació elementals en un entorn de programació també elemental, amb la idea que sigui el primer volum d'una sèrie de dos. Malgrat això, aquest llibre és completament autosuficient i no requereix que es llegeixi el segon volum. El segon llibre introduceix un altre petit entorn de programació. Posa èmfasi en temes de nivell intermedi com ara trobar un camí en un laberint o dibuixar fractals. També pot jugar el paper de llibre d'ampliació per a lectors que volen saber-ne més o que volen adaptar l'entorn d'aquest llibre a les seves necessitats. Finalment, s'introduceix la programació orientada a objectes.

El lector ideal que tinc al cap és un individu que vol divertir-se programant. Aquest individu pot ser jove o adult, mestre d'escola o algú que ensenya als nens a programar. No cal que sigui hàbil programant en cap altre llenguatge.

El material d'aquest llibre ha estat desenvolupat originalment per a la meva dona, que és professora de física i matemàtiques en una escola francesa amb estudiants d'entre onze i quinze

anys. A finals de 1998, la meva dona es va fer càrrec d'ensenyar informàtica i es va quedar sorpresa per la manca de material adequat. Va començar ensenyant HTML, Word i altres temes però no li van resultar prou satisfactoris, ja que cap d'aquests ensenyaments no promou una actitud *científica* cap a la informàtica. El seu objectiu era ensenyar informàtica com un procés de resolució de problemes.

Com a informàtic, jo era conscient de la feina que s'ha fet amb el llenguatge de programació Logo, i em va agradar la idea de l'experimentació com a base de l'aprenentatge. També era conscient que el llenguatge de programació Smalltalk ha estat influenciat per les idees de Seymour Papert per al llenguatge Logo, i que darrera del seu origen hi va haver força recerca en la manera d'ensenyar a nens a programar. És més, Smalltalk té una sintaxi força senzilla que imita el llenguatge natural. Més o menys per aquelles dates, l'entorn de programació Squeak havia arribat a un estat madur i van començar a aparèixer llibres sobre Squeak, cap a finals de 1999. Aquests, però, estaven destinats a programadors amb experiència, de manera que m'hi vaig posar i vaig escriure aquest llibre que teniu entre mans.

Els entorns que utilitzo en aquest llibre i el que l'acompanya són completament funcionals. Els he anat millorant a partir de les respostes que he rebut de professors i mestres. Ha estat una pauta en la meva feina modificar el mínim possible l'entorn Squeak, ja que el meu objectiu és que els lectors d'aquest llibre ampliïn les idees aquí presents i ells mateixos en desenvolupin de noves.

## **Estructura orientada a objectes i vocabulari**

Els capítols d'aquest llibre són relativament breus, de manera que cada capítol es pot convertir en sessions de laboratori d'una o dues hores. No defensaré la idea d'introduir el material directament als nens per a l'auto-aprenentatge, però cada capítol conté de fet tot el material que cal si es vol fer d'aquesta manera.

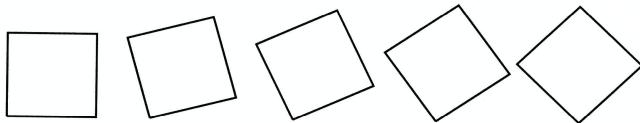
Encara que la programació orientada a objectes no es desenvolupa en aquest llibre, utilitzo el seu vocabulari. És a dir, crearem objectes a partir de classes i els enviarem missatges. El comportament dels objectes ve definit per mètodes. He triat de fer-ho així ja que la metàfora que em proporciona la programació orientada a objectes és força natural, i els nens i nenes tenen una comprensió intutiva dels objectes i del seu comportament.

Aquells que han utilitzat Logo es poden preguntar per quina raó els nostres robots no disposen de l'operació "llapis amunt" i "llapis avall", enlloc de "ves" i "salta", on la primera fa que un robot es mogui deixant una traça i la darrera mou un robot sense deixar cap marca. Crec que el paradigma que ens proporciona "ves" i "salta" s'adqua millor a les idees de la programació orientada a objectes i encapsulament de dades que el tradicional disseny basat en "llapis amunt" i "llapis avall". Una anàlisi excel·lent d'aquestes dues perspectives va ser realitzada per Didier Basset, que va col·laborar amb mi al principi d'aquest projecte.

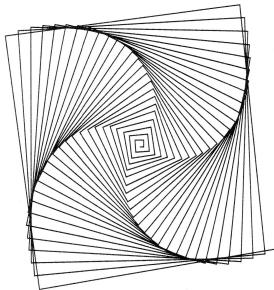
## Organització

El llibre està dividit en cinc parts, tal com està descrit més avall.

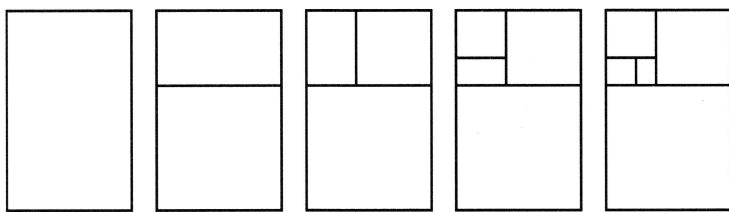
**Començar.** La primera part ens ensenya a engegar l'entorn Squeak. Explica el procés d'instal·lació i d'arrencada d'Squeak, i després introduceix alguns robots amb el seu comportament. Es presenta un primer programa senzill per dibuixar algunes línies.



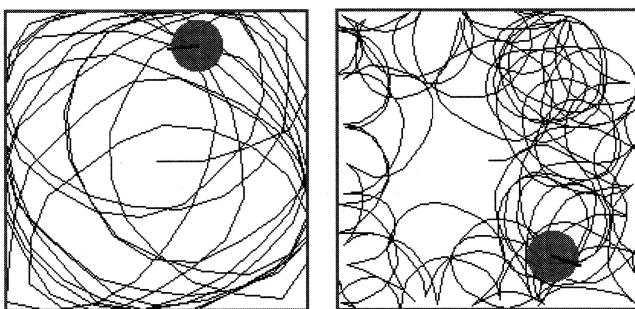
**Conceptes elementals de programació.** La segona part introduceix els primers conceptes de programació, com les repeticions i les variables. Mostra també com es resolen els missatges enviats a robots.



**Posar en joc l'abstracció.** La tercera part introduceix la necessitat de l'abstracció, és a dir, de mètodes o procediments que poden ser reutilitzats per programes diferents. El concepte més difícil que s'introduceix és la idea de com posar nous mètodes a partir de mètodes ja existents per resoldre problemes més complexos. Proposaré diversos experiments no triviais, com ara dibuixar rectangles d'or. També introduirem tècniques i eines per depurar (*debug*) programes.



**Condicionals.** La quarta part introduceix les nocions de condicional, bucle condicional i expressió booleana, totes fonamentals per a la programació. Aquesta part explica també el concepte de referència en un espai bidimensional i altres tipus de comportaments robòtics. Finalment, es presenten maneres d'utilitzar els robots per simular el comportament d'animals simples.



**Altres Mons Squeak.** La cinquena part presenta dos entorns de programació força entretinguts que estan disponibles en Squeak: el sistema d'*scripting* gràfic eToy i l'entorn 3D Alice.

## Per quina raó Squeak i Smalltalk?

Podeu preguntar-vos perquè, d'entre el gran nombre de llenguatges de programació existents avui dia, he triat Smalltalk. Els he triat per les següents raons:

- Smalltalk és un llenguatge molt potent. Podeu construir sistemes molt complexos en un llenguatge que és simple i uniforme.
- Smalltalk va ser dissenyat com un llenguatge per a l'ensenyament. Va inspirar-se en Logo i en Lisp, i va influenciar fortament llenguatges com Java o C#. Tot i així, aquests llenguatges

són massa complexos per a una primera aproximació a la programació. A més, han perdut la bellesa de la simplicitat d'Smalltalk.

- Smalltalk és un llenguatge tipat dinàmicament i això torna irrelevants una sèrie de qüestions relacionades amb tipus i coerció de tipus que són difícils d'explicar i de poc interès pels neòfits.
- Amb Smalltalk només cal aprendre conceptes clau, essencials, que es poden trobar en tots els llenguatges de programació. Així, puc concentrar-me a explicar els conceptes importants sense haver de tractar altres aspectes més difícils i poc atractius dels llenguatges més complexos.
- Squeak és un potent entorn multimèdia, de manera que després de llegir aquests llibres hom podrà construir programes en un entorn realment ric.
- Squeak està disponible gratuïtament i es pot executar en totes les plataformes principals. I hauria de ser fàcilment portable a altres plataformes del futur.
- Squeak és popular. Per exemple, a Espanya<sup>3</sup> s'utilitza a les escoles, on s'executa en uns 80.000 ordinadors.

---

<sup>3</sup>*Nota del traductor:* Ducasse es refereix a la utilització que es fa d'Squeak a Extremadura. Veure <http://squeak.educarex.es/squeakpolis>



# **Part I**

# **Començar**



# Capítol 1

## Instal·lació i creació de robots

Poseu en marxa el vostre cronòmetre! En cinc minuts, l'*entorn* de joc dels robots, que utilitzareu en aquest llibre, estarà executant-se i preparat per a que us divertiu fent-lo servir. En aquest capítol aprendreu com instal·lar l'entorn, a conèixer-ne les diferents parts i començareu a interactuar amb els robots que viuen en aquest entorn. Aprendreu com programar aquests robots per aconseguir que realitzin tasques interessants tot enviant-los *missatges*.

Així, comencem instal·lant l'entorn i preparant-nos pels reptes que vindran. Si el vostre entorn ja està instal·lat, apagueu el cronòmetre, salteu la primera secció i aneu directament a les seccions següents, que fan un resum de l'entorn. Després que hagueu adquirit una mica de soltura amb els robots als capítols 2, 3 i 4, entraré més en detall sobre la utilització de l'entorn al capítol 5.

### Instal·lar l'entorn

L'entorn utilitzat en aquest llibre ha estat desenvolupat per executar-se sobre Squeak. Squeak és un entorn multimèdia *open source* ric i potent escrit completament en Smalltalk i disponible gratuïtament per a la majoria de sistemes operatius a <http://www.squeak.org>. Tingueu en compte, però, que no utilitzarem la distribució Squeak per defecte. Farem servir una distribució que hem preparat per utilitzar amb aquest llibre. Es pot aconseguir de l'editorial que el publica<sup>1</sup>, <http://www.apress.com>, a la secció de *downloads*.

Squeak s'executa exactament igual en totes les plataformes. Malgrat això, per fer-vos la vida més fàcil, he preparat diversos fitxers comprimits que depenen de la plataforma. El principi és exactament el mateix en un Mac, un PC, o qualsevol altra plataforma. Les úniques diferències són les eines que cal fer servir per descomprimir fitxers i la manera en què Squeak arrenca. Un

---

<sup>1</sup>Nota del Traductor: Es refereix al llibre original, publicat en anglès per Apress.

Cop s'ha obtingut un fitxer anomenat Ready[Mac/PC].zip, cal descomprimir-lo i arrosegar el fitxer anomenat Ready.image (Mac) o Ready (PC) sobre l'aplicació Squeak i ja està! El fitxer Ready[.image] conté l'entorn complet utilitzat en aquest llibre. Tingueu en compte que pot ser que els fitxers no es diguin exactament igual, però això no té importància a l'hora de fer-los funcionar<sup>2</sup>.

## Instal·lació en un Macintosh

Per instal·lar l'entorn en un Macintosh hauríeu de tenir un fitxer ZIP anomenat ReadyMac.zip. Usualment, fent doble clic sobre la icona s'hauria d'executar el programa de descompressió corresponent, com per exemple l'*Stuffit Expander*. Un cop aquest arxiu ha estat descomprimit, hauríeu d'obtenir sis fitxers, tal com es mostra en la figura 1.1. Hauríeu d'identificar dos fitxers: el fitxer anomenat Ready.image i el fitxer executable Squeak.app (l'extensió .app pot no aparèixer)



**Figura 1.1 —** Els fitxers preparats per ser utilitzats pel Macintosh. Esquerra: el fitxer ZIP. Dreta: Els fitxers descomprimits.

## Instal·lació en Windows

Per instal·lar l'entorn sobre windows, hauríeu de tenir un fitxer ZIP anomenat ReadyPC.zip. Un cop es descomprimeix aquest fitxer, hauríem d'obtenir vuit fitxers, tal com es veu a la figura 1.2. Hauríeu d'identificar dos fitxers: el fitxer anomenat Ready i el fitxer executable Squeak

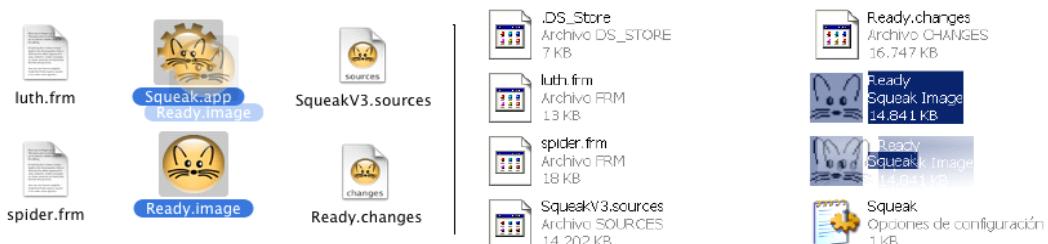
<sup>2</sup>Nota del Traductor: Recordeu que hi ha una versió en català de l'entorn. Només cal utilitzar els fitxers ReadyPC.zip o ReadyMac.zip del servidor mencionat al principi del llibre. Tota la resta de l'explicació que fa el capítol és vàlida, tot i que al final obtindrem la imatge en català. Aquesta és la imatge que utilitzarem en aquest llibre traduït



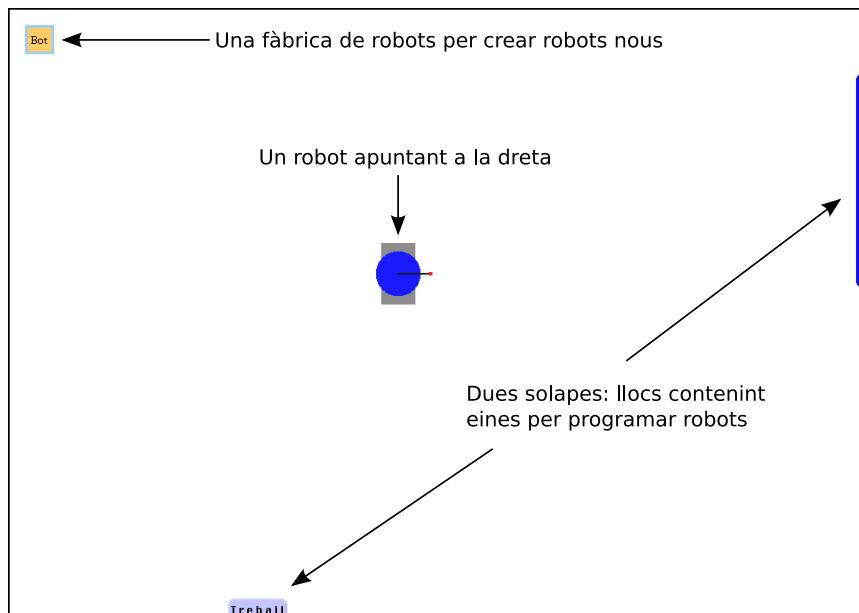
**Figura 1.2** — Els fitxers preparats per ser utilitzats per un PC. Esquerra: el fitxer ZIP. Dreta: Els fitxers descomprimits.

## Obrir l'entorn

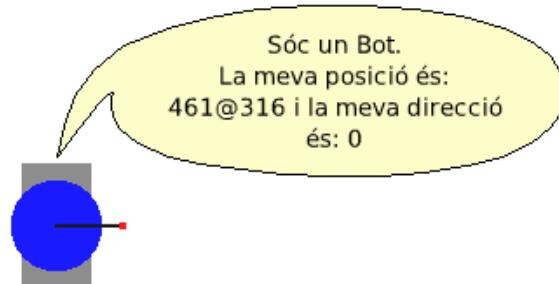
Per obrir l'entorn, arrossegueu el fitxer Ready[.image] sobre la icona del fitxer executable Squeak[.app], com veieu a la figura 1.3. Hauríeu d'obtenir l'entorn tal com es mostra a la figura 1.4. Si no obtenuï aquest entorn, llegiu la secció “Problemes amb la Instal·lació” que trobareu al final d'aquest capítol.



**Figura 1.3** — Arrosegar el fitxer imatge i deixar-lo anar sobre el fitxer executable Squeak, l'entorn s'obre en un Mac (esquerra) o en un PC (dreta).



**Figura 1.4** — L'entorn preparat per ser utilitzat.



**Figura 1.5** — Poseu el ratolí sobre un robot per fer aparèixer una bafarada amb informació sobre el robot

## Ajuts a la instal·lació

L'entorn pot ser obert simplement fent doble clic sobre el fitxer imatge. Això, però, té alguns desavantatges: Heu d'identificar l'aplicació Squeak i de vegades una altra aplicació pot interferir i provar d'utilitzar el fitxer d'imatge. És més, podeu tenir problemes si teniu diverses instal·lacions de diferents versions d'Squeak. Així doncs, us suggereixo que sempre obriu l'entorn arrossegant la imatge sobre l'aplicació Squeak o un algun àlies de l'aplicació. Penseu que si no teniu prou espai per a la instal·lació en el disc dur, podeu fer servir un àlies pel fitxer SqueakV3.sources, que es pot compartir entre diverses instal·lacions.

---

**Important!** Per arrencar l'entorn, arrossegueu el fitxer Ready (amb la extensió .image pel Mac) sobre l'aplicació Squeak.

---

## Primeres interaccions amb un robot

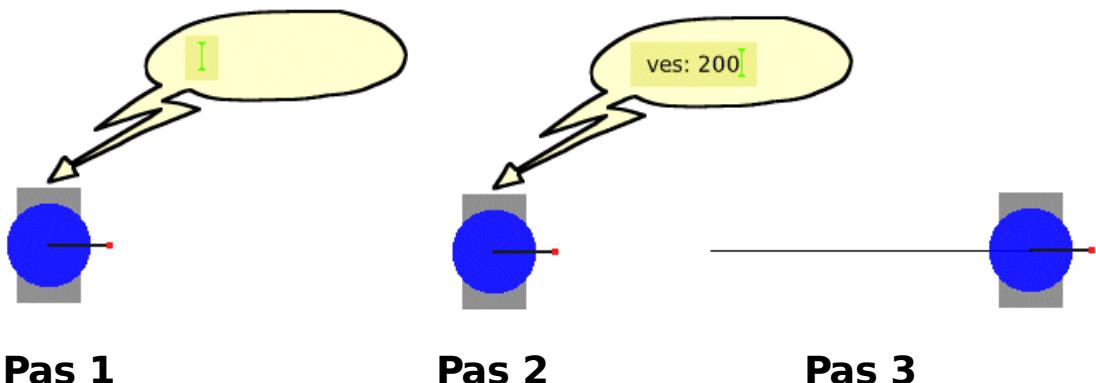
Un cop heu obert l'entorn arrossegant el fitxer anomenat Ready[image] sobre l'aplicació Squeak tal com heu vist abans, l'entorn que s'obté hauria d'assemblar-se al que podem veure a la figura 1.4.

L'entorn està compost d'una fàbrica de robots i dues solapes. Una solapa és com una capsula que conté eines per programar. No les necessitareu de seguida, de manera que ja les descriuré en un altre capítol, més endavant. Hauríeu de veure un petit robot blau al mig de la pantalla. Aquest no és un robot fet de cables i metall sinó un programa-robot, vist des de dalt, apuntant cap al cantó dret de la pantalla. Un robot és una rodona blava; té dues rodes i un petit cap de color vermell que apunta en la seva direcció actual. Més endavant, si aneu seguint el llibre, enviareu ordres als robots. Aquestes ordres s'anomenen *missatges*, i direm que els robots *executen* aquests missatges.

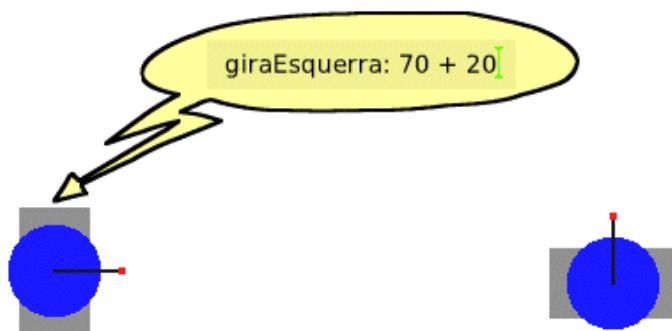
Poseu el ratolí sobre el robot i espereu un segon. Apareix una bafarada amb alguna informació sobre el robot, com la seva posició actual i la seva direcció, tal com veiem en la figura 1.5. Com que els monitors poden ser de diferents mides i resolucions, la posició del vostre robot no ha de ser necessàriament la de la figura.

## Enviar missatges al robot

Podeu interactuar directament amb un robot fent clic amb el botó esquerre del ratolí sobre el robot (o fent només un clic amb un ratolí d'un sol botó). Una bafarada de missatges apareix, tal com veiem a la part esquerra de la figura 1.6. En aquesta bafarada hi podem escriure missatges que s'enviaran al robot. Després d'escriure el missatge, l'envieu al robot prement la tecla *return*, i el robot aleshores l'executa.

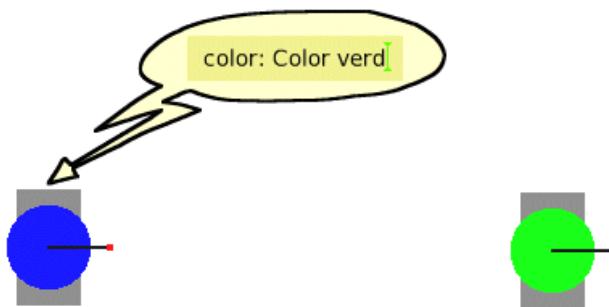


**Figura 1.6** — Pas 1: Fent clic amb el botó esquerre del ratolí sobre el robot, apareix una bafarada de missatges. Pas 2: Podeu escriure un missatge al robot per fer-lo moure 200 píxels endavant, i prémer return. Pas 3: El robot executa el missatge; s'ha mogut, deixant un rastre a la pantalla en forma de línia.



**Figura 1.7** — Esquerra: Enviar un missatge compost. Dreta: El missatge fa que el robot giri a l'esquerra 90 graus.

Per exemple, si hom escriu el missatge `ves: 200` seguit de `return`, el robot rep l'ordre de desplaçar-se endavant 200 píxels en la seva direcció actual. Si hom escriu el missatge `giraEsquerra: 20 + 70`, s'està ordenant al robot que giri cap a l'esquerra (en el sentit contrari al de les agulles del rellotge)  $20 + 70 = 90$  graus, tal com veiem a la figura 1.7. Aquest segon missatge és més complicat que l'anterior, ja que el valor que representa el nombre de graus que el robot ha de girar és en ell mateix un missatge (tal com explicarem ben aviat), és a dir,  $20 + 70$ . Anomenarem a aquests missatges *missatges compostos*. Quan s'envia el missatge `color: Color verd`, el robot canvia de color, tal com veiem a la figura 1.8 (us heu d'imaginar el color verd si la imatge és en escala de grisos).



**Figura 1.8** — Esquerra: S'ordena al robot canviar el seu color a verd. Dreta: El color ha canviat.

Pot ser que no entengueu el format dels missatges que acabem de presentar. Alguns poden semblar una mica complicats. De fet, `color: Color verd` és un altre missatge compost. Explicaré més tard com podeu construir els vostres propis missatges. Ara per ara, simplement escriviu els missatges que anirem presentant de manera que us aneu familiaritzant amb l'entorn dels robots. Si voleu repetir un missatge que ja havíeu escrit, no cal que el torneu a escriure. Utilitzeu les tecles *amunt* i *avall* per navegar pels missatges que ja havíeu enviat al robot. En propers capitols, aprendreu pas a pas tots els missatges que un robot entén, i a més, aprendreu com definir nous comportaments per als vostres robots.

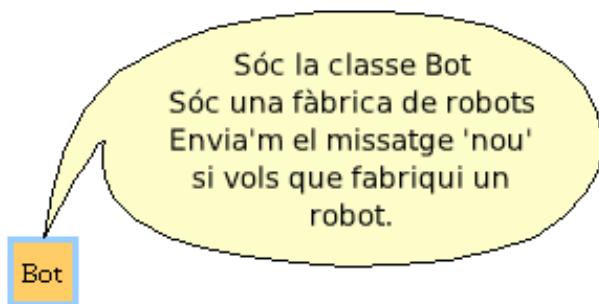
---

**Nota** Per interactuar amb un robot, feu clic al damunt, escriviu un missatge i premeu la tecla `return`

---

## Crear un nou robot

L'entorn ja conté un robot; ara, però, us ensenyaré com crear nous robots. Si no esteu satisfets tenint només un robot, podeu crear-ne un de nou enviant el missatge adequat a la *fàbrica* de robots. Una fàbrica de robots és representada gràficament per una caixa de color taronja envoltada per una caixa de color blau cel, al mig de la qual hi ha escrita la paraula Bot, com veieu a la figura 1.9 . En l'argot d'Squeak, i en general en l'argot de la programació orientada a objectes, una fàbrica de robots és anomenada una *classe*. Les classes (fàbriques que produeixen *objectes*, com els robots) tenen un nom que comença per majúscula. Per tant aquesta és la classe Bot, i no bot.



**Figura 1.9** — En l'argot d'Squeak, una fàbrica de robots s'anomena una *classe*. Les classes produueixen *objectes*. La classe Bot produeix nous robots.

De la mateixa manera que vau fer amb els robots, podeu interactuar amb una fàbrica de robots enviant-li missatges. El missatge per crear un robot nou és el missatge nou, com veieu a la figura 1.10. Cada nou robot, igual que el robot original, apunta cap a la dreta de la pantalla. Cada robot té una existència independent, i podeu enviar missatges a cadascun d'ells.

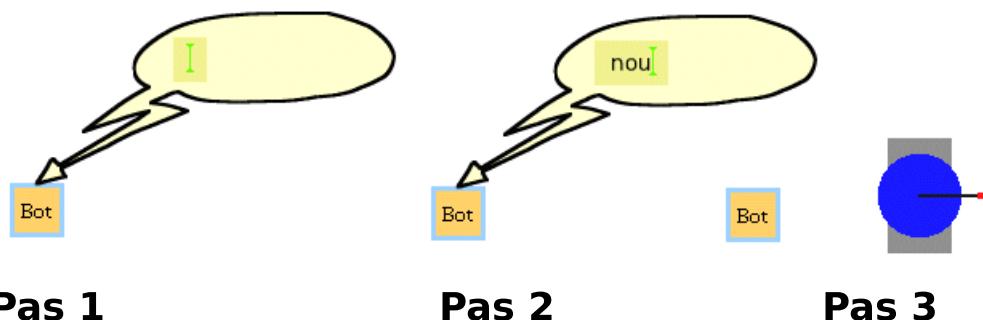
---

Per crear un nou robot, envieu el missatge nou a la fàbrica de robots, que és la classe Bot. Quan es crea un robot, sempre apunta cap a l'est, és a dir, a la dreta de la pantalla.

---

## Sortir i guardar

El fons de la finestra d'Squeak s'anomena el Món (*World*). El Món té un menú que ofereix un cert nombre d'opcions. Per mostrar el menú del Món tan sols heu de fer clic amb el botó esquerre del



**Figura 1.10** — Pas 1: Comenceu a escriure un missatge. Pas 2: El missatge nou s'ha enviat a la fàbrica de robots. Pas 3: Com a resposta, la fàbrica ha creat un robot i us l'ha lliurat.

ratolí sobre el fons de la finestra de l'entorn. Hauríeu d'obtenir un menú similar al que mostro a la figura 1.11. El darrer grup d'opcions són totes aquelles accions que podeu fer servir per sortir de l'entorn o guardar la vostra feina.



**Figura 1.11** — El menú del Món (World) inclou accions per sortir i guardar.

Seleccionar l'opció **sortir** senzillament tanca l'entorn sense guardar res de la vostra feina. El resultat és que la propera vegada que obriu l'entorn, estarà exactament en el mateix estat que la darrera vegada que el vau guardar. Seleccionar l'opció **guardar** guarda tot l'entorn. La propera vegada que obriu l'entorn, estarà exactament en l'estat en que l'acabreu de guardar. Finalment, seleccionar l'opció **guardar com...**, l'entorn us preguntarà per un nom nou, i crearà dos fitxers

nous amb aquest nom: un amb l'extensió `.image` i un altre amb l'extensió `.changes`. Així és com jo vaig crear els fitxers `Ready[image]` i `Ready.changes`. Per obrir l'entorn que heu guardat amb el nou nom, arrossegueu el fitxer amb el nou nom que té l'extensió `.image` sobre l'aplicació Squeak, tal com vau fer al començament per obrir l'entorn arrossegant el fitxer `Ready[image]`.

## Problemes amb la instal·lació

Algunes vegades les coses no van com podríem esperar, així que en aquesta secció donaré informació que us pot ajudar si teniu problemes amb la instal·lació. Primer, explicaré el rol dels principals fitxers que us heu trobat en descomprimir el fitxer descarregat.

Per executar l'entorn proporcionat amb aquest llibre, o amb qualsevol altra distribució Squeak, són necessaris quatre fitxers. Saber-ne alguna cosa pot ajudar-vos a resoldre alguns dels problemes que podríeu trobar.

**Imatge i canvis.** El fitxer `Ready[image]`, anomenat simplement el fitxer *imatge*, i el fitxer `Ready.changes`, anomenat simplement el fitxer de *canvis*, contenen informació sobre el vostre sistema Squeak actual. Aquests dos fitxers estan sincronitzats automàticament per Squeak i haurien de tenir els permisos d'escriptura habilitats. Cada cop que guardeu l'entorn, aquests dos fitxers es sincronitzen. No els hauríeu d'editar mai amb un editor de textos o canviar-los el nom manualment. Si voleu utilitzar diferents noms, utilitzeu l'opció **guardar com...** del menú del Món. Squeak crearà un nou parell de fitxers per a vosaltres.

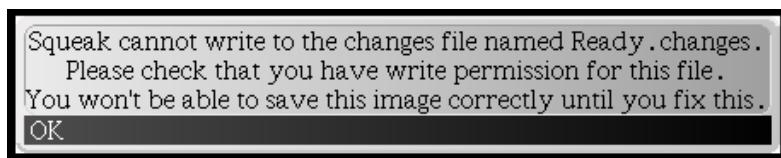
**Fonts.** El fitxer anomenat `SqueakV3.sources`, anomenat el fitxer dels *fons*, conté el codi font de part de l'entorn Squeak. No el necessitareu per estudiar aquest llibre, de manera que no proveu d'editar-lo manualment. Tot i així, aquest fitxer hauria d'estar sempre al mateix directori on teniu el fitxer *imatge*.

**Aplicació.** El fitxer executable `Squeak[.app]` per Mac, o `Squeak.exe` pel PC, és l'aplicació Squeak. Aquest fitxer és l'aplicació que s'executa quan esteu programant en Squeak. Hauria de tenir el permís d'execució habilitat. Aquest fitxer s'anomena l'aplicació Squeak. En l'argot de la informàtica, aquesta aplicació s'anomena una *màquina virtual*, o MV.

Recordeu que els fitxers d'*imatge* i de *canvis* haurien de tenir els permisos d'escriptura habilitats. Alguns sistemes operatius canvien les propietats dels fitxers a “només lectura” quan són copiats des de fonts externes. Si passa això, Squeak us avisa amb un missatge en anglès<sup>3</sup>, com el que us ensenyem a la figura 1.12. Si us surt aquest missatge, sortiu d'Squeak sense guardar, canvieu la propietat del fitxer per permetre l'escriptura, i tornieu a obrir l'entorn.

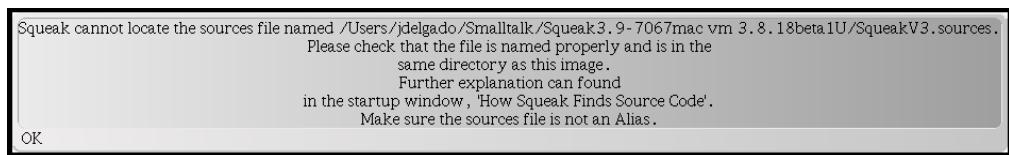
---

<sup>3</sup>Nota del Traductor: L'Squeak amb què treballeu és un sistema força gran i, encara que pràcticament tot l'entorn BotsInc que fareu servir amb aquest llibre ha estat traduït, és molt possible que de tant en tant us apareguin missatges en anglès que pertanyen a l'entorn Squeak general.



**Figura 1.12** — Aquest missatge apareix si la imatge (Ready[.image]) o el fitxer de canvis (Ready.changes) no tenen els permisos d'escriptura habilitats.

Un altre problema que podeu trobar està relacionat amb el fitxer dels fonts SqueakV3.sources. Aquest fitxer, o un àlies referenciant-lo, hauria d'estar present al directori de la imatge. Si el fitxer no hi és, podeu trobar-vos amb el missatge (també en anglès) de la figura 1.13. Per resoldre aquest problema, creeu un àlies al fitxer SqueakV3.sources dins el directori que conté el fitxer de la imatge o simplement copieu el fitxer SqueakV3.sources dins aquest directori. No hauríeu de tenir aquest problema si esteu utilitzant la distribució feta per a aquest llibre.



**Figura 1.13** — Missatge que indica que el fitxer dels fonts (SqueakV3.sources) no és al directori que conté el fitxer de la imatge.

## Resum

Per obrir l'entorn, arrossegueu el fitxer Ready[.image] o algun altre fitxer que hagueu guardat amb l'extensió .image dins l'aplicació Squeak.

- Per enviar un missatge a un robot, feu clic sobre el robot amb el botó esquerre del ratolí, escriviu el missatge i premeu *return*.
- Per crear un robot nou, envieu el missatge nou a la classe Bot, que és la vostra fàbrica de robots.
- Quan es crea un robot, sempre apunta a l'est, és a dir, a la dreta de la pantalla.
- Per obtenir el menú per guardar l'entorn feu clic en el fons de la finestra de l'entorn.

## Capítol 2

# Un primer *script* i les seves implicacions

Enviar missatges als robots via interacció directa és una manera divertida i potent de programar-los, però és bastant limitada com a tècnica per escriure programes complicats. Per expandir els horitzons de les vostres possibilitats a l'hora de programar els robots, us ensenyaré la noció d'*script*, que és una seqüència d'expressions, juntament amb els conceptes fonamentals i el vocabulari que necessitareu a la resta del llibre. Aquest capítol també servirà de mapa per als propers capítols, que introduiran en profunditat els conceptes breument presentats aquí.

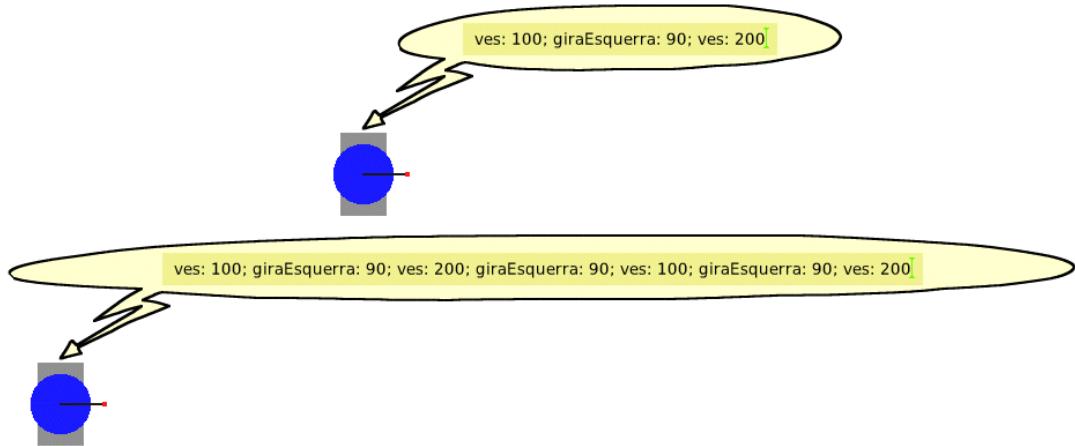
Primer, us mostraré com enviar múltiples missatges al mateix robot, tot separant una seqüència de missatges amb punts i coma. Després aprendreu com escriure un *script* utilitzant una eina anomenada *Espai de Treball* (*workspace*). Explicaré els elements diferents que componen un *script* i mostraré alguns dels errors que hom pot cometre quan escriu un programa.

### Utilitzar una cascada per enviar múltiples missatges

Imagineu que voleu fer que el robot que teniu en pantalla dibuixi un rectangle d'alçada 200 píxels i amplada 100 píxels. Per fer això, podríeu fer clic sobre el robot i començar a escriure el primer missatge, ves: 100, premeu la tecla *return*, després feu clic i escriviu la segona expressió, *giraEsquerra*: 90 i premeu la tecla *return*, aleshores torneu a fer clic sobre el robot i escriviu ves: 200 i així successivament. Ràpidament us adoneu que interactuar així amb el vostre robot és força tediós. Seria molt més convenient si poguéssiu escriure totes les instruccions de cop i executar-les totes només prement un botó.

De fet, podeu enviar múltiples missatges a un robot separant els missatges amb un punt

i coma (;). Per enviar els missatges ves: 100, giraEsquerra: 90 i ves: 200 a un robot, escriuï-los separats per punts i coma, ves: 100; giraEsquerra: 90; ves: 200 (veure figura 2.1). Aquesta manera d'enviar múltiples missatges a un mateix robot s'anomena *cascada de missatges* en l'argot d'Squeak.



**Figura 2.1** — Podeu enviar d'una sola vegada diversos missatges a un robot utilitzant el caràcter punt i coma (;).

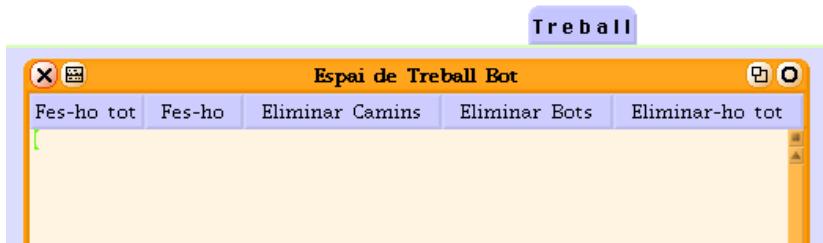
Malgrat tot, la tècnica d'escriure una cascada de missatges (vull dir, enviar a un robot múltiples missatges separats per punts i coma) no funciona bé per a programes complexos. De fet, fins i tot per dibuixar un simple rectangle, la seqüència de missatges ràpidament creix fins a ser massa llarga, com podeu veure en el segon missatge de la figura 2.1. I hi ha altres problemes. Per exemple, els programadors tenen en consideració aspectes com guardar una seqüència de missatges per executar-la més tard, o bé reutilitzar els seus missatges sense haver-los d'escriure altre cop. Per aquestes raons, necessiteu alguna altra manera de programar robots. La primera manera que aprendreu és a escriure una seqüència de missatges, anomenada un *script*<sup>1</sup>, en un editor de text i demanar a l'entorn que executi el vostre *script*.

---

<sup>1</sup>Nota del Traductor: Per manca d'una traducció satisfactòria de la paraula anglesa "script" (que voldria dir "petit programa"), la deixarem en anglès i en cursiva, per deixar clar que és un anglicisme.

## Primer *script*

L'entorn Bot proporciona un petit editor de text, anomenat l'Espai de Treball Bot (*Bot Workspace*), que està pensat per a l'execució de programes. Feu clic a la solapa que trobareu a la part de sota, anomenada Treball. Per defecte conté un editor Espai de Treball Bot, com podeu veure a la figura 2.2.



**Figura 2.2** — L'Espai de Treball Bot és un petit editor de text dedicat a l'execució de programes per als robots.

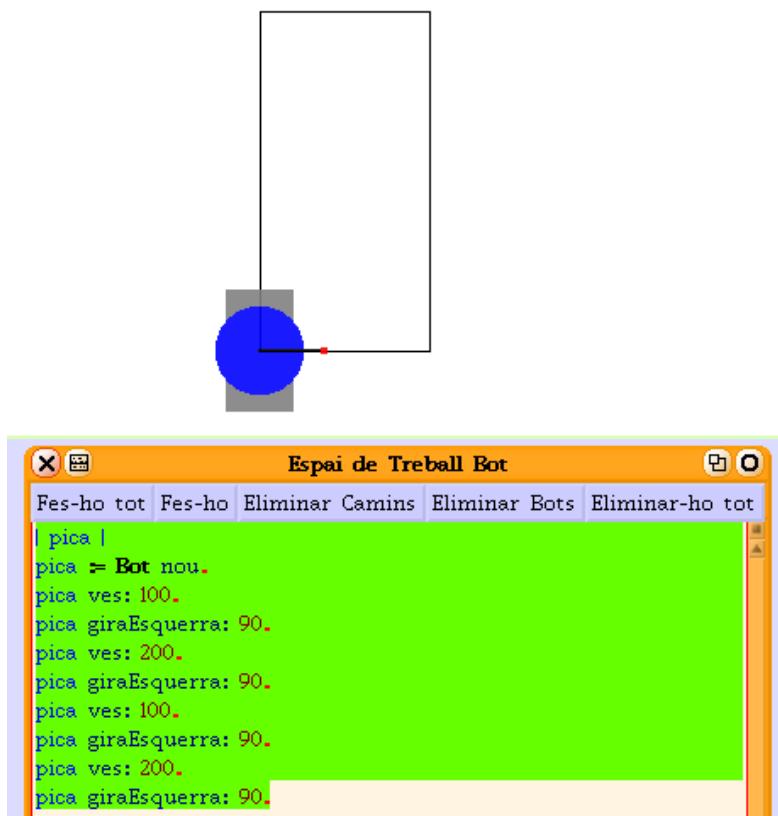
Començarem escrivint un *script* que dibuixa un rectangle, i després l'explicarem en detall (*Script 2.1*).

**Script 2.1** Creem el robot pica i el fem moure i girar

```
| pica |
pica := Bot nou.
pica ves: 100.
pica giraEsquerra: 90.
pica ves: 200.
pica giraEsquerra: 90.
pica ves: 100.
pica giraEsquerra: 90.
pica ves: 200.
pica giraEsquerra: 90.
```

La figura 2.3 mostra l'*script* en un Espai de Treball Bot i el resultat de l'execució, obtingut prement el botó **Fes-ho tot**. Intenteu aconseguir el mateix resultat: escriviu l'*script* i premeu el botó **Fes-ho tot**. He anomenat el robot Pica com a abreviatura de Picasso, ja que els nostres robots dibuixen, igual que ho feia el gran pintor espanyol.

El botó **Fes-ho tot** de l'Espai de Treball Bot executa *tots* els missatges que conté l'espai de treball. Per tant, abans d'escriure un *script*, assegureu-vos que no hi ha cap altre text escrit en l'espai de treball. És més, els ordinadors i els llenguatges de programació no poden tractar ni el més mínim error, per obvi que sigui, així doncs, tingueu molta cura d'escriure el text exactament com està escrit en l'*Script* 2.1. Per exemple, heu d'escriure la majúscula "B" de Bot a la segona línia, i heu d'acabar cada línia amb un punt (no cal posar el punt al final de la darrera línia, ja que els punts *separen* missatges en Squeak. Tampoc cal un punt després de la primera línia, ja que no conté cap missatge). Ho ampliarem més endavant en aquest capítol. Podeu veure l'*script* i el resultat d'executar-lo a la figura 2.3.



**Figura 2.3** — Un *script* executat utilitzant el botó **Fes-ho tot** de l' Espai de Treball Bot i el resultat de l'execució.

## Squeak i Smalltalk

L'*Script* 2.1 és molt senzill, tot i així és un autèntic programa informàtic. Un *programa* és una llista d'*expressions* que un ordinador pot executar. Per escriure programes necessitem llenguatges de programació, és a dir, llenguatges que permeten als programadors escriure instruccions que un ordinador pot “entendre” i executar.

### Llenguatges de programació

Un llenguatge de programació ben dissenyat serveix per ajudar els programadors a expressar solucions als seus problemes. Per ajuda vull dir que el llenguatge hauria de, entre d’altres coses, facilitar l’expressió de la tasca a realitzar, proporcionar una execució eficient del codi del programa i confiança en el resultat, donar al programador la capacitat de demostrar que els seus programes són correctes, promoure la producció de codi lleigible i facilitar als programadors fer canvis en les aplicacions. No existeix ni l’ideal, ni el “millor” llenguatge de programació que satisfaci totes aquestes propietats tan desitjables, i llenguatges de programació diferents estan millor adaptats a diverses tasques.

### Smalltalk i Squeak

Aquest llibre us ensenyàrà com programar en el *llenguatge de programació* Smalltalk dins de l'*entorn de programació* Squeak. Un entorn de programació és un conjunt d’eines que els programadors utilitzen per desenvolupar aplicacions. Squeak conté un gran nombre d’eines útils: editors de text, exploradors de codi (*code browsers*), depurador (*debugger*), inspector d’objectes, compilador, *widgets*, i molts altres. I això no és tot! En l’entorn Squeak podeu programar música, fitxers *flash* animats, accedir a l’Internet, mostrar objectes 3D i molt més. Malgrat tot, abans no pugueu programar aplicacions sofisticades heu d’aprendre alguns principis bàsics, i aquest és el propòsit d’aquest llibre.

Els programadors Squeak desenvolupen les seves aplicacions escrivint programes en el llenguatge de programació Smalltalk. Smalltalk és un llenguatge de programació *orientat a objectes*. Altres llenguatges orientats a objectes són Java i C++, però Smalltalk és el més pur i el més senzill. Tal com suggereix el terme “orientat a objectes”, aquest llenguatge de programació utilitza *objectes*. Els objectes que són creats i utilitzats no són, naturalment, objectes reals sinó estructures lògiques, o objectes “virtuals”, dins l’ordinador. Són anomenats objectes perquè és útil pensar en aquestes estructures com a objectes reals manufacturats, com un robot, que és capaç d’entendre els missatges que rep i executar les instruccions que estiguin contingudes en aquests missatges. La raó de l’analogia amb els objectes és que podem utilitzar un robot, o una ràdio, o una càmera, sense conèixer la seva estructura interna. Només ens cal saber com utilitzar-los prement botons o enviant-los missatges via comandaments a distància.

D'on venen els objectes reals manufacturats? D'una fàbrica, naturalment. Les fàbriques d'objectes s'anomenen *classes* en els llenguatges de programació orientats a objectes. Definir classes és una mica complicat, igual que la programació orientada a objectes en general, de manera que en aquest llibre introductorí no us ensenyaré a definir classes. Enlloc d'això, el que acabarem fent és definir nous comportament pels nostres robots, i això ens donarà una bona base en els conceptes elementals de la programació.

He triat Smalltalk com a llenguatge d'aquest llibre perquè és simple, uniforme i pur. Diem que és pur ja que a Smalltalk, *tot* és un objecte que envia i rep missatges a i des d'altres objectes. Smalltalk és simple ja que només hi ha unes poques regles bàsiques, i és uniforme ja que aquestes regles s'apliquen sempre de manera consistent. De fet, Smalltalk va ser dissenyat originalment per ensenyar als neòfits a programar. Això, però, no significa que només puguem utilitzar Smalltalk per escriure aplicacions "de joguina". Hi ha aplicacions grans i complexes que han estat escrites en Smalltalk, com les aplicacions que controlen les màquines que fan els xips AMD que potser duu el vostre ordinador.

Una altra aplicació escrita completament en Smalltalk és l'entorn Squeak mateix. No us sembla interessant? Això significa que un cop entengueu bé el llenguatge Smalltalk, podeu modificar l'entorn Squeak per adaptar-lo a allò que vulgueu fer o simplement per aprendre més sobre el sistema. Així doncs, amb Smalltalk teniu un gran potencial a les vostres mans.

Espero que aquesta discussió sobre llenguatges de programació en general, i Smalltalk en particular, us hagi motivat prou com per voler aprendre a programar. Si us plau, sigueu conscients que aprendre a programar és com aprendre a tocar el piano o pintar a l'oli. No és simple, per tant no us desanimeu si trobeu algunes dificultats. Així com un pianista principiant no comença amb la sonata de Beethoven *Waldstein Sonata*, i un pintor principiant no intenta reproduir el sostre de la capella sixtina de Michelangelo, un programador novell comença amb tasques senzilles. He dissenyat aquest llibre de manera que els continguts siguin introduïts en un ordre lògic i el que apreneu en cada capítol necessita del que heu après en capítols anteriors i us prepara per al que vindrà en capítols posteriors.

## Programes, expressions i missatges

Ara estem preparats per fer una ullada als detalls del vostre primer *script*.

### Escriure i executar programes

Quan vau escriure l'*Script 2.1*, vau escriure un text, constituint una sèrie d'expressions, i vau demanar a Squeak que l'executés tot prement el botó **Fes-ho tot**. Squeak va executar la seqüència d'expressions; això és, va transformar la representació textual del vostre programa en una forma comprensible per a l'ordinador, i aleshores cada expressió va ser executada en sèrie. En aquest

primer *script*, executar la seqüència d'instruccions va crear un robot anomenat pica i pica va executar, un darrera l'altre, els missatges que li vau enviar.

Un programa en Squeak consisteix en una sèrie d'*expressions* que han de ser executades per l'entorn Squeak. En aquest llibre anomenarem a aquesta seqüència un *script*.

---

**Important!** Un *script* és una seqüència d'*expressions*.

---

Un programa és com una recepta per a un pastís de xocolata. Una bona recepta descriu tots els passos que s'han de dur a terme en la seqüència correcta: barrejar els rovells d'ou amb sucre, fondre la cobertura de xocolata al bany maria, afegir mantega i remenar, muntar les clares i barrejar-ho tot, remenar-ho i posar-ho en un motlló per al forn; deixar cuure al forn a 180 graus durant 30 minuts; deixar refredar i servir. I bon profit! Igualment, un programa informàtic descriu tots els passos en seqüència per aconseguir l'efecte desitjat: declarar (triar) el nom per a un robot; crear un robot amb aquest nom; dir-li al robot que es mogui 100 píxels; dir-li al robot que giri 90 graus; etcètera.

## Anatomia d'un *script*

Ja és hora d'analitzar el nostre primer *script*, que reproduïm aquí com a *Script 2.2*.

### Script 2.2

```
| pica |
pica := Bot nou.
pica ves: 100.
pica giraEsquerra: 90.
pica ves: 200.
pica giraEsquerra: 90.
pica ves: 100.
pica giraEsquerra: 90.
pica ves: 200.
pica giraEsquerra: 90.
```

Resumint, l'*Script 2.2* comença declarant que utilitzarà una *variable* anomenada pica per referir-se al robot que crea. Un cop s'ha creat el robot i està associat amb la variable pica, l'*script* li diu al robot que faci una seqüència específica de moviments a diferents llocs de la pantalla mentre

gira 90 graus a l'esquerra després de cada moviment. Analitzem cada línia pas per pas. No us amoïneu si alguns conceptes com la noció de variable no queden clars. Tots ells seran explicats quan toqui, si no en aquest capítol, en un de posterior.

`| pica |:` Aquesta primera línia declara una *variable*. Diu a Squeak que volem utilitzar el nom pica per referir-nos a un objecte. Penseu-ho com si diguéssiu a un amic: "d'ara en endavant utilitzaré la paraula pica en les meves frases per referir-me al robot que estic a punt d'encarregar a la fàbrica de robots". Aprendreu més sobre variables al capítol 8.

`pica := Bot nou.:` Aquesta línia crea un robot nou enviant el missatge nou a la fàbrica (classe) de robots i associa el robot amb el nom pica, la variable declarada en el pas anterior. La paraula Bot requereix una B majúscula ja que és una classe, en aquest cas la classe (fàbrica) per produir robots.

`pica ves: 100.:` En aquesta expressió, el missatge ves: 100 és enviat al robot anomenat pica. Aquesta línia pot ser entesa com: "pica, mou-te 100 unitats pel monitor de l'ordinador". És implícit en el missatge que el robot que rep un missatge ves: sap en quina direcció ha d'anar. De fet, un robot sempre apunta en alguna direcció, i quan rep un missatge ves:, sap que s'ha de moure en la direcció que està apuntant. Fixeu-vos que el missatge ves: acaba amb dos punts. Això vol dir que aquest missatge necessita informació addicional, en aquest cas una longitud. Per exemple, ves: 100 diu que el robot s'hauria de moure 100 píxels. El nom del missatge és ves:.

`pica giraEsquerra: 90.:` Aquesta línia li diu a pica que giri 90 graus a la seva esquerra (en sentit contrari a les agulles del rellotge). Aquesta línia és un altre missatge enviat al robot anomenat pica. El nom del missatge giraEsquerra: acaba en dos punts, i per tant cal donar informació addicional, en aquest cas un angle.

La resta de línies de l'*script* són similars.

**Important!** Qualsevol missatge que acaba amb dos punts indica que el missatge necessita informació addicional, com una longitud o un angle. Per exemple, el nom del missatge giraEsquerra: necessita un nombre que representi l'angle que el robot ha de girar en sentit contrari a les agulles del rellotge.

## Sobre els píxels

La unitat de distància en una pantalla d'ordinador s'anomena *pixel*. Aquesta paraula va ser inventada cap al 1970 i és un acrònim de *picture element*. Un pixel és la mida del punt més petit

que es pot dibuixar a la pantalla d'un ordinador. La mida del pixel pot variar en funció del tipus de monitor. Es poden veure els píxels individuals mirant la pantalla amb una lupa.

## Expressions, missatges i mètodes

Hem estat utilitzant els termes *expressió* i *missatge*. Ara és el moment de definir-los. També definirem el terme important *mètode*.

### Expressió

Una *expressió* és qualsevol element amb significat d'un programa. Aquí teniu alguns exemples d'*expressions*:

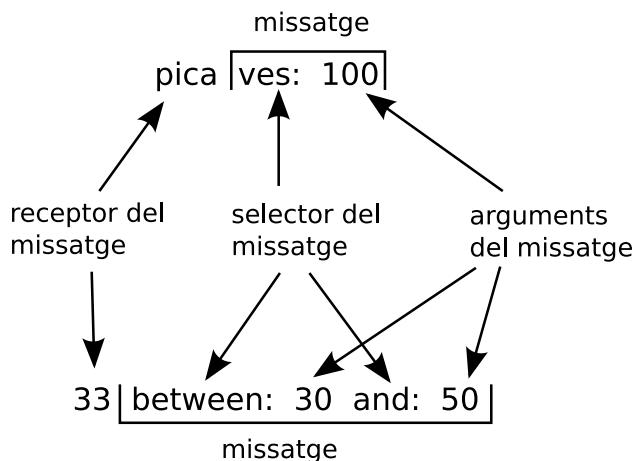
- | pica | és una expressió que declara una variable (més al capítol 8).
- pica := Bot nou. és una expressió amb una operació, anomenada *assignació*, que associa un valor amb una variable (veure capítol 8). Aquí un robot, acabat de crear en haver enviat el missatge nou a la classe Bot, s'associa amb la variable pica.
- pica ves: 100. és una expressió que envia un missatge a un objecte. Aquesta expressió s'anomena *enviament de missatge*. El missatge ves: 100 és enviat a l'objecte anomenat pica.
- 100 + 200 és també un enviament de missatge. El missatge + 200 és enviat a l'objecte 100.

### Missatge

Un *missatge* és una parella composada pel nom del missatge, també anomenat *selector del missatge*, i els possibles *arguments* del missatge, que són els valors que l'objecte que rep el missatge necessita per poder executar-lo. Aquestes relacions estan il·lustrades a la figura 2.4. L'objecte que rep el missatge s'anomena el *receptor del missatge*. Un missatge, juntament amb el receptor del missatge és el que anomenem un *enviament de missatge*. Aquí teniu alguns exemples de missatges:

- A l'expressió pica tornarInvisible, el missatge tornarInvisible s'ha enviat a un receptor, en aquest cas un robot. Aquest missatge no té cap argument.
- A l'expressió pica ves: 100, el missatge ves: 100 ha estat enviat a un receptor, un robot anomenat pica. Està compost del selector del missatge ves: i un sol argument, el nombre 100. Aquí, 100 representa la distància en píxels que el robot hauria de recórrer. Fixeu-vos que els dos punts formen part del selector del missatge.
- A l'expressió 33 between: 30 and: 50 el missatge between: 30 and: 50 està compost del selector del missatge between: and: i de dos arguments 30 i 50. Aquest missatge demana al receptor, aquí el nombre 33, si està entre dos valors, aquí els nombres 30 i 50.

- A l'expressió 4 vegadesRepetir: [pica ves: 100], el missatge vegadesRepetir: [pica ves: 100], que ha estat enviat al nombre 4, es compon del selector del missatge vegadesRepetir:<sup>2</sup> i de l'argument [pica ves: 100]. Aquest argument s'anomena un *bloc*, que és una seqüència d'expressions (en aquest cas n'hi ha només una) dins d'un parell de claudàtors (més al capítol 7).
- A l'expressió 100 + 200, el missatge + 200 es compon del selector de missatge + i un argument, el nombre 200. El receptor és el nombre 100.



**Figura 2.4** — Dos enviaments de missatges compostos d'un receptor del missatge, un nom de missatge (o selector del missatge) i uns arguments.

## Separació de missatges

Tal com hem dit abans, cada línia de l'*Script* 2.1, excepte la primera i l'última, s'acaba amb un punt. La primera línia no conté cap missatge. Una línia com aquesta s'anomena una *declaració de variables* en argot informàtic. Així, podem fer la següent observació: cada missatge enviat ha de separar-se del següent amb un punt. Fixeu-vos que posar el punt després del darrer missatge és possible però no obligatori. Smalltalk ho accepta de les dues maneres.

<sup>2</sup>Nota del Traductor El mètode `between: and:` pertany al llenguatge Smalltalk general més que a l'entorn dels bots preparat per a aquest llibre. Per això no ha estat traduït. Hi ha, però, alguns mètodes d'Smalltalk que no pertanyen pròpiament a l'entorn dels bots que *sí* que hem traduït, com per exemple `vegadesRepetir:`, que és `timesRepeat:` en Smalltalk. Això ho hem fet essencialment amb mètodes molt utilitzats i amb algunes estructures de control.

---

**Important!** Els enviaments de missatges haurien de separar-se amb un punt. El darrer enviament no requereix un punt. Aquí teniu quatre missatges separats per tres punts.

```
pica := Bot nou.  
pica ves: 100.  
pica giraEsquerra: 90.  
pica ves: 100
```

---

---

**Important!** El caràcter punt . és un separador de missatges, de manera que no cal posar-ne un en acabar un enviament de missatge si no hi ha un altre enviament de missatge després. Per tant, no cal cap punt al final d'un *script* o d'un bloc de missatges.

---

## Mètode

Quan un robot (o un altre objecte) rep un missatge, executa un *mètode*, que és una mena d'*script* amb nom. Més formalment, un *mètode* és una seqüència d'expressions amb nom que un objecte receptor executa en resposta a la recepció d'un missatge. Un *mètode* és executat quan un objecte rep un missatge del mateix nom que un dels seus *mètodes*. Per exemple, un robot executa el seu *mètode* ves: quan rep un missatge el nom del qual és ves:. Així doncs, l'expressió pica ves: 224 provoca que el receptor del missatge, pica, executi el seu *mètode* ves: amb argument 224, amb el resultat d'efectuar un moviment de 224 píxels endavant en la direcció actual. Més endavant explicarem com podeu definir nous *mètodes* per als vostres robots, ara per ara no cal saber-ho per començar a programar.

## Cascada

Tal com vam mencionar en la primera secció d'aquest capítol, podeu enviar múltiples missatges a un robot separant-los amb punts i coma. Aquesta seqüència de missatges s'anomena una *cascada*. També podeu utilitzar una cascada en un *script* per enviar diversos missatges a un robot. L'*Script* 2.3 és equivalent a l'*Script* 2.2, excepte que ara tots els missatges enviats al robot pica estan separats per punts i coma. Utilitzar cascades és pràctic quan voleu evitar escriure repetides vegades el nom del receptor dels múltiples missatges. Les cascades són útils ja que fan els *scripts* més curts. Però aneu amb compte! Les dreceres poden donar-vos problemes si no mireu per on aneu; així doncs, assegureu-vos que realment voleu enviar tots els missatges al mateix receptor.

### Script 2.3

```
| pica |
pica := Bot nou.
pica
  ves: 100 ; giraEsquerra: 90 ; ves: 200 ; giraEsquerra: 90 ;
  ves: 100 ; giraEsquerra: 90 ; ves: 200 ; giraEsquerra: 90 .
```

**Important!** Per enviar múltiples missatges a un robot, utilitzeu el caràcter punt i coma ; per separar els missatges, seguint l'estructura *unBot missatge1 ; missatge2*. Un exemple: pica ves: 100 ; giraEsquerra: 90 ; ves: 200 ; giraEsquerra: 90 .

### Crear nous robots

Per aconseguir un robot nou heu d'encarregar-lo a la fàbrica de robots. És a dir, heu d'enviar el missatge nou a la classe Bot. Això ja ho sabíem. És exactament el que vau fer en el capítol anterior quan vau fer clic sobre la capseta blava i taronja anomenada Bot, que representa la classe amb aquest nom, i vau escriure nou a la bafarada. A Squeak, sempre cal enviar missatges als robots, altres objectes o classes per interactuar amb ells. No hi ha cap diferència en el tractament, excepte que els objectes i les classes entenen diferents missatges. La feina de les classes és crear objectes. Un objecte no sap com crear altres objectes, de manera que obtenim un error si enviem el missatge nou a un robot. Les classes, per altra part, generalment no tenen colors i no saben moure's, per tant enviar missatges com color o ves: 135 a una classe no té gaire sentit, i obtenim un error en fer-ho. Malgrat tot, en ambdós casos estem enviant missatges!

La classe Bot no és l'única fàbrica d'objectes en l'entorn Squeak. Hi ha altres classes, que entenen diferents missatges i utilitzen mètodes diferents per crear tipus diversos d'objectes. Per exemple, la classe Color fabrica colors (objectes-color). Retorna un color blau o verd en resposta al missatge blau o verd. En aquest llibre, quan calgui obtenir un objecte a partir d'una classe específica, ja us direm com fer-ho.

---

**Important!** Per obtenir un objecte nou a partir d'una classe, usualment s'envia el missatge nou a la classe<sup>3</sup>. Així, Bot nou crea un robot nou. Altres classes poden oferir diferents missatges per crear objectes nous. Per exemple, Color blau diu a la classe Color que ha de crear un objecte color blau nou

---

## Errors als programes

Els ordinadors són molt bons fent càlculs molt complexos a velocitats increïbles, però no tenen intel·ligència per corregir petits errors. Si escriguéssim accidentalment “ara, engega el teu ordinador” podríeu riure amb l'error comès, però no tindríeu cap problema entenent el que volem dir. Els ordinadors, però, no tenen aquest tipus d'intel·ligència, i això vol dir que cada expressió que vulguem proporcionar a un ordinador ha de ser introduïda de manera precisa, sense cap error. L'error més petit, per insignificant que ens sembli, fins i tot quelcom tan trivial com canviar una lletra minúscula per una majúscula, no serà entés per l'ordinador. Si teniu errors en els vostres *scripts*, dues coses poden anar malament: o bé apareixerà per pantalla un missatge d'error, i això és probable que passi en els vostres primers experiments, o bé el programa serà executat però els resultats no seran els que volíeu. Així, quan les coses van malament, no desespereu i proveu de trobar l'error al vostre programa.

Squeak té eines força útils per corregir i prevenir errors. Acoloreix les lletres mentre escriuviu. Quan una paraula es torna vermella significa que esteu escrivint alguna cosa que Squeak no entén. Podeu veure un exemple a la figura 2.5. Si una paraula és blava (per a un nom de variable o de missatge) o negre (per a un nom de classe) vol dir que tot és estructuralment correcte.

Si intenteu executar una expressió que conté un error, Squeak intenta ajudar-vos, notificant-vos-ho quan troba l'error al vostre codi. Els missatges d'error que Squeak fa servir són menús. La part de dalt de la finestra del menú conté una petita descripció de l'error; després, depenent del tipus d'error, se suggeren algunes correccions en forma de llista d'opcions. Si no voleu triar cap de les opcions, sempre podeu cancel·lar l'execució triant “cancel·lar” al menú. Aleshores, hauríeu de trobar què és el que Squeak no entén del vostre *script*, corregir-ho i provar d'executar l'*script* altre cop.

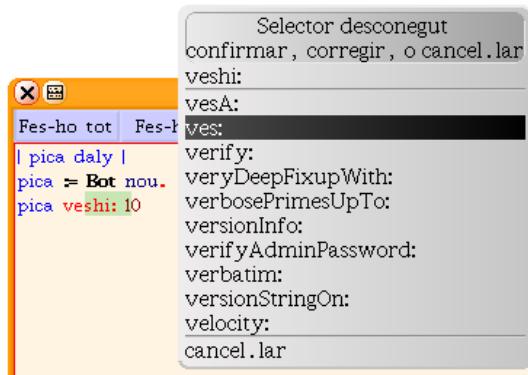
Ara us explicarem alguns dels errors més comuns.

---

<sup>3</sup>Nota del Traductor: En realitat el missatge que usualment cal enviar a una classe per crear objectes dins de l'entorn Squeak general és new. Com que aquí estem treballant amb l'entorn més petit dels robots, el missatge new per a la classe Bot ha estat traduït i per crear nous robots podem enviar el missatge nou.

## Escriure malament un selector de missatge

Escriure malament el nom d'un missatge dóna lloc a un error. A la figura 2.5, hem escrit malament el selector de missatge `ves:`, escrivint `veshi:`. El missatge `veshi:` no existeix a Squeak, per tant Squeak acoloreix la paraula de vermell. Ignorant l'amable avís d'Squeak hem triat d'executar l'*script*. Squeak ha intentat endevinar quin selector de missatge teníem al cap, i ens ha suggerit un menú de possibilitats. En aquest moment, podríem triar el selector de missatge correcte (`ves:`) i el missatge `veshi:` seria substituït per `ves:`. O podríem triar "cancel·lar". Si decidim triar aquesta opció, haurem de canviar `veshi:` per `ves:` manualment.

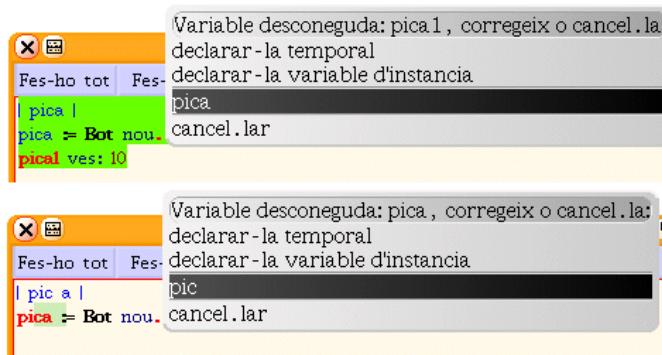


**Figura 2.5** — Hem escrit malament el missatge `ves:` escrivint `veshi:` per error. El missatge `veshi:` no existeix (a Squeak). Per tant Squeak suggereix possibles correccions.

## Escriure malament el nom d'una variable

Hi ha dues maneres d'escriure malament el nom d'una variable: en el cos de l'*script* i quan es declara (entre barres verticals, com a `| pica |`). La figura 2.6 mostra els dos casos: A dalt hem declarat la variable `pica`, però hem escrit `pica1` a l'*script*, enlloc de `pica`. Squeak s'ha adonat que intentàvem utilitzar una variable no declarada, de manera que l'ha acolorit de vermell i suggereix o bé que declarem la variable `pica1` com a nova variable, o bé que substituïm `pica1` per `pica`. Com que `pica` és el nom de variable que en realitat volem i `pica1` només era un error, triem l'opció `pica`, com es veu a la figura. A baix es mostra com hem escrit accidentalment un espai entre la `c` i la `a` en escriure `pica` mentre la declaràvem. Squeak no ha considerat que això sigui un error, simplement ha "pensat" que estàvem intentant declarar dues variables, `pic` i `a`. Aleshores, dins l'*script* hem escrit `pica`, pensant que ja havíem declarat la variable. Squeak, però, ha detectat que

pica és una variable no declarada de manera que l'ha acolorit de vermell i ens ha suggerit algunes opcions, incloent-hi declarar una nova variable amb nom pica o substituir el que hem escrit per la variable declarada pic.



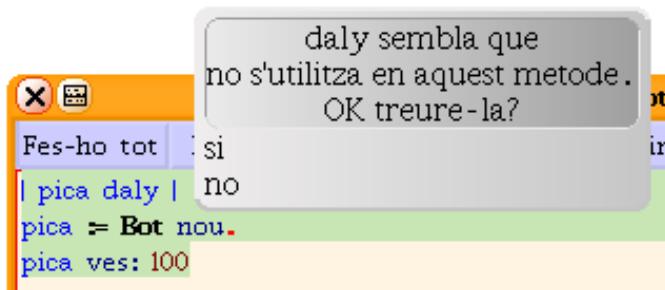
**Figura 2.6 — Dos exemples d'error.** A dalt: Hem escrit pica1 enllloc del nom de la variable declarada pica. A baix: Accidentalment hem escrit pic a quan provàvem de declarar la variable pica. Això té com a resultat la declaració de les variables pic i a i no de la variable pica.

## Variables no utilitzades

Pot passar que accidentalment declareu massa variables. Per exemple, podríeu declarar les variables pica i daly, pensant que necessitareu dos robots, però no utilitzar mai daly a l'*script*. Això no és realment un error, i el vostre programa podria executar-se correctament amb variables declarades i no utilitzades. Això és anàleg a comprar dues maletes, per si de cas, però utilitzar-ne només una. Simplement tenim equipatge extra que no fem servir. Tot i així, per si realment volíeu utilitzar daly i us n'heu oblidat, Squeak comprova si hi ha variables declarades i no utilitzades i, si en troba alguna, suggereix que potser voldríeu esborrar-la. Per exemple, a la figura 2.7, l'*script* declara les variables pica i daly però només utilitza pica. Squeak s'adona d'això i pregunta si voldríeu esborrar la variable no utilitzada daly.

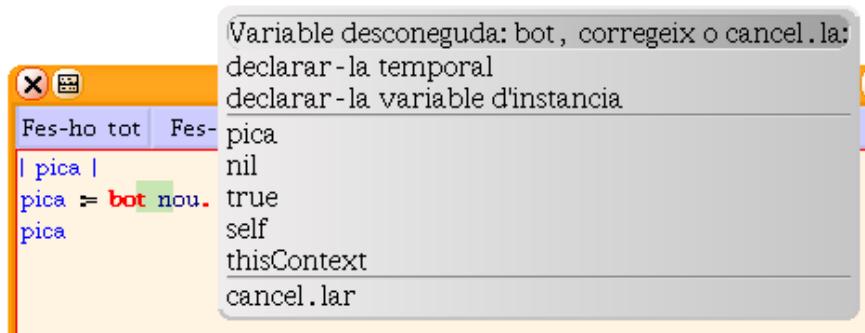
## Majúscules o minúscules?

Un altre error comú és oblidar que hi ha lletres que cal posar en majúscules. Els noms de les classes comencen amb un caràcter en majúscules, així que no ho oblideu quan vulgueu enviar un missatge a una fàbrica d'objectes. La figura 2.8 ens mostra com hem escrit distretament bot



**Figura 2.7** — Totes les variables i els enviaments de missatges són correctes. La variable daly, però, s'ha declarat tot i no fer-la servir, de manera que Squeak ens ho fa notar i ens suggereix si voldríem esborrar-la. Les variables declarades i no utilitzades no són un error, però les coses com més senzilles millor, així que hauríem d'esborrar-la.

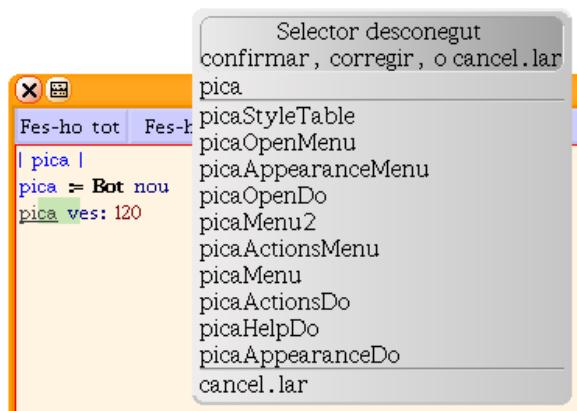
enlloc de Bot. Squeak ha provat d'endevinar què és el que volíem dir, però no ha pogut i cap de les opcions que ofereix per arreglar el problema serveix. En aquest cas, cal que un mateix corregeixi l'error. En el context d'aquest llibre, les úniques classes de què us heu de preocupar són Bot, la fàbrica de robots, i Color, la fàbrica de colors.



**Figura 2.8** — Hem oblidat la majúscula B al nom de la classe Bot, la fàbrica de robots. Squeak sap que alguna cosa no va bé, però no està segur de quina és. Haurem de corregir l'error nosaltres mateixos.

## Oblidar un punt

Finalment, un dels errors més comuns, un que fins i tot els programadors cometen, és oblidar un punt entre dos enviaments de missatge, o un punt i coma entre missatges en una cascada. Un punt indica que està a punt de començar un nou enviat de missatge, però sense el punt Squeak creu que el missatge actual continua, i que la variable que hauria de ser el receptor d'un nou missatge és només un altre selector de missatge. Com que no hi ha cap selector de missatge amb el nom de la variable, Squeak us diu que heu escrit un selector desconegut i us ofereix algunes correccions. Per exemple, a la figura 2.9 no hi ha punt darrera l'expressió pica := Bot nou, i Squeak analitza (vull dir, intenta endevinar quina és l'estructura) el missatge pica := Bot nou pica ves: 120, i d'acord amb les regles de la sintaxi (l'estructura) dels missatges, que aprendreu en el capítol 11, pica hauria de ser un selector de missatge. Però aquest selector de missatge no existeix, de manera que Squeak es queixa i proposa algunes possibles substitucions. Com que vosaltres ja sabeu que pica és la vostra variable declarada i no un selector de missatge, us adoneu que heu oblidat un punt, trieu "cancel·lar" i poseu el punt manualment.

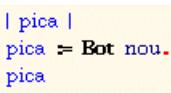
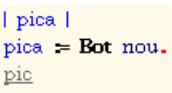


**Figura 2.9** — Les conseqüències d'oblidar un punt entre enviaments de missatges: Squeak creu que el receptor del missatge del segon enviat de missatge és un selector de missatge inexistent.

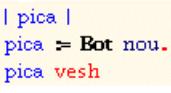
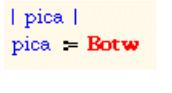
## Paraules que canvien de color

Squeak prova d'identificar els errors mentre esteu escrivint els vostres *scripts*. Si detecta alguna cosa que no lliga amb el que espera, canvia el color del text i proporciona algunes indicacions visuals que sugereixen el que pot estar malament. La figura 2.10 ens mostra algunes situacions

típiques. Malauradament, si no veieu la figura en color haureu d'utilitzar la vostra imaginació!

			
<b>(a) començant a escriure una variable no declarada</b>	<b>(b) variable declarada</b>	<b>(c) començant a escriure el principi d'una variable declarada</b>	<b>(d) variable no declarada</b>

		
<b>(e) variable no declarada</b>	<b>(f) missatge desconegut</b>	<b>(g) classe desconeguda</b>

**Figura 2.10 — Squeak utilitza colors per ajudar-vos a trobar errors i per saber si tot va bé.**

Aquí teniu una explicació més detallada de la figura:

- (a) Hem començat a escriure la primera lletra d'una variable desconeguda o no declarada. Com que no ha estat declarada cap variable començant amb la lletra x, Squeak l'acoloreix de vermell, fent-nos saber que alguna cosa està malament.
- (b) Hem acabat d'escriure una variable que ha estat declarada. Squeak ens diu que hem escrit una variable declarada acolorint el nom de blau.
- (c) Estem escrivint el nom d'una variable. Mentre el que anem escrivint es correspongui amb el començament del nom d'alguna variable declarada, Squeak el subratlla per fer-nos saber que fins ara tot és correcte.
- (d) Tan aviat com escriguem un caràcter al nom de la variable de manera que la seqüència de caràcters ja no sigui el començament d'una variable declarada, Squeak acoloreix el text de vermell. Fixeu-vos en la diferència respecte del cas anterior. En el cas (c), podríem haver escrit el caràcter a i completar el nom de la variable declarada pica, com en el cas (a). Hem escrit, però, el caràcter b i hem obtingut una seqüència de lletres (picb) que no és el començament del nom de cap variable declarada.

- (e) Després d'escriure el nom d'una variable declarada (pica, com en el cas (b)), accidentalment hi hem afegit un caràcter més a, que dóna lloc a picaa, que no és el començament d'una variable declarada.
- (f) Squeak intenta fer el mateix que fa amb els noms de les variables per als selectors de missatge. Aquí, hem escrit malament el missatge ves: i hem escrit vesh en el seu lloc. Squeak estava esperant un selector de missatge i tan aviat com hem escrit el caràcter h, s'ha adonat que no hi ha cap missatge que comenci per vesh, de manera que ha acolorit de vermell el que estàvem escrivint.
- (g) Squeak prova de fer el mateix amb les classes. Aquí hem escrit el caràcter w després de Bot, i Squeak, esperant un nom de classe ja que hem començat el nom amb B majúscula, acoloreix Botw de vermell ja que no hi ha cap classe al sistema el nom de la qual comenci així.

## Resum

- Per executar una expressió, premeu el botó **Fes-ho tot** de l'espai de treball.
- Un *script* és una seqüència d'expressions que fa alguna tasca.
- Un missatge està compost d'un selector de missatge i possiblement un o més arguments. Alguns selectors de missatge no necessiten cap argument, com a l'enviament de missatge pica tornarInvisible.
- Qualsevol selector de missatge que acaba amb dos punts requereix informació addicional (un o més arguments), com una longitud o un angle. Per exemple, el selector de missatge giraEsquerra: necessita un argument el valor del qual és un nombre que representa l'angle que el robot hauria de girar en sentit contrari a les agulles del rellotge.
- Per obtenir un objecte nou, usualment cal enviar el missatge nou a una classe. Per exemple, Bot nou crea un robot nou. Pot ser que altres classes entenguin altres missatges per crear objectes nous. Per exemple, Color groc demana a la classe Color que creï un color groc nou.
- Una classe és una fàbrica per produir objectes. Els noms de les classes sempre s'han de fer començar amb una lletra majúscula. Per exemple, Bot és la fàbrica per crear nous robots, i Color és la fàbrica de colors. El missatge Bot nou color: Color groc demana a la classe Bot que creï un robot nou, i després demanem a la fàbrica de colors que fabriqui un nou objecte color groc. Finalment, el missatge color: s'envia al nou robot amb el nou objecte color groc com a argument, resultant en un robot nou de color groc.

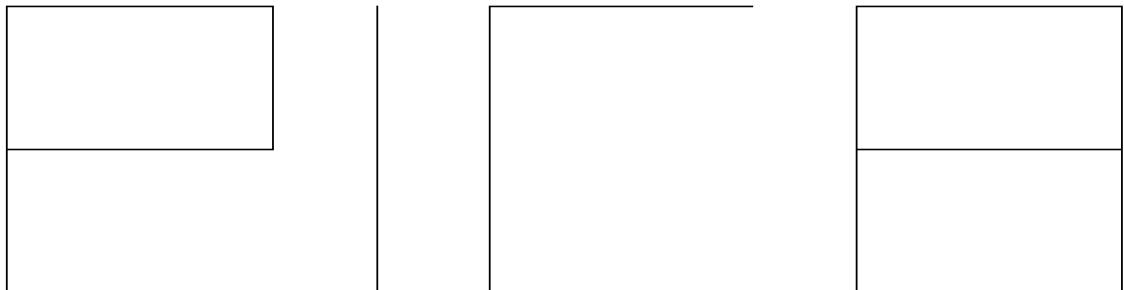
- Els enviaments de missatges han de ser separats amb un punt. No cal posar un punt al final, després del darrer enviament de missatge. Aquí teniu un exemple de quatre enviaments de missatge separats per tres punts:

```
pica := Bot nou.  
pica ves: 100.  
pica giraEsquerra: 90.  
pica ves: 100
```

- Per enviar múltiples missatges al mateix objecte podeu utilitzar un punt i coma per separar els missatges, com a unBot *missatge1* ; *missatge2*. Per exemple, pica ves: 100 ; giraEsquerra: 90 ; ves: 200 ; giraEsquerra: 200 envia una seqüència de quatre missatges (1) ves: 100, (2) giraEsquerra: 90, (3) ves: 200, (4) giraEsquerra: 200 al robot anomenat pica.

## Capítol 3

# Homes i robots



En aquest capítol<sup>1</sup> descrivim la creació de robots i els diferents tipus de moviments que els robots coneixen i són capaços de realitzar. Perque pugueu practicar el que heu après en capítols anteriors, us proposem alguns experiments senzills. També us ensenyarem com els robots poden canviar de direcció seguint els punts cardinals.

### Crear robots

Al capítol anterior heu creat *un* robot, no *el* robot. Volem dir que els robots no són únics, i que podeu crear tants robots com vulgueu. L'*Script* 3.1 crea dos robots: pica i daly.

<sup>1</sup>Nota del Traductor: El títol del capítol en anglès és *Of Robots and Men* recordant el títol de la novel·la de John Steinbeck *Of Mice and Men*. D'acord amb la traducció al català de la novel·la publicada per l'editorial La Galera, el títol traduït de la novel·la és "Homes i Ratolins". Per tant, he traduït el títol del capítol per mantenir la coherència amb la traducció catalana de l'obra d'Steinbeck.

**Script 3.1** *El naixement de dos robots.*

```
| pica daly |
pica := Bot nou.
daly := Bot nou.
pica color: Color groc.
daly salta: 100.
```

La segona línia crea un robot anomenat pica, com a l'*script 2.1*. La tercera línia crea un robot nou a què ens referirem utilitzant la variable daly (igual que vam fer amb la variable pica, el nom de la qual és un homenatge a Pablo Picasso, el nom de la variable daly és un homenatge a Salvador Dalí). Tots dos robots són creats al mateix lloc de la pantalla. A la línia quatre, li diem a pica que canviï el seu color a groc de manera que puguem distingir els dos robots.

Tal i com ja hem dit abans, Smalltalk és un llenguatge orientat a objectes. Això no només significa que podem crear objectes i interactuar amb ells, sinó que a més els objectes poden crear altres objectes i comunicar-se amb ells. És més, a Smalltalk hi ha objectes especials, anomenats *classes*, que s'utilitzen per crear objectes. En general, enviant el missatge new a una classe creem un objecte tal i com és descrit a la classe. Enviar el missatge nou a la classe Bot crea un robot.

Per entendre què són les classes, imagineu una classe com una mena de fàbrica. Una fàbrica per crear capses pot fabricar un gran nombre de capses sense cap ús en especial, totes de la mateixa forma, mida i color. Després de fabricades, algunes capses poden acabar contenint galetes i d'altres es poden trencar. Quan una capsà es treu, les altres capses no es veuen afectades. El mateix passa amb els objectes creats dins d'Squeak. En el nostre cas, daly no ha canviat de color, tot i que pica sí que ho ha fet, mentre que pica no s'ha mogut i daly sí. Podeu pensar en una classe com en una fàbrica capaç de produir un nombre il·limitat d'objectes d'un mateix tipus. Un cop fabricats, cada objecte existeix independentment dels altres i el podem modificar tan com vulguem.

A Smalltalk, els noms de les classes sempre comencen amb una lletra majúscula. Aquesta és la raó per la qual el nom de la classe robot és Bot, amb "B" majúscula. Fixeu-vos que a la instrucció Color groc, la paraula Color s'ha escrit amb una "C" majúscula. Això és per que Color és una classe, i el que fabrica són objectes color. Especificant el nom del color, obtenim un objecte color del color que volem (l'expressió Color groc és una manera abreujada de crear un objecte color groc. Primer es crea un objecte color enviant el missatge new a la classeColor, i després alguns missatges més defineixen l'objecte color com a groc).

---

**Important!** Una classe és una fàbrica que manufactura objectes. En general, enviant el missatge new a una classe creem un objecte d'aquella classe. Els noms de les classes sempre comencen amb una lletra majúscula. Aquí, Bot és el nom de la fàbrica per crear nous robots, i Color és la fàbrica de colors.

Així, l'enviament de missatge Bot nou color: Color blau envia un missatge a la classe Bot per crear un robot nou i després envia un missatge al nou robot perquè s'acoloreixi ell mateix de blau.

---

## Dibuixar segments de línia

Demanar a un robot que dibuixi una línia és força senzill, tal com heu vist al capítol anterior. El missatge ves: 100 li diu al robot que es mogui endavant 100 píxels, i el robot deixa una marca mentre es mou. Quan dibuixem, però, cal que de tant en tant aixequem el llapis del paper, fins i tot si som uns escriptors experts en cal·ligrafia xinesa o japonesa. Per això, un robot sap saltar; és a dir, el robot sap moure's sense deixar cap traça. Els robots entenen el missatge salta: l'argument del qual és el mateix que per al missatge ves:, una distància donada en píxels. L'*Script* 3.2 dibuixa dos segments. Per què quedí clar el que dibuixen els robots, aquests s'han fet desapareixer de la il·lustració tot enviant-los el missatge tornarInvisible.

**Script 3.2** Creem pica i després dibuixa dues línies.



```
| pica |  
pica := Bot nou.  
pica ves: 30.  
pica salta: 30.  
pica ves: 30.
```

### Experiment 3-1 (crear i moure un robot)

Experimenteu canviant els valors de l'*script* anterior.

---

### Experiment 3-2 (SOS)

Escriviu un *script* que dibuixi el missatge “SOS” en codi Morse (en codi Morse, una “S” es representa amb tres línies curtes i una “O” amb tres línies llargues, com mostrem a la figura)

— — — — — — — — — — — —

## Canviar de direcció

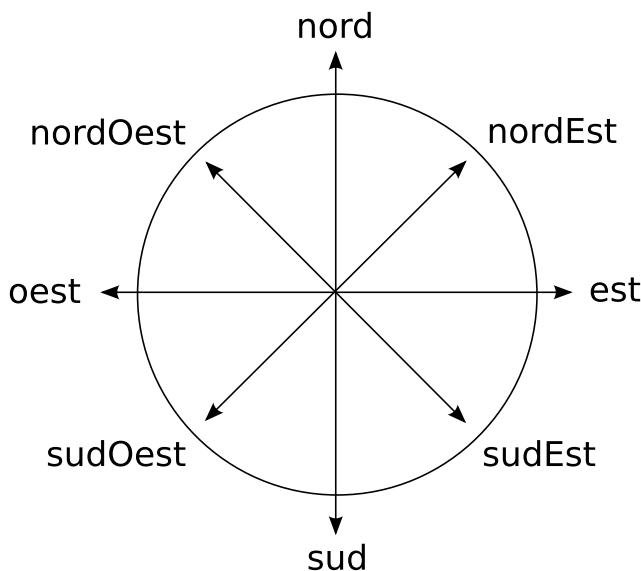
Un robot es pot orientar en les vuit direccions principals d’una brúixola, com es pot veure a la figura 3.1. Les direccions són com les d’un mapa normal i corrent: est és a la dreta, oest és a l’esquerra, nord és amunt i sud és avall. Aquestes direccions són *absolutes*, la qual cosa significa que no importa en quina direcció estigui apuntant el robot, si li diem que apunti a l’est, el robot apuntarà a la dreta de la pantalla, no a la seva dreta. Per apuntar un robot en una direcció absoluta, només cal enviar-li un missatge amb el nom de la direcció. Així, si volem que pica apunti al sud, simplement escrivim pica sud.

Els robots entenen els següents missatges de direcció respecte als punts cardinals: est, nord, nordEst, nordOest, sud, sudEst, sudOest i oest. Al proper capítol, us ensenyarem com fer que un robot giri un angle qualsevol de manera relativa a la seva posició actual.

L’*Script* 3.3 il·lustra les quatre direccions cardinals amb quatre robots diferents; aquí Picasso i Dali són acompanyats per Paul Klee i Alfred Sisley. Excepte per pica, que es manté en la direcció est, on apunten per defecte els robots, hem orientat a cada robot en una direcció diferent abans de dir-li que es mogui.

**Script 3.3** *Un grup de robots en moviment.*

```
| pica daly klee sisli |
pica := Bot nou.
pica color: Color verd.
pica ves: 100.
daly := Bot nou.
daly nord.
daly color: Color groc.
daly ves: 100.
klee := Bot nou.
```



**Figura 3.1** — Les direccions absolutes d'una brúixola a què un robot pot apuntar.

klee oest.

klee color: Color vermell.

klee ves: 100.

sisl := Bot nou.

sisl sud.

sisl ves: 100.

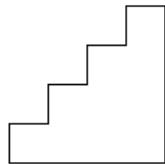
Podeu utilitzar aquests mètodes d'orientació per fer dibuixos molt més complicats.

### Experiment 3-3 (quadrat)

Com a primer exercici, dibuixeu un quadrat de costat 50 píxels. Després dibuixeueu-ne un altre de costat 250 píxels.

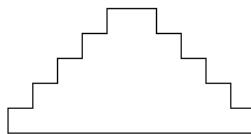
### Experiment 3-4 (escala)

No només podeu dibuixar quadrats. Podeu crear un ampli ventall de figures geomètriques. Per exemple, aquí teniu el dibuix d'una petita escala. Escriviu un *script* per reproduir aquest dibuix.



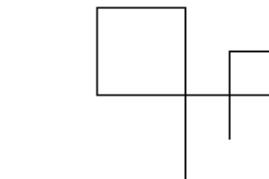
### Experiment 3-5 (la piràmide esglaonada de Saqqara)

Ara ja podeu desplegar el vostre enginy arquitectònic i dibuixar una vista esquemàtica de la piràmide esglaonada de Saqqara, construïda cap al 2900 a.c. per l'arquitecte Imhotep. Feu un *script* que dibuixi una vista lateral de la piràmide (veieu la figura). La piràmide té quatre esglaons, i la part superior és el doble de llarga que cada esglaó.



### Experiment 3-6 (art abstracte)

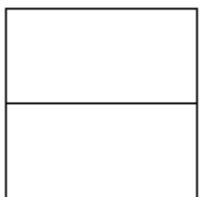
Escriviu un *script* per reproduir el dibuix d'aquesta figura.



## L'ABC del dibuix

Fins i tot sense tenir encara gaire control sobre la direcció en què el vostre robot dibuixa segments de línia, podeu començar a programar a pica per dibuixar lletres. L'*Script 3.4* dibuixa una "A" més aviat primitiva.

**Script 3.4** *Dibuixem la lletra A.*



```
| pica |
pica := Bot nou.
pica nord.
pica ves: 100.
pica est.
pica ves: 100.
pica sud.
pica ves: 100.
pica nord.
pica ves: 50.
pica oest.
pica ves: 100.
```

Dibuixar una lletra "C" no és més difícil. Podeu fins i tot escriure la paraula "pica".

### Experiment 3-7 (PICA)

Dibuixeu el nom "pica" tal com es mostra al principi d'aquest capítol. Per separar les lletres individuals hauríeu d'utilitzar el missatge salta:

---

---

**Observació** Es podria argumentar que l'*script* 3.4 es pot millorar. Per exemple, la meitat inferior de la línia vertical de la dreta de la “A” s’ha dibuixat dues vegades, ja que el robot retorna sobre aquest segment –un cop anant cap al nord, un altre cop anant cap al sud– per posar-se en posició de dibuixar la línia horitzontal. Decidir quina és la millor manera de resoldre un problema de programació pot ser difícil. Hi ha molts aspectes del problema a considerar, tals com la rapidesa, la complexitat o la llegibilitat del codi, i aquestes qüestions tindran diferents respuestes depenent de quin llenguatge de programació i quins mètodes s’hagin fet servir. Tot i així, podem considerar una primera aproximació triant la solució més senzilla. Aleshores, si estem insatisfets perquè el programa és massa lent o perquè no té les peculiaritats que desitjaríem, sempre podem modificar-lo per fer-lo més ràpid o afegir-li més ampliacions.

---

## Controlar la visibilitat del robot

Podeu controlar si un robot es mostra a la pantalla utilitzant els missatges tornarInvisible i tornarVisible. El missatge tornarInvisible amaga al receptor del missatge. Un robot ocult actua exactament com un de visible, però no ens ensenya on és. Aneu amb compte de no utilitzar el mètode hide (vol dir amagar), que està definit a Squeak pel seu ús particular i pot fer malbé l’entorn dels robots si s’utilitza inadequadament. El missatge tornarVisible fa que el robot receptor del missatge sigui visible. Un robot nou és visible per defecte.

## Resum

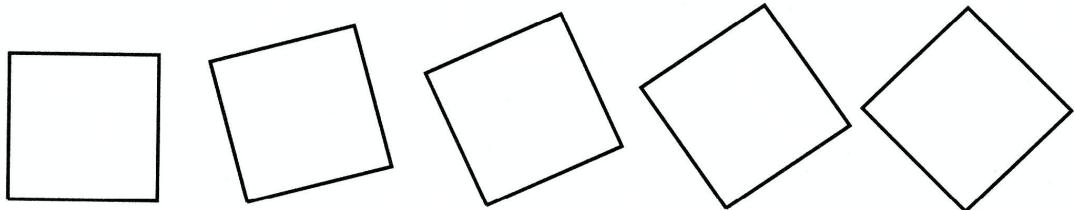
La taula següent resumeix les expressions i missatges apareguts en aquest capítol.

Expressions / Missatges	Descripció	Exemple
Bot nou	Crea un robot.	pica := Bot nou
x y	Declara variables per utilitzar en un <i>script</i>	pica
salta: <i>unEnter</i>	Diu al robot que s'ha de moure endavant un determinat nombre de píxels sense deixar cap traça	pica salta: 10
ves: <i>unEnter</i>	Diu al robot que s'ha de moure endavant un determinat nombre de píxels deixant una marca	pica ves: 10
tornarInvisible	Diu al robot que s'ha de tornar invisible	pica tornarInvisible
tornarVisible	Diu al robot que s'ha de tornar visible	pica tornarVisible
est, nordEst, nord, nordOest, oest, sudOest, sud, sudEst	Diu al robot que ha d'apuntar en una determinada direcció	pica nord
Color <i>nomDeColor</i>	Crea el color <i>nomDeColor</i>	Color blau
color: <i>unColor</i>	Demana al robot que canviï de color	pica color: Color vermell



## Capítol 4

# Direccions i angles



Ara ja hauríeu d'estar cansats de dibuixar figures només seguint direccions *fixades*. En aquest capítol aprendreu com canviar la direcció a què apunta un robot, permetent al robot apuntar a *qualsevol* direcció, girar qualsevol angle relatiu a la seva posició actual, i, per tant, dibuixar línies en qualsevol direcció. Si enteneu bé què és un angle i com mesurar angles en graus, podeu saltar la secció “L’Enfocament Adequat” i seguir amb els exemples i experiments de la secció “Dibuixos Senzills”.

Començarem presentant els missatges elementals que els robots entenen per canviar de direcció. Amagarem els robots de les il·lustracions utilitzant el missatge tornarInvisible de manera que els dibuixos es vegin més clars.

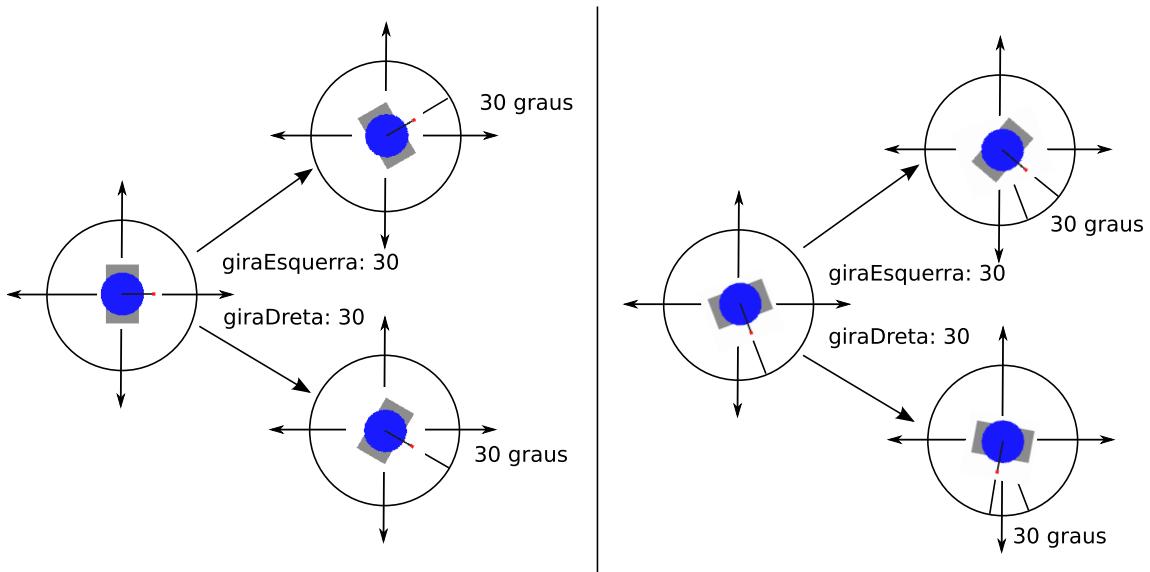
### Dreta o esquerra?

Al capítol anterior vau aprendre que podíem fer que un robot apuntés a diferents direccions amb els missatges est, nord, nordEst, nordOest, sud, sudEst, sudOest i oest. Amb aquests missatges, però, no podeu canviar la direcció del vostre robot un angle qualsevol, com per exemple 15 graus. A

més, tampoc podeu fer girar un robot, diguem, un quart de circumferència respecte de la seva posició actual.

Per girar un robot un determinat angle hem d'utilitzar els mètodes `giraEsquerra`: i `giraDreta`: que ordenen al robot girar a la dreta o a l'esquerra. Tal com indiquen els dos punts al final del nom dels mètodes, tots dos mètodes esperen un argument. Aquest argument és l'angle que el robot hauria de girar relatiu a la seva posició actual. És a dir, l'argument és la diferència entre la direcció del robot abans que el missatge fos enviat i la seva direcció després que el missatge sigui enviat. L'angle es dóna en graus. Per exemple, l'expressió `pica giraEsquerra: 15` demana a `pica` que giri a l'esquerra 15 graus partint de la seva posició actual, i `pica giraDreta: 30` fa girar `pica` a la dreta trenta graus partint de la seva posició actual. La figura 4.1 il·lustra l'efecte dels missatges `giraEsquerra`: i `giraDreta`: primer, quan un robot apunta a l'est i segon, quan un robot apunta a alguna altra direcció.

A mesura que aneu practicant fent girar els robots diversos angles, tingueu en compte que quan es crea un robot nou, sempre apunta cap a l'est, és a dir, a la dreta de la pantalla.



**Figura 4.1 — Esquerra:** Un robot apuntant a l'est gira 30 graus a l'esquerra o a la dreta. **Dreta:** Un robot apuntant a una altra direcció gira 30 graus a l'esquerra o a la dreta.

### Experiment 4-1 (*scripts* misteriosos)

Els *Scripts* 4.1 i 4.2 són problemes en els quals heu d'endevinar què farà el robot creat a cada *script*. Després que estudiueu aquests dos *scripts*, experimenteu amb ells canviant-los els valors dels angles, per exemple, per determinar quin angle fa que el robot giri un quart de circumferència, mitja circumferència o una circumferència sencera. Si us cal revisar la noció d'angle, llegiu la secció "L'Enfocament Adequat" abans de continuar.

---

#### Script 4.1 Què fa pica? (Problema 1)

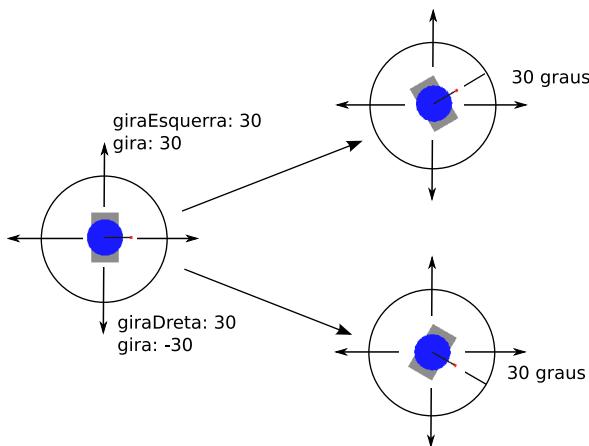
```
| pica |
pica := Bot nou.
pica ves: 100.
pica giraEsquerra: 45.
pica ves: 50.
pica giraEsquerra: 45.
pica ves: 100.
```

#### Script 4.2 Què fa pica? (Problema 2)

```
| pica |
pica := Bot nou.
pica ves: 100.
pica giraDreta: 60.
pica ves: 100.
pica giraEsquerra: 60.
pica ves: 100.
```

## Una convenció direccional

En matemàtiques és una convenció general que la rotació d'un angle negatiu es considera en el sentit de les agulles del rellotge, mentre que una rotació d'un angle positiu és en el sentit contrari al de les agulles d'un rellotge. Podeu fer servir aquesta convenció matemàtica utilitzant el missatge `gira:`. Per tant, el missatge `giraEsquerra: unNombre` és equivalent al missatge `gira: unNombre`, mentre que el missatge `giraDreta: unNombre`, és equivalent a `gira: -unNombre`, on `-unNombre` és el negatiu de `unNombre`. Aquesta relació està il·lustrada a la figura 4.2.



**Figura 4.2 — Girar 30 graus des de la direcció est.**

## Orientació absoluta vs. orientació relativa

Ja hauríeu de tenir prou confiança en les vostres habilitats manipulant robots com per fer que un robot realitzi qualsevol dibuix compost per línies rectes. Abans de continuar, estigueu segurs que enteneu la diferència entre orientar un robot de manera *absoluta* utilitzant les mètodes nord, sud, sudEst, est, etc., i fer servir els mètodes gira:, giraEsquerra: i giraDreta: per orientar al robot de manera *relativa* a la seva orientació actual.

Els experiments 4-2, 4-3 i 4-4 us ajudaran a acabar d'entendre aquesta diferència.

### Experiment 4-2 (un quadrat relatiu)

Escriviu un *script* per dibuixar un quadrat utilitzant el mètode giraEsquerra: o giraDreta:




---

### Experiment 4-3 (girar el quadrat)

Modifiqueu l'experiment anterior afegint la línia `pica giraEsquerra: 33.` abans de la primera línia contenint el missatge `ves: 100`. Obtindreu un quadrat altre cop, però aquesta vegada estarà girat 33 graus respecte al que ja havíeu dibuixat.

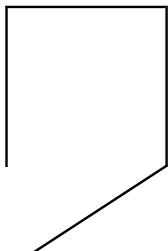
---

### Experiment 4-4 (un quadrat trencat)

Finalment, executeu l'*script* 4.3, que intenta dibuixar un quadrat girat utilitzant els mètodes `nord`, `sud`, `est`, i `oest` que hem introduït al capítol anterior.

---

**Script 4.3** *Un quadrat trencat.*

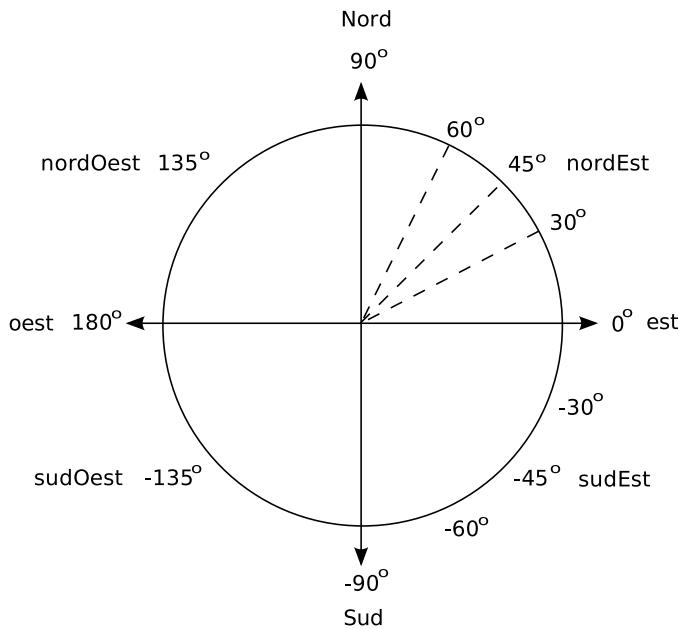


```
| pica |
pica := Bot nou.
pica giraEsquerra: 33.
pica ves: 100.
pica nord.
pica ves: 100.
pica oest.
pica ves: 100.
pica sud.
pica ves: 100.
```

Encara obtenuiu un quadrat? No! El primer costat dibuixat pel robot està tort, mentre que els altres costats són horitzontals o verticals. L'*script* que vau escriure per a l'Experiment 4-3 i l'*Script* 4.3 demostren la diferència tan important que hi ha entre canvis *relatius* i *absoluts* de direcció:

- Els mètodes nord, sud, est, i oest canvien la direcció d'una manera *absoluta*. La direcció a què el robot acabarà apuntant *no depèn* de la direcció actual a què està apuntant.
- Els mètodes giraEsquerra: i giraDreta: canvien la direcció de manera *relativa*. La direcció a què el robot apunta *depèn* de la seva direcció actual.

La figura 4.3 mostra l'equivalència entre moviments relatius a partir d'un robot que apunta a l'est i moviments absoluts. Com ja sabeu, aquesta equivalència és només vàlida si el robot està apuntant a l'est i no si està apuntant a qualsevol altra direcció. Fixeu-vos que girant el robot 180 graus apunta a la direcció oposada; aquest truc s'utilitza sovint als *scripts*.



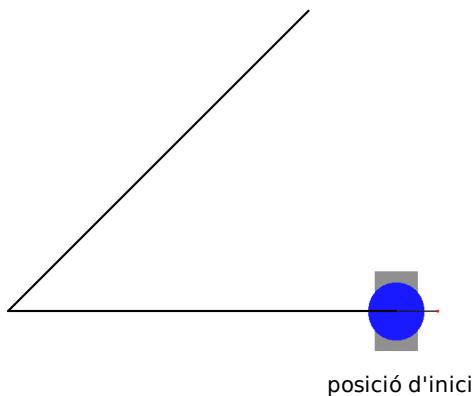
**Figura 4.3 — Comparar l'orientació relativa i absoluta començant per l'est.**

## L'enfocament adequat

Un robot nou, com ja sabeu, apunta l'est, és a dir, cap al costat dret de la pantalla. Si demanem al robot que giri a l'esquerra 90 graus, acabarà apuntant al nord. Si li demanem que giri a la dreta

90 graus, apuntarà al sud. L'*Script* 4.4 il·lustra el resultat d'un gir a l'esquerra de 45 graus. Per ajudar-vos a entendre l'*script*, la figura us mostra el punt de partida del robot.

**Script 4.4** *Jugar amb els angles (1)*



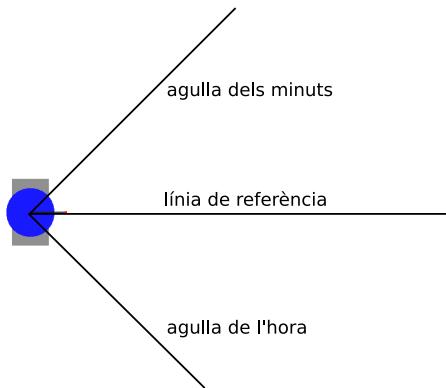
```
| pica |
pica := Bot nou.
pica oest.
pica ves: 100.
pica est.
pica giraEsquerra: 45.
pica ves: 100.
```

La primera part de l'*script* 4.4, fins a la línia pica est, dibuixa una línia horitzontal, que farà el paper de línia de referència per indicar la direcció est. La darrera part dibuixa una línia en la direcció 45 graus a l'esquerra de la direcció est. Podeu fer variar el valor de l'angle per veure quina mena d'angles representen altres quantitats en graus. Proveu els valors 60, 120, 180, 240, 360 i 420. En particular, fixeu-vos que un gir de 180 graus és el mateix que tombar el robot per a que apunti a la direcció oposada a la que apuntava.

Veieu alguna diferència entre els arguments 60 i 420? Representen el mateix angle! Qualsevol parell de valors tals que la seva diferència sigui 360 o qualsevol múltiple de 360 són equivalents, ja que 360 graus representen una circumferència completa. Intenteu un angle de 1860 (1860 =  $60 + 360 \times 5$ ). El resultat és el mateix que obtindríeu amb valors 60 i 420. Així, quan treballieu amb angles recordeu que l'orientació d'un robot no canvia afegint un o més girs complets a aquesta orientació.

Ara provarem de divertir-nos amb el mètode `giraDreta()`. L'*Script* 4.5 dibuixa les agulles d'un rellotge (l'hora i els minuts) i una línia que ens servirà de referència. Utilitza dos robots, que podeu utilitzar per investigar la correspondència entre els girs a l'esquerra i els girs a la dreta. Hem afegit comentaris entre cometes i hem utilitzat diferents fonts per ajudar-vos a identificar les diverses parts de l'*script*. Fixeu-vos que no heu d'escriure els comentaris, ja que no s'executaran.

#### **Script 4.5 Jugar amb els angles (2)**



```
| pica daly |
pica := Bot nou.
pica salta: 200.           "dibuixar la línia de referència"
pica giraEsquerra: 180.
pica ves: 200.
pica giraEsquerra: 180.
pica color: Color blau.
pica giraEsquerra: 45.     "dibuixar l'agulla dels minuts"
pica ves: 150.
daly := Bot nou.
daly color: Color vermell.
daly giraDreta: 45.      "dibuixar l'agulla de l'hora"
daly ves: 100.
```

A l'*Script* 4.5, el codi en itàlica dibuixa la línia de referència –és a dir, la línia que representa la direcció del robot abans de fer cap gir– utilitzant el fet que un gir de 180 graus és el mateix que girar per apuntar a la direcció oposada. La línia de referència és també la línia més llarga. Així,

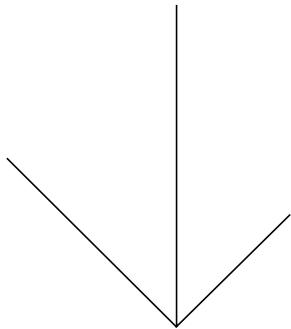
encara serà visible si les línies dibuixades pels robots se superposen a la línia de referència. El text en font normal és el codi que dibuixa l'agulla minutera (utilitzant pica) i en negreta, el codi dibuixant l'agulla de l'hora utilitzant el robot daly.

### Experiment 4-5 (moure les agulles del rellotge)

Experimenteu amb diferents valors dels angles per a cada un dels dos robots; és a dir, canviieu els valors dels angles per als dos mètodes per girar. Després, compareu l'efecte del mètode giraEsquerra: 60 (per a pica) i giraDreta: 300 (per daly). Podeu veure que girar a l'esquerra 60 graus dóna el mateix resultat que girar a la dreta 300 graus. Això és així ja que la suma dels valors és 360 graus, és a dir, una circumferència sencera.

Ara veurem el que passa quan el robot gira a partir d'una altra direcció. Aquí teniu un *script* igual a l'*script* 4.4, però aquest cop comencem a girar des del nord. En aquest *script* hem substituït daly per un altre robot, berthe, que fa honor a pintor impressionista francés Berthe Morisot.

#### Script 4.6 Jugar amb els angles (3)



```
| pica berthe |
pica := Bot nou.
pica nord.
pica salta: 200.
pica giraEsquerra: 180.
pica ves: 200.
pica giraEsquerra: 180.
pica color: Color blau.
```

```
pica giraEsquerra: 45.  
pica ves: 150.  
berthe := Bot nou.  
berthe nord.  
berthe color: Color vermell.  
berthe giraDreta: 45.  
berthe ves: 100.
```

### Experiment 4-6 (canviar la direcció de referència)

Continueu experimentant amb l'*Script* 4.6 canviant la direcció de referència. Per tal que la comparació sigui significativa, haureu d'orientar *berthe* en la mateixa direcció que *pica* després de crear-lo. Intenteu qual-sevol valor pels angles i proveu de fer prediccions respecte al resultat abans d'executar l'*script*. Continueu experimentant amb l'*script* fins que les vostres prediccions siguin prou acurades.

Fixeu-vos que sempre hauríeu de ser capaços de predir el que passarà abans d'executar un *script*, ja que l'ordinador executarà totes les instruccions vàlides cegament, incloent-hi les més absurdes.

## Un rellotge robot

He mencionat abans que les línies dibuixades a l'*Script* 4.6 són similars a les agulles d'un rellotge. L'analogia entre el temps i els angles és una bona analogia, ja que la noció de grau està força correlacionada amb la noció d'hora. Les civilitzacions antigues van descobrir la noció del temps mesurant l'angle del Sol (o d'un estel) relatiu a una direcció de referència. Un *script* com l'*Script* 4.6, però, us permet posar les agulles del rellotge en una posició que no indica cap moment real del dia. Per exemple, podríeu dibuixar un rellotge amb l'agulla de l'hora apuntant al nord i l'agulla dels minuts apuntant al sud. En canvi, en un rellotge real, si l'agulla dels minuts està apuntant al sud vol dir que ha passat mitja hora des de l'hora en punt i per tant l'agulla de l'hora hauria d'estar a mig camí entre dues hores.

Ara estudiareu la relació que hi ha entre l'agulla de l'hora i l'agulla minutera en un rellotge *real* que representa algun moment *real* del dia.

### Experiment 4-7 (un rellotge “real”)

Modifiqueu l'*Script* 4.6 de la manera següent:

- Manteniu la direcció de referència cap al nord (tal com està escrit a l'*Script* 4.6). Aquesta línia de referència indica les 12:00 del migdia o de la mitjanit.
- Utilizeu el mètode *giraDreta*: per ambdós robots. Després de tot, les agulles del rellotge es mouen en el sentit de les agulles del rellotge, que és cap a la dreta.
- Ara podem demanar a *pica* que dibuixi l’agulla minutera multiplicant el nombre de minuts que passen de l’hora que volem indicar per 6 (ja que durant els 60 minuts d’una hora, l’agulla minutera viatja els  $6 \times 60 = 360$  graus d’una circumferència sencera). Per exemple, per representar l’agulla minutera per 20 minuts passats de l’hora en punt, hauríeu d’utilitzar l’expressió *giraDreta*: 120 (ja que  $120 = 6 \times 20$ ).
- Podem demanar *berthe* que dibuixi l’agulla de l’hora multiplicant el nombre d’hores que volem indicar per 30 (12 hores multiplicat per 30 graus fan 360 graus) afegint mig (0.5) grau per cada minut que passi de l’hora en punt, ja que en 60 minuts, l’agulla de l’hora es mou 30 graus. Per exemple, podeu posicionar l’agulla de l’hora a les 2 en punt amb el missatge *giraDreta*: 60 ( $60 = 30 \times 2$ ), mentre que per a les 4:26 cal posicionar l’agulla de l’hora amb el missatge *giraDreta*: 133 ( $133 = 30 \times 4 + 26 \times 0.5$ ).

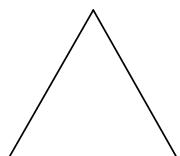
Intenteu dibuixar unes quantes hores i minuts del dia amb aquest *script* modificat.

---

## Dibuixos senzills

Per començar, aquí teniu un *script* per dibuixar un triangle amb tres costats iguals.

**Script 4.7** *Un triangle equilàter.*



```
| pica |
pica := Bot nou.
pica ves: 100.
pica giraEsquerra: 120.
pica ves: 100.
```

pica giraEsquerra: 120.

pica ves: 100.

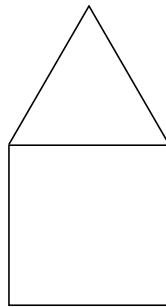
pica giraEsquerra: 120.

La darrera línia de codi no és necessària per dibuixar el triangle; serveix per apuntar pica altra cop cap a l'est (la seva posició inicial).

Ara, ja esteu preparats per dibuixar una casa.

### Experiment 4-8

Dibuixeu una casa tal com es mostra a la figura. Intenteu dibuixar cases de diferents formes.



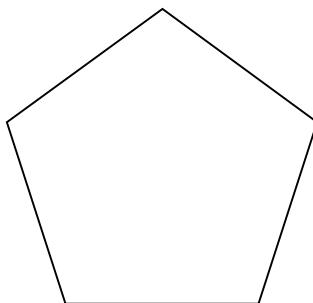
## Polígons regulars

Un polígon regular és una figura composta per segments de línia tots de la mateixa longitud i tots els angles de la qual són iguals. Un triangle equilàter és un polígon regular amb tres costats. Un quadrat és un polígon regular amb quatre costats. Per exemple, l'*Script 4.7* dibuixa un triangle equilàter de costats de mida 100 pixels. S'obté dient-li a pica d'anar endavant 100 pixels i girar a la dreta 120 graus, i repetir aquests dos missatges dues vegades més, de manera que en total són executats tres vegades.

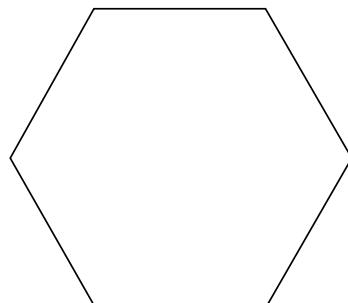
Podeu programar un robot per dibuixar un polígon regular amb qualsevol nombre de costats demanant-li que es mogui una certa quantitat de pixels i després giri a l'esquerra o a la dreta 360 graus dividit pel nombre de costats del polígon; aquesta seqüència s'ha de repetir tantes vegades com costats tingui el polígon. Fixeu-vos que el darrer gir del robot no cal que el posem, ja que el robot ja ha dibuixat l'última línia del polígon.

**Experiment 4-9**

Dibuixeu un pentàgon regular (un polígon regular amb cinc costats), tal com es mostra a la figura, amb costats de 100 píxels.

**Experiment 4-10**

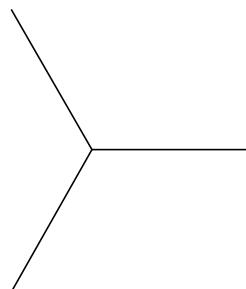
Dibuixeu un hexàgon regular (un polígon regular amb sis costats), tal com es mostra a la figura, amb costats de 100 píxels.



Si teniu prou curiositat per veure fins a on podeu arribar en aquest procés, podeu utilitzar les possibilitats de tallar i enganxar de l'Espai de Treball Bot per generar polígons regulars d'un gran nombre de costats. Si voleu, aneu incrementant el nombre de costats. Al capítol 7, us ensenyarem com podeu escriure una seqüència d'expressions i fer que es repeteixi tantes vegades com vulgueu.

### Experiment 4-11

Dibuixeu aquesta figura amb tres radis.



## Resum

- Un robot es pot orientar *relativament* a la seva direcció actual amb els mètodes giraEsquerra: i giraDreta::
- El paràmetre que cal proporcionar als mètodes giraEsquerra: i giraDreta: es dóna en graus.
- Girar 360 graus correspon a girar una circumferència sencera
- Girar 180 graus correspon a girar mitja circumferència.
- Els angles els valors dels quals difereixen en algun múltiple de 360 graus són equivalents.

Aquí teniu la llista de mètodes que heu après en aquest capítol.

Mètode	Sintaxi	Descripció	Exemple
giraEsquerra:	giraEsquerra: unNombre	Demana al robot que canviï la seva direcció un determinat nombre de graus a l'esquerra.	pica giraEsquerra: 30
giraDreta:	giraDreta: unNombre	Demana al robot que canviï la seva direcció un determinat nombre de graus a la dreta.	pica giraDreta: 30
gira:	gira: unNombre	Demana al robot que canviï la seva direcció un determinat nombre de graus, seguint la convenció matemàtica que ens diu que un nombre positiu representa un gir a l'esquerra i un nombre negatiu representa un gir a la dreta.	pica gira: 30
tornarInvisible	tornarInvisible	Amaga el receptor	pica tornarInvisible
tornarVisible	tornarVisible	Mostra el receptor	pica tornarVisible



# Capítol 5

## L'entorn de Pica

En aquest capítol us presentarem l'entorn de pica, us ensenyarem com utilitzar les eines disponibles i a guardar els vostres *scripts*. També retornarem a la noció de missatge i us ensenyarem que no només podeu demanar a l'entorn que executi un missatge, sinó que a més li podeu demanar que escrigui el *resultat* de l'execució del missatge.

### El menú principal

Quan feu clic al fons de l'entorn obtenuï el menú principal, com podeu veure a la figura 5.1.

Si voleu saber què fa una determinada opció del menú, mogueu el punter del ratolí sobre l'opció durant un segon i voilà! apareixerà una bafarada que descriu l'opció. El menú principal dóna accés a cinc grans grups de funcionalitats: l'accés a eines, les captures de pantalla, l'accés a alguns comportaments dels robots, l'aspecte i guardar l'entorn. Els submenús estan agrupats de la següent manera:

- El menú **obre...** recull diverses eines com ara l'explorador del codi dels robots, l'Espai de Treball Bot, un explorador de fitxers i altres eines que anirem veient a mida que les necessitem.
- El menú **accions BotInc...** recull diverses accions com per exemple indicar quina versió de l'entorn estem utilitzant o esborrar tots els robots i les seves traces, també n'inclou d'altres per reinstal·lar l'entorn si cal: reinstal·lar les preferències per defecte assigna els valors per defecte a les preferències que poguéssiu haver modificat amb el menú d'aspecte.
- El menú **aspecte...** recull accions que canvien l'aspecte de l'entorn, com canviar les fonts utilitzades, activar el mode de pantalla completa o modificar color del fons de l'entorn.

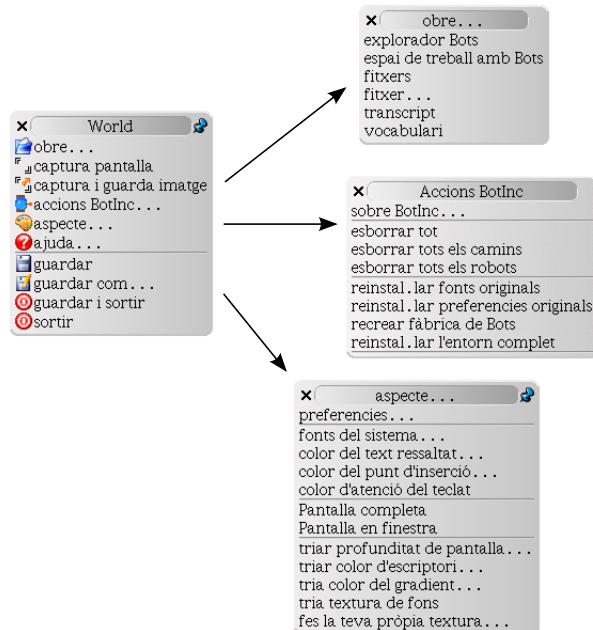


Figura 5.1 — Opcions del menú de l'entorn

## Obtenir un Espai de Treball Bot

Si tanqueu l'Espai de Treball Bot per accident no us amoïneu. Podeu obtenir-ne un de nou a partir de la solapa blava, tal com es mostra a l'esquerra de la figura 5.2, o a partir del menú del Món, com veiem a la figura 5.1. Per instal.lar un nou Espai de Treball Bot a la solapa de Treball, obriu-la (la solapa que està situada a la part de sota) i arrossegueu l'Espai de Treball Bot des de la solapa blava fins a la solapa de Treball.

La solapa blava conté altres eines que utilitzarem més endavant. La segona eina és bàsicament un explorador de codi que utilitzareu quan comenceu a definir nous mètodes per als robots.

L'entorn conté una eina senzilla (figura 5.3) que llista els missatges més importants que un robot pot entendre. Podeu accedir-hi via el menú **obrir... vocabulari** o el menú **d'ajuda** (opcio **obre vocabulari**). La finestra del vocabulari llista els missatges, agrupats d'acord al seu tipus. Per exemple, els missatges est, nord i similars estan llistats sota **direccions absolutes**.



**Figura 5.2 —** Obtenir un nou Espai de Treball Bot a partir de les solapes.

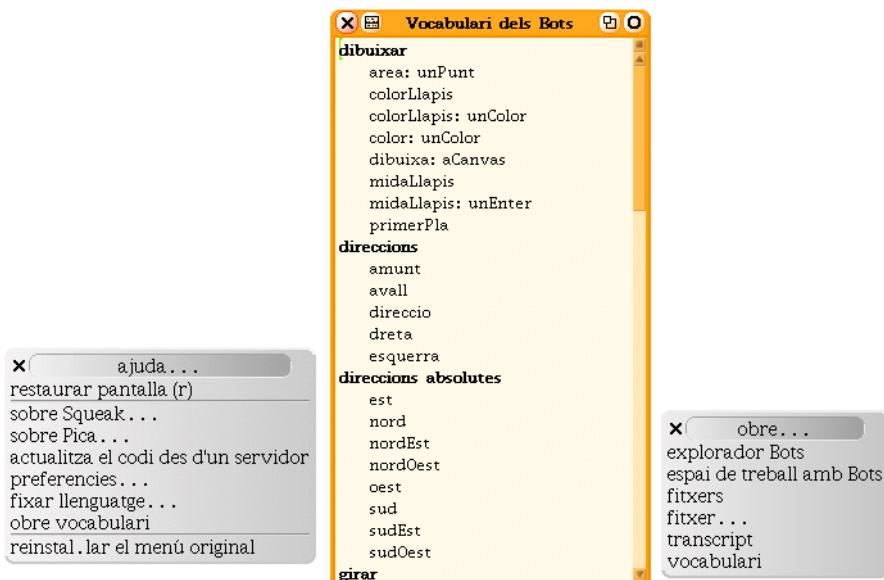
## Interaccionar amb Squeak

La interacció amb Squeak està basada en la suposició que teniu un ratolí de tres botons, tot i que hi ha equivalències per a ratolins de dos botons (Windows) o d'un sol botó (Mac), com mostrem a la taula 5.1. Cada botó està associat amb un cert conjunt d'operacions. El botó esquerre serveix per obtenir menus contextuais i per apuntar i seleccionar, el botó del mig és per manipular finestres (fer que una finestra estigui davant de tot o per moure una finestra), i el botó dret és per obtenir *nances*<sup>1</sup>, que són petites icones rodones que apareixen al voltant dels elements gràfics (veure figura 5.4). Col·lectivament, les nances s'anomenen un *halo*. Les nances són útils ja que permeten que l'usuari interacció directament amb el robot. Els introduirem en detall al proper capítol.

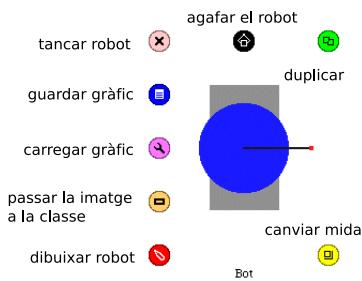
**Taula 5.1 —** Combinacions amb tecles i botons del ratolí

	Apuntar i Seleccionar	Menus Sensibles al Context	Obrir l'Halo
Tres botons	clic esquerre	clic central	clic dret
Windows: 2 botons	clic esquerre	<i>Alt</i> - clic esquerre	clic dret
Mac: 1 botó	clic	<i>Option</i> - clic	<i>Command</i> - clic

<sup>1</sup>Nota del Traductor: No sé encara com traduir *handle* d'una manera satisfactòria, però segons el TERMCAT *nansa* és la traducció correcta en Informàtica.



**Figura 5.3 — Els missatges més importants per als robots.**

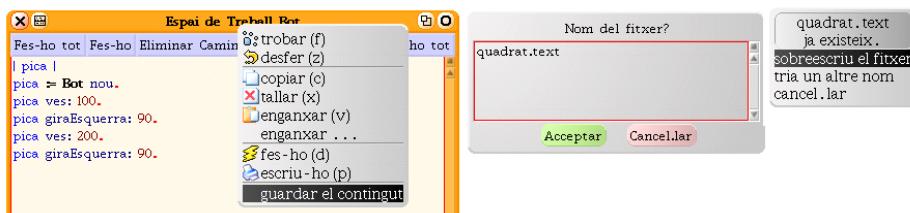


**Figura 5.4 — Fent clic amb el botó de la dreta apareix l'halo.**

## Utilitzar l'Espai de Treball Bot per guardar un *script*

L'Espai de Treball Bot té cinc botons i un menú que us permet guardar *scripts*. El botó **Fes-ho tot** executa tot l'*script* contingut en l'espai de treball. El botó **Fes-ho** executa el tres seleccionat de l'*script* dins l'espai de treball. El botó **Eliminar Camins** esborra els camins dibuixats pels robots sense esborrar els robots. El botó **Eliminar Bots** esborra només els robots sense eliminar els camins. El botó **Eliminar-ho tot** esborra els robots i els camins.

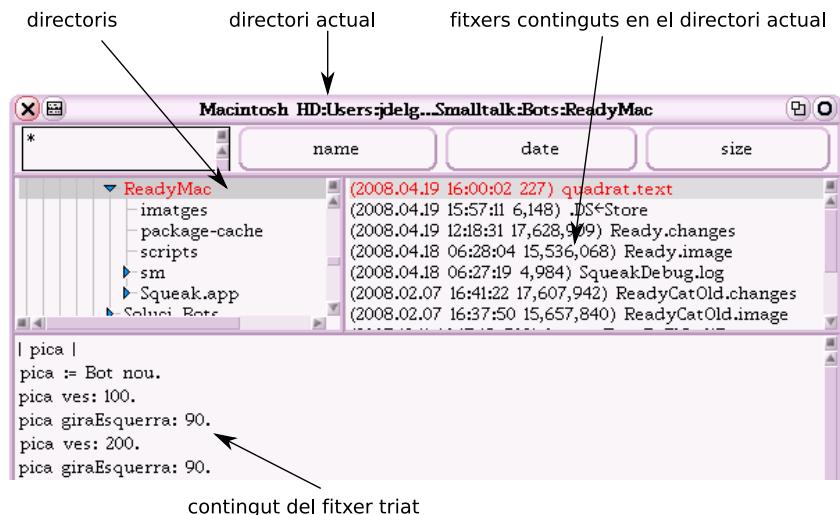
Un cop heu escrit un *script*, pot ser que vulgueu guardar-lo en un fitxer, per, si cal, tornar-lo a utilitzar. L'Espai de Treball Bot us permet guardar i carregar fitxers via el menú de l'espai de treball. Feu clic dins l'espai de treball per fer aparèixer el menú associat, tal com podeu veure a la figura 5.5. L'opció **guardar el contingut** guardarà tot el contingut de l'espai de treball en un fitxer. Triar aquesta opció fa que aparegui una finestra de diàleg, com veieu a la figura. Fixeu-vos que els sistema comprova si ja existeix algun fitxer amb el mateix nom. Si és així, el sistema us dóna l'opció de sobreescrivir el fitxer o guardar-lo amb un altre nom.



**Figura 5.5** — Esquerra: Opcions del menú de l'Espai de Treball Bot. Mig: Especifiquem el nom del fitxer en què volem guardar l'*script*. Dreta: Si el fitxer ja existeix, podeu sobreescrivir'l o reanomenar-lo.

## Carregar un *script*

Per carregar un *script*, heu de fer servir una llista de fitxers (*file list*), una eina que us permet seleccionar i carregar fitxers diferents dins Squeak. Podeu obtenir la llista de fitxers seleccionant l'opció **obre... fitxers** a partir del menú principal. Una llista de fitxers conté diverses subfinestres. La finestra de dalt a l'esquerra us permet explorar discs i carpetes; cada vegada que seleccioneu una opció d'aquesta finestra, la finestra de dalt a la dreta s'actualitza. Mostra tots els fitxers continguts a la carpeta seleccionada dins la finestra de l'esquerra. Quan trieu un fitxer de la finestra de la dreta, la finestra de baix automàticament mostra el seu contingut. La figura 5.6 mostra que estem a la carpeta ReadyMac, dins la qual hem seleccionat el fitxer *quadrat.text*.



**Figura 5.6 — La llista de fitxers està oberta a l'script quadrat.text.**

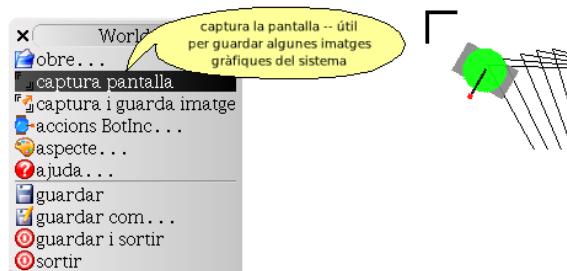
Per carregar un *script*, senzillament heu de copiar el contingut de la finestra de baix utilitzant l'opció del menú **copy**<sup>2</sup> i enganxar-la dins l'Espai de Treball Bot utilitzant l'opció **enganxar**, tal com farieu amb qualsevol editor de text.

## Capturar un dibuix

Per guardar els vostres dibuixos, podeu utilitzar la possibilitat de capturar la pantalla del vostre ordinador. Amb alguns ordinadors això és problemàtic. Per evitar aquests problemes, l'entorn us ofereix un mecanisme simple per capturar la pantalla, tant se val amb quin ordinador treballeu. Obriu el menú principal fent clic al fons de l'entorn. El menú ofereix dues opcions per capturar pantalles, anomenats **captura pantalla** i **captura i guarda imatge**, tal i com es mostra a la figura 5.7.

L'opció més senzilla de les dues és utilitzar **captura i guarda imatge**. Quan seleccioneu aquesta opció, Squeak mostra que està preparat per capturar una imatge canviant la forma del cursor al dibuix d'una cantonada, com es veu a la dreta de la figura 5.7. Poseu el cursor en una cantonada de la regió rectangular que voleu capturar, feu clic, i arrossegueu el ratolí per

<sup>2</sup>Nota del Traductor: La llista de fitxers (*file list*) és una eina de l'entorn Squeak general, no de l'entorn dels bots, per això ni l'eina ni els seus menus han estat traduïts.



**Figura 5.7** — Esquerra: Dues possibilitats per capturar i guardar el dibuix. Dreta: El cursor ha canviat i indica que Squeak està preparat per a la captura. Ara feu clic per indicar una cantonada de la regió rectangular que voleu capturar.

delimitar la regió desitjada. La imatge de la regió es mostrarà a la cantonada superior esquerra de la finestra d'Squeak i Squeak us demanarà el nom sense extensió que ha de donar al fitxer.

Si voleu capturar una regió de la pantalla, utilitzeu l'opció **captura pantalla**. En aquest cas, Squeak no us demanarà guardar el fitxer, enlloc d'això us permetrà capturar una regió de la pantalla, mostrant-la a la cantonada superior esquerra de l'entorn. Ara, podeu manipular aquesta imatge utilitzant l'halo que obteniu fent clic sobre la imatge amb el botó dret del ratolí. Un cert nombre de nanses apareixen al voltant de la imatge, com veieu a l'esquerra de la figura 5.8. Ara podeu fer clic damunt de la nansa vermella, i s'obrirà la possibilitat de fer un cert conjunt d'accions sobre la vostra imatge<sup>3</sup>. Trieu **export...** i Squeak us demanarà en quin format voleu guardar la imatge. Llavors, Squeak us demanarà pel nom del fitxer. Fixeu-vos que podeu importar aquests fitxers dins Squeak arrossegant-los des de l'escriptori de l'ordinador.

## Resultat dels missatges

A Smalltalk els objectes només es comuniquen enviant i rebent missatges a i des d'altres objectes. Un cop un objecte rep un missatge, l'executa, i, addicionalment, retorna un resultat. Un resultat és un objecte que l'objecte receptor retorna a l'objecte que ha enviat el missatge. La comunicació entre objectes mitjançant missatges és similar a la comunicació entre persones mitjançant cartes: Algunes cartes ens poden exigir la realització d'alguna acció (com un avís de l'ajuntament que hem de pagar l'impost de circulació), mentre que altres cartes no les podrem rebre sense una

<sup>3</sup>Nota del Traductor: Altre cop, el menú de l'halo és una eina de l'entorn Squeak general, no de l'entorn dels bots, per això no ha estat traduït



**Figura 5.8** — Obriu l'halo i trieu l'opció **export...** del menú de la nansa vermella per guardar la imatge a disc.

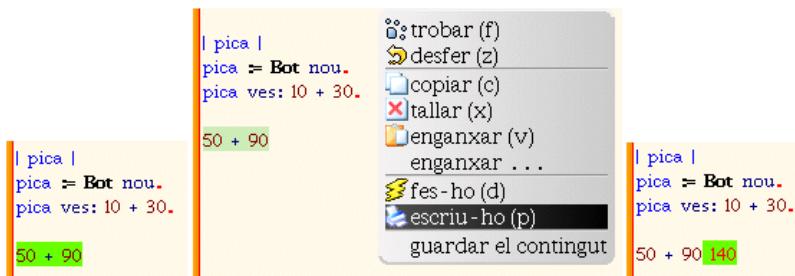
signatura de conformitat (una carta certificada).

A Squeak, el receptor d'un missatge sempre retorna un resultat, que per defecte és el receptor del missatge. Sovint, però, aquest resultat no és gaire interessant. Per exemple, enviar el missatge `ves: 100` a un robot fa que el robot es mogui 100 píxels en la seva direcció actual. No en fem res del resultat retornat, que és el mateix robot, de manera que en aquest cas ignorem el resultat retornat. En molts casos el resultat de l'execució és important. Per exemple, l'expressió `2 + 3` envia el missatge `+ 3` a l'objecte 2, que retorna l'objecte 5. Enviar el missatge `color` a un robot retorna el seu color actual. El resultat d'un missatge pot ser utilitzat en un altre missatge, formant part d'un missatge compost. Per exemple, quan l'expressió `(2 + 3) * 10` s'executa, l'expressió `(2 + 3)` s'executa enviant el missatge `+ 3` a l'objecte 2 i retornant 5. El resultat 5 és utilitzat com l'objecte a què un segon missatge `* 10` és enviat. Així, 5 és el receptor del missatge, i retorna com a resultat 50.

L'entorn Squeak us permet executar missatges sense haver de tractar amb el resultat del missatge, però també us permet executar missatges i escriure el valor retornat per l'enviament del missatge. La propera secció il·lustrarà amb detalls aquesta diferència.

**Nota** Un *resultat* és un objecte que l'objecte receptor retorna a l'objecte que li ha enviat un missatge. Per exemple, `2 + 5` retorna 7 i `pica color` retorna el color de pica, un objecte color.

A la figura 5.9, l'expressió  $50 + 90$  se selecciona, després s'executa utilitzant el menú i el resultat, 140, s'escriu a la pantalla.



**Figura 5.9** — Esquerra: Seleccionar l'expressió  $50 + 90$ . Mig: Obrir el menú. Dreta: Executar el missatge i escriure el resultat.

## Executar un *script*

Hi ha tres maneres d'executar un *script*.

- Utilitzant els botons de l'editor de l'Espai de Treball Bot. Al capítol 2 vau veure una manera senzilla d'executar el vostre primer *script* tot prement el botó **Fes-ho tot** de l'Espai de Treball Bot. Però per executar un *script*, també podeu *seleccionar* amb el ratolí el text que voleu executar (la selecció es torna de color verd) i premer el botó **Fes-ho** de l'Espai de Treball Bot.
- Utilitzant el menú. Seleccioneu el tros de l'*script* que voleu executar, com veieu a la figura 5.10. Obriu el menú fent clic amb el botó del mig del ratolí (o prement la tecla *option* mentre feu clic amb el botó esquerre), i trieu l'opció **fes-ho (d)** o l'opció **escriu-ho (p)** del menú, com heu vist a la figura 5.9.
- Utilitzant les tecles de drecera. Seleccioneu un fragment del text, després premeu **command+D** en un Mac o **alt+D** en un PC.

## Consells

Per seleccionar automàticament tot el text d'un *script*, podeu fer clic al començament del text (abans del primer caràcter), al final del text o a la línia següent a la darrera expressió. Si voleu seleccionar una paraula, podeu fer doble clic a qualsevol lloc de la paraula. Si voleu seleccionar



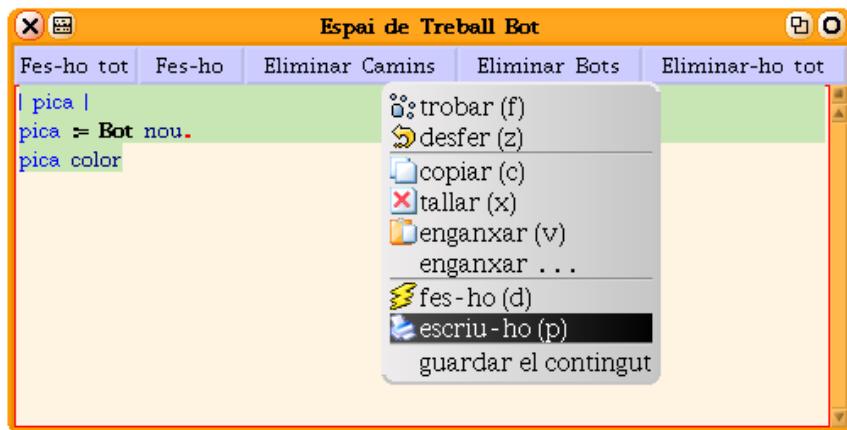
**Figura 5.10** — Seleccionar un tros d'un script i executar-lo explícitament utilitzant el menú

una línia, feu doble clic al començament (abans del primer caràcter) o al final (després del darrer caràcter) de la línia.

## Dos exemples

Quan executeu l'expressió `pica color` utilitzant l'opció **fes-ho (d)** del menú, el missatge `color`, que demana al robot el seu color, és enviat i executat. La sensació que fa, però, és que no ha passat res. Això és perquè no heu demanat al sistema que faci alguna cosa amb el resultat de l'execució del missatge. Si esteu interessats en el resultat del missatge, haurieu de fer servir l'opció del menú **escriu-ho (p)**, com veieu a la figura 5.11. Això provoca que s'executi el fragment de codi seleccionat *i* que s'escrigui el resultat del darrer missatge del codi. A la figura l'expressió `Bot nou` s'executa i el missatge `color` s'envia al nou robot tot just creat. El missatge `color` s'executa i el color del robot receptor és retornat i escrit, com es mostra a la figura 5.12. El text (`TranslucentColor r: 0.0 g: 0.0 b: 1.0 alpha: 0.847`) ens diu que el color del robot és un color transparent amb tres components de color, vermell (`r` de *red*), verd (`g` de *green*) i blau (`b` de *blue*).

Anem a fer una ullada a un exemple final per assegurar-nos que heu entès quan heu d'utilitzar **escriu-ho (p)**. Quan executeu l'expressió `100 + 20` utilitzant l'opció **fes-ho (d)** del menú, el missatge `+ 20` s'envia a l'objecte `100`, al que se suma `20`. Tot i així, no veieu res. Això és normal, ja que en aquest cas l'execució del missatge `+ 20` retorna un nou nombre representant la suma, però no heu demanat a Squeak que l'escrigui. Per veure el resultat, heu d'escriure el resultat



**Figura 5.11** — Obriu el menú i trieu l'opció **escriu-ho (p)** per executar el fragment de codi seleccionat i escriure el resultat retornat.



**Figura 5.12** — El resultat del missatge s'escriu com una representació textual d'un color.

de l'execució del missatge utilitzant l'opció **escriu-ho (p)** del menú. A partir d'ara, escriurem “– Escriure el valor retornat:” per indicar que estem utilitzant la comanda d'escriure per executar una expressió i escriure'n el resultat, com veieu a l'script 5.1. Fixeu-vos que utilitzarem aquesta convenció només si el resultat és important.

**Script 5.1** *Escriure el resultat d'executar una expressió.*

(100 + 20) \* 10  
– *Escriure el valor retornat:* 1200

---

**Important!** Hi ha dues maneres d'executar una expressió: (1) utilitzant l'opció **fes-ho (d)** del menú per executar una expressió, i (2) utilitzant l'opció **escriu-ho (p)** del menú per executar i escriure el resultat retornat.

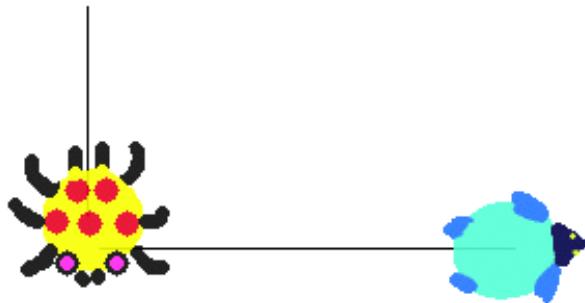
---

## Resum

- Per executar una expressió, selecciona un bocí de text representant una o diverses expressions i prem el botó **Fes-ho** o selecciona l'opció **fes-ho (d)** del menú d'execució.
- Un resultat és un objecte que s'obté d'un missatge. Per exemple, pica color retorna el color del robot.
- Hi ha dues maneres d'executar una expressió, (1) utilitzant l'opció **fes-ho (d)** del menú per executar una expressió, i (2) utilitzant l'opció **escriu-ho (p)** del menú per executar i escriure el resultat retornat.

## Capítol 6

# Divertim-nos amb els robots

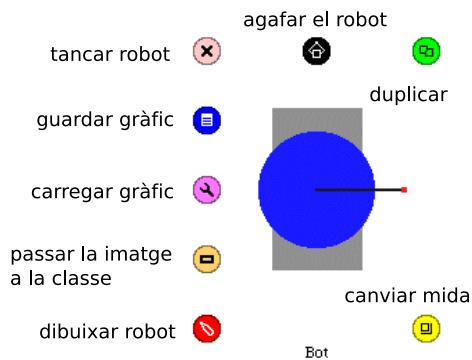


L'aparença bàsica d'un robot és força senzilla. No seria interessant poder crear robots que tinguessin una mica més de gràcia? Afortunadament, això és possible i podeu crear els vostres propis robots. En aquest capítol us ensenyarem com canviar la forma, la mida del llapis i el color dels vostres robots. Podeu fer que el vostre robot sembli un animal, un monstre o fins i tot el famós robot R2D2 de la pel·lícula *Star Wars*.

### Nances del robot

Ja heu après com enviar missatges a un robot fent clic sobre el robot i obrint una bafarada de missatge. Ara aprendreu a accedir i manipular altres funcionalitats dels robots, així podreu moure'ls, duplicar-los o canviar-los l'aparença. Aquestes capacitats extres estan disponibles via

l'halo de nanses, que, tal com ja vam mencionar breument al capítol 5, podeu fer aparèixer fent clic amb el botó dret del ratolí (fent *command-clic* en un Mac) sobre el robot. Les nanses són les icones petites i rodones que envolten el robot com un halo, com mostrem a la figura 6.1. Explicaré les funcions de les diferents nanses a mida que les necessitem. Podeu aconseguir més informació sobre una nansa deixant el ratolí quiet al damunt; tot seguit apareix una bafarada que explica què fa la nansa. Ara per ara, proveu de fer una còpia del robot fent clic sobre la nansa verda (“duplicar el robot”), proveu de moure el robot fent clic sobre la nansa negra (“agafa el robot”) tot arrossegant el robot, o elimineu el robot amb la nansa de color rosa fluix (“tancar el robot”) amb la “X”.



**Figura 6.1** — Fent clic amb el botó dret del ratolí (*command-clic* en un Mac) sobre el robot feu aparèixer l'halo de nanses.

## Mida del llapis i color

Quan, en capítols anteriors, els vostres robots es movien per la pantalla, dibuixaven el seu trajecte amb una línia negra. No esteu, però, limitats al color -per defecte- negre. Podeu canviar el color del llapis d'un robot tot enviant-li el missatge `colorLlapis`: amb un objecte color d'argument. Una de les maneres d'obtenir un objecte color és enviant un missatge a la classe `Color`, que és una fàbrica d'objectes color, amb el nom del color. Per exemple, `Color blau` genera un objecte color de color blau, i `Color groc` en genera un de color groc. Podeu canviar el color del llapis del robot pica i tornar-lo blau amb el missatge `pica colorLlapis: Color blau`. Explicarem més coses sobre colors a la propera secció.

També podeu canviar el gruix del llapis del robot enviant el missatge `midaLlapis:` amb un nombre com a argument. Per exemple, `pica midaLlapis: 5` ordena a `pica` que la mida del seu llapis sigui de 5 píxels d'ample. L'*Script 6.1* dibuixa una línia blava de gruix 5 píxels.

**Script 6.1** *Pica pot dibuixar una línia blava i gruixuda.*

```
| pica |
pica := Bot nou.
pica colorLlapis: Color blau.
pica ves: 100.
pica midaLlapis: 5.
pica ves: 100.
```

L'*Script 6.2* dibuixa unes ulleres de llarga vista incrementant repetidament la mida del llapis.

**Script 6.2** *Pica dibuixa unes ulleres de llarga vista.*



```
| pica |
pica := Bot nou.
pica ves: 40.
pica midaLlapis: 2.
pica ves: 40.
pica midaLlapis: 4.
pica ves: 40.
pica midaLlapis: 6.
pica ves: 40.
```

Podeu canviar el color del robot mateix utilitzant el mètode `color:`. Per exemple, l'enviament de missatge `berthe color: Color groc` fa que el robot `berthe` sigui de color groc. L'*Script 6.3* ordena a `berthe` que canviï el seu color a groc i que vagi endavant 100 píxels, mentre `pica` es queda enrera amb el seu color per defecte i sense moure's.

**Script 6.3** *Berthe canvia el seu color i se'n va a passejar, mentre pica es queda enrera.*

```
| pica berthe |
pica := Bot nou.
berthe := Bot nou.
berthe color: Color groc.
berthe ves: 100.
```

## Més sobre els colors

Tal com hem dit abans, Squeak és un entorn que està construït a partir d'objectes i que utilitza objectes. Per tant, programar en Squeak és crear objectes i enviar-los missatges. En particular, un color és un objecte creat per la classe Color. Per obtenir un objecte color, heu d'enviar un missatge a la classe Color.

Alguns missatges per a la classe Color tenen el mateix nom del color que representen. Per exemple, Color vermell fa que Color creï un objecte color corresponent al color vermell. Aquí teniu una llista dels selectors de missatge predefinits que podeu enviar a la classe Color per crear el color: beigPàlid, blanc, blau, blauClar, blauPàlid, cyanClar, gris, grisClar, grisFosc, grisMoltClar, grisMoltFosc, grisMoltMoltClar, grisMoltMoltFosc, groc, grocClar, grocPàlid, magentaClar, magentaPàlid, marró, marróClar, negre, préssecPàlid, rosa, tanPàlid, taronja, taronjaClar, taronjaPàlid, verd, verdClar, verdPàlid, vermell, vermellClar, vermellMoltPàlid i vermellPàlid<sup>1</sup>.

La classe Color és com una fàbrica de colors de veritat. No només pot crear un nombre força gran de colors estàndard, sinó que també pot crear colors especials combinant quantitats diferents de vermell, verd i blau. La taula 6.1 mostra uns quants exemples de creació de colors utilitzant el missatge `r: quantitat de vermell g: quantitat de verd b: quantitat de blau`. Els arguments que hem de proporcionar al selector de missatge `r:g:b:` han de ser nombres decimals entre 0 i 1 que representen les quantitats de vermell, verd i blau per combinar. Per exemple, l'expressió Color `r: 1 g: 0 b: 0` crea el mateix color vermell que obteniu com a resultat de Color vermell. Utilitzant la mateixa quantitat dels tres colors produeix una mena de gris. Tot uns produeix el color blanc i tot zeros produeix el color negre.

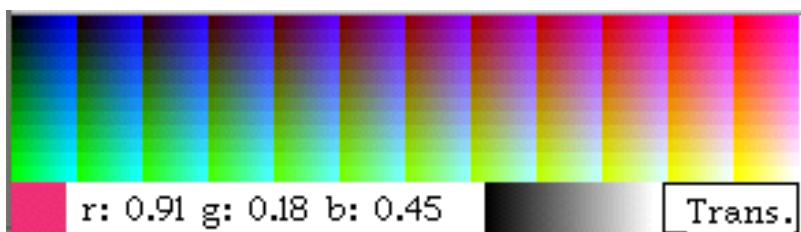
Finalment, el mètode `fromUser` us deixa triar un color d'una paleta de colors que surt en pantalla, i us mostra les quantitats de vermell, verd i blau que formen la corresponent combinació, com il·lustrem a la figura 6.2 (tot i que si la figura és en blanc, negre i gris us haureu d'imaginar els colors). Us caldrà executar l'expressió Color `fromUser` utilitzant l'opció **escriu-ho (p)** per escriure el resultat de l'expressió.

---

<sup>1</sup>Nota del Traductor: Sóc conscient que "pàlid" s'escriu amb ela geminada, malgrat això els noms dels selectors de missatges, pensats per ser escrits en anglès, no permeten segons quins caràcters.

**Taula 6.1 — Crear colors amb Color r:g:b:**

Color	r: (Vermell)	g: (Verd)	b: (Blau)
vermell	1	0	0
gris fluix	0.1	0.1	0.1
groc	1	1	0
blanc	1	1	1
negre	0	0	0
gris	0.5	0.5	0.5
gris pàlid	0.8739	1	0.8348

**Figura 6.2 — Trieu el vostre color de la paleta de colors amb el missatge Color fromUser.**

## Canviar la forma i la mida dels robots

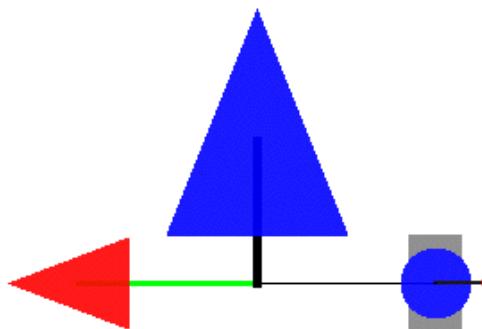
Ja heu vist que podeu canviar el color del robot. Això, però, no és la única cosa que podeu canviar, també podeu canviar-ne la forma. A més de la forma per defecte, dues formes més, un cercle i un triangle, estan disponibles a la fàbrica Bot de robots (també és possible dibuixar les vostres pròpies formes amb una eina de dibuix, com veurem a la propera secció). El missatge `aparentaTriangle` dóna al robot una forma triangular. El missatge `aparentaCercle` dóna al robot una forma circular. Podeu recuperar la forma per defecte amb `aparentaBot`.

Una altra característica del robot que podeu canviar és la mida, amb el missatge `area`: `amplada-alçada` on els valors `amplada-alçada` representen l'amplada i l'alçada del rectangle dins del qual es dibuixarà el robot. L'argument `amplada-alçada` és un parell de nombres, el que en Squeak

s'anomena un *punt*. Està compost per dos nombres separats pel símbol @. Per exemple, el punt 50@100 representa un rectangle de 50 píxels d'amplada i 100 píxels d'alçada.

Per tant, per crear un robot anomenat picagran amb forma triangular i que cèpiga dins un quadrat de dimensions 150@150, hauríeu d'enviar a picagran el missatge aparentaTriangle i després el missatge area: 150@150.

La figura 6.3 mostra algunes formes de robot creades utilitzant les formes circulars i triangulars disponibles a Bot, i l'script 6.4 us ensenya a crear robots d'aquestes formes i mides, i a moure'ls a les posicions mostrades a la figura.



**Figura 6.3 — Els robots poden tenir diverses formes i mides.**

**Script 6.4** Crear robots de diferents formes i mides (cercles i triangles).

```
| pica daly picagran |
pica := Bot nou.
pica aparentaTriangle.
pica oest.
pica color: Color vermill.
pica colorLlapis: Color verd.
pica midaLlapis: 3.
pica ves: 100.
daly := Bot nou.
daly area: 60@60.
daly est.
daly ves: 100.
```

```

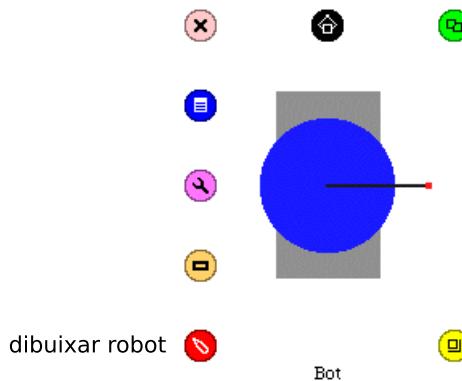
picagran := Bot nou.
picagran aparentaTriangle.
picagran area: 150@150.
picagran midaLlapis: 5.
picagran nord.
picagran ves: 80.

```

## Dibuixar el vostre propi robot

Squeak us permet dibuixar un robot personalitzat. Fins i tot podeu crear un robot que s'assembli a un dels que heu vist al començament d'aquest capítol. Ara us explicarem pas a pas com dibuixar el vostre propi robot.

**Pas 1: Obrir l'eina de dibuix via la nansa vermella.** El primer pas és obrir l'eina de dibuix que ve inclosa amb Squeak. Feu clic amb el botó dret del ratolí (o *command-clic* amb un Mac) per fer aparèixer l'halo al voltant del robot que voleu dibuixar, com es veu a la figura 6.4. Feu clic a la nansa vermella, la que té un llapis pintat. Això obrirà l'editor de dibuixos, que es mostra a la figura 6.5. No feu cas de les altres nanses. Fixeu-vos que si ja heu dibuixat alguna cosa, aquesta es mostrerà dins l'eina de dibuix.

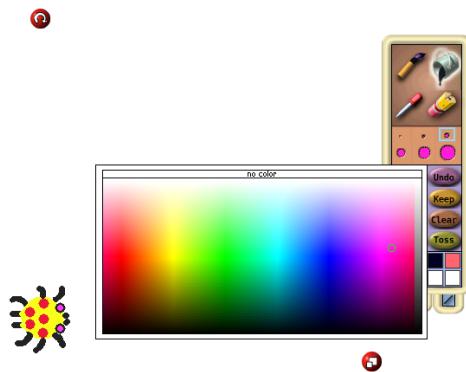


**Figura 6.4** — Feu clic amb el botó dret del ratolí (o *command-clic*) per fer aparèixer l'halo. Trieu l'editor de dibuixos amb la nansa vermella.



**Figura 6.5 — L'editor de dibuixos.**

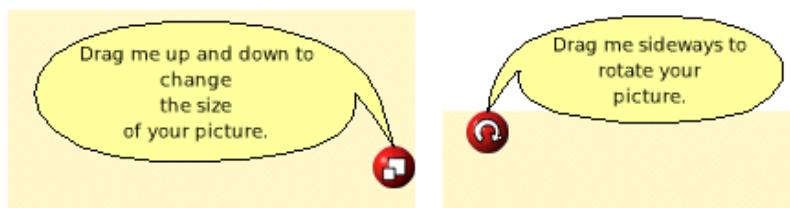
**Pas 2: Dibuixeu la nova aparença del robot.** El segon pas és dibuixar un gràfic nou per al vostre robot. Dibuixeu el vostre robot apuntant a la dreita, com veieu a la figura 6.6. L'editor de dibuixos té les opcions usuals dels programes de gràfics: triar la mida del pinzell, omplir una certa regió, repetir una regió seleccionada o triar el color amb què es dibuixa. L'eina de dibuix també té dos botons (mostrats a la figura 6.7, per girar i fer zoom amb el dibuix).



**Figura 6.6 — Aquest robot sembla una aranya amb taques.**

**Pas 3: Guardeu el vostre dibuix.** Un cop us agradi el que heu dibuixat, hauríeu de prémer

el botó **keep**. Això tanca l'eina de dibuix. Ara el vostre robot té l'aparença del que heu dibuixat.



**Figura 6.7 — Els botons per girar i fer zoom.** Esquerra “Arrossegueu-me amunt i avall per canviar la mida del vostre dibuix”. Dreta “Arrossegueu-me de costat a costat per girar el vostre dibuix”.

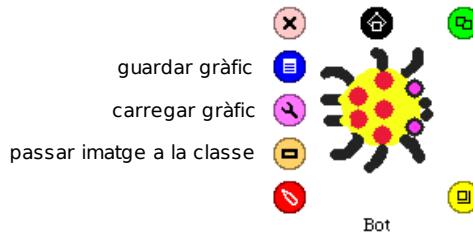
## Guardar i restaurar dibuixos

Si heu passat molta estona dibuixant un robot i el voleu guardar per fer-lo servir altres vegades, podeu guardar-lo en un fitxer. Un cop guardat, podreu carregar-lo en diferents entorns i compartir-lo amb els vostres amics. Podeu fins i tot començar a construir una mena de biblioteca de dibuixos de robot. Ara us ensenyarem a guardar i a carregar un dibuix. Després us ensenyarem a associar un gràfic amb un robot o amb la classe Bot, de manera que els nous robots ja siguin creats amb l'aparença del gràfic que heu dibuixat. Començarem mostrant-vos com realitzar totes aquestes manipulacions interaccionant directament amb els robots, i després veurem com escriure *scripts* per fer tot això automàticament.

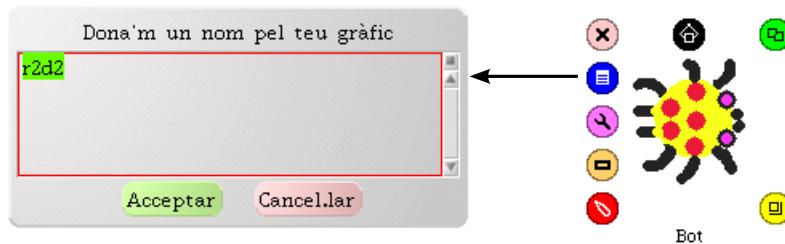
### La nansa “Guardar Gràfic”

Per guardar un gràfic, senzillament feu clic a la nansa blava, la que té la icona del fitxer (figura 6.8). L'hem fet de color blau per fer-vos pensar en un llac congelat: guardar el gràfic “congela” la forma del vostre robot per preservar-la. El sistema us demanarà que doneu un nom al gràfic guardat, com mostrem a la figura 6.9. Aquesta operació guardarà el vostre gràfic en un fitxer, a la mateixa carpeta on teniu la imatge de Squeak, amb el nom que heu introduït i amb extensió .frm.

Podeu invertir l'operació i carregar un gràfic fent clic damunt de la nansa rosa, la que té dibuixada una eina. Hem triat el color rosa per fer-vos pensar en tornar el robot a la vida. Quan feu clic a la nansa rosa, el sistema us demanarà el nom del gràfic que voleu carregar. El robot prendrà l'aparença del gràfic que tot just heu carregat.



**Figura 6.8 — El robot ara sembla una aranya. Apunta cap a la dreta**

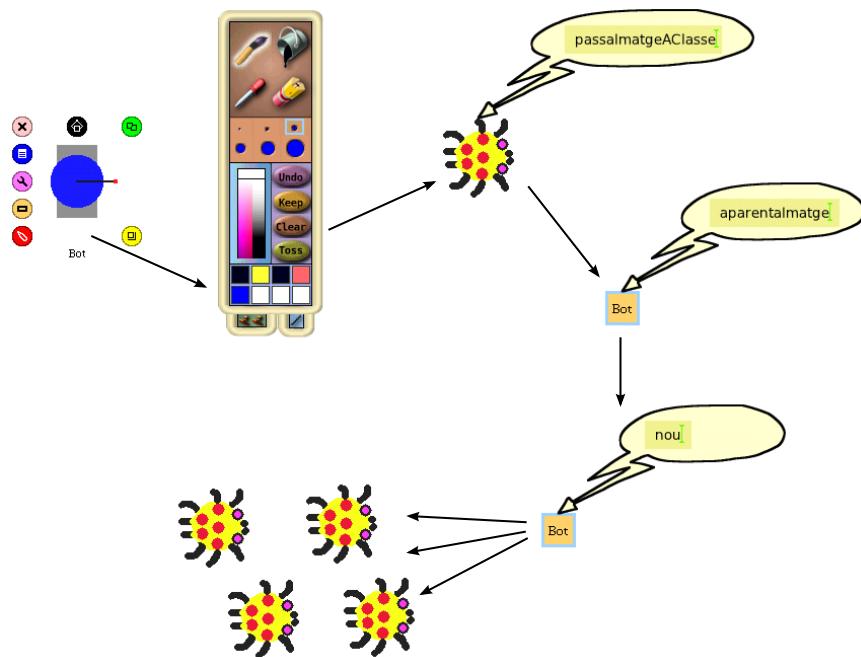


**Figura 6.9 — Fer clic a la nansa blava fa que el sistema us demani un nom.**

## Re-equipar la fàbrica de robots

Heu dibuixat i guardat una bonica aranya plena de taques, i us agradaria que la fàbrica de robots fes un robot amb aquest gràfic, però quan li dieu a la classe Bot que creï un nou robot, en crea un amb el gràfic per defecte. Per fer possible que la classe Bot creï el vostre robot aranya, li heu de dir al robot que li passi la imatge a la classe, utilitzant el missatge `passalimatgeAClasse`. Després d'haver enviat aquest missatge, en crear un robot nou i demanar-li que apparenti la imatge, apparentarà el dibuix que tot just li ha estat passat a la classe.

Una altra manera d'obtenir el mateix resultat és enviar el missatge `aparentalimatge` o qualsevol dels missatges `aparenta` a la classe Bot mateixa. En fer això, la classe es configurarà per crear nous robots amb el nou gràfic o la nova forma. Per exemple, si envieu el missatge `aparentaCercle` a la classe Bot, tots els robots nous tindran forma de cercle. Per tant, si voleu que la classe Bot creï robots amb forma d'aranya, heu de (1) crear un robot, (2) dibuixar l'aranya o carregar-ne una de guardada, (3) passar la imatge de l'aranya a la classe, i (4) dir-li a la classe de fer robots amb aquella imatge tot enviant-li el missatge `aparentalimatge`. Aleshores tots els robots nous semblaran una aranya amb taques, com veieu a la figura 6.10.



**Figura 6.10** — Passar una imatge a la classe Bot i enviar-li el missatge aparentamatge fa que tots els robots nous siguin creats amb l'aparença de la imatge.

### Operacions gràfiques utilitzant *scripts*

També podeu escriure un *script* per carregar i guardar gràfics i associar-los amb un sol robot o una classe.

L'*Script* 6.5 crea dos robots i carrega un gràfic per a cada un d'ells de dues maneres diferents. Després de crear pica, el missatge carregamatge li és enviat, que resulta en la petició a l'usuari del nom de la imatge a carregar. Aleshores es crea berthe, i li enviem el missatge carregamatge: 'aranya', la qual cosa canvia la seva imatge a aquella emmagatzemada en el fitxer aranya.frm.

Fixeu-vos que és important la distinció entre els missatges carregamatge i carregamatge: '*nomFitxer*'. El primer no té cap paràmetre, i es demana a l'usuari el nom del fitxer que s'ha de carregar. El segon missatge té un paràmetre, '*nomFitxer*', que representa el nom del fitxer on hi ha la imatge que volem carregar, entre cometes simples i sense extensió.

Podeu guardar la imatge utilitzant els missatges guardamatge i guardamatge: '*nomFitxer*'. Primer li enviem a berthe el missatge guardamatge, i es demana a l'usuari el nom del fitxer on

vol guardar la imatge. Finalment, enviem a pica el missatge guardalmatge: 'aranya2', que guarda la seva aparença en un fitxer de nom aranya2.frm.

**Script 6.5 Dues maneres de carregar i guardar dibuixos de robots.**

```
| pica berthe |
pica := Bot nou.
pica carregalmatge.           "Es demana a l'usuari el nom de la imatge a carregar"
berthe := Bot nou.
berthe carregalmatge: 'aranya'. "El paràmetre dóna el nom del fitxer a carregar"
berthe guardalmatge.          "Es demana a l'usuari el nom del fitxer on es guardarà la imatge"
pica guardalmatge: 'aranya2'. "El paràmetre dóna el nom del fitxer on es guardarà la imatge"
```

De la mateixa manera que vosaltres podeu guardar i carregar gràfics associats a un robot individual, també podeu guardar i carregar els gràfics associats a la classe Bot. Els mateixos missatges que hem fet servir amb els robots es poden utilitzar amb la classe. Només cal enviar-los a Bot i no a pica o a berthe. L'*Script 6.6* associa primer la imatge aranya.frm amb la classe Bot. Després la imatge es guarda amb un altre nom, aranyaBot.frm.

Podeu utilitzar també els mètodes carregalmatge i guardalmatge (no hi ha dos punts, per tant no cal argument), que demanen a l'usuari el nom del fitxer. L'expressió Bot initialitzalmatge torna la classe Bot a l'estat original on genera robots amb l'aparença per defecte. Això implica que en executar l'*script* ho feu en un escenari predictable.

**Script 6.6 Carregar i guardar un gràfic associat a la classe Bot**

```
| pica berthe |
Bot initialitzalmatge.        "Elimina qualsevol gràfic prèviament associat amb la classe Bot "
berthe := Bot nou.            " berthe té l'aspecte per defecte dels robots"
Bot carregalmatge: 'aranya'. "La imatge a aranya.frm s'associa amb la classe Bot"
Bot aparentalmatge.
pica := Bot nou.              "El robot pica té l'aparença d'una aranya"
Bot guardalmatge: 'aranya3'. "La imatge de l'aranya es guarda amb el nom aranya3.frm "
```

Els següents *scripts* (*Scripts 6.7 i 6.8*) donen per fet que els fitxers luth.frm i aranya.frm són a la mateixa carpeta que el fitxer imatge de l'entorn. Aquests fitxers s'inclouen amb la distribució del programari en aquest llibre.

L'*Script 6.7* utilitza el mètode carregalmatge: per associar una imatge amb un robot, i el mètode aparentalmatge per ordenar al robot que la seva aparença sigui la de la imatge amb què se l'ha

associat. Després que el robot pica es creï, se li demana que canviï la seva aparença per la de la seva imatge associada (pica aparentalmatge). Com que no hi ha cap gràfic associat amb pica, demanar-li que canviï la seva aparença no produceix cap efecte i pica es queda com està. Aleshores la imatge del fitxer luth.frm s'associa amb pica enviant-li el missatge carregalmatge: 'luth'. Ara, quan s'envia a pica el missatge aparentalmatge la seva aparença canvia, i es transforma en una tortuga de mar. A la darrera línia de l'script, una imatge diferent s'associa amb pica. Com que ja li havíem enviat el missatge aparentalmatge, a partir d'aquest moment el robot tindrà l'aparença de qualsevol imatge que se li associi. Per tant, després d'executar l'expressió pica carregalmatge: 'aranya', pica semblarà una aranya.

L'script continua creant un robot nou, berthe. Com que la classe Bot no té cap imatge associada, berthe tindrà l'aparença per defecte dels robots, i si li enviem el missatge aparentalmatge res no canvià, ja que no té associada cap imatge.

#### **Script 6.7 Canviar l'aparença d'un robot**

```
| pica berthe |
Bot inicialitzalmatge.
```

pica := Bot nou.  
pica aparentalmatge.

*"No hi ha cap imatge carregada o creada, així doncs, no canvia res"*

pica carregalmatge: 'luth'.  
pica aparentalmatge.

*"Carrega una imatge i demana al robot que la utilitzi per a la seva aparença"*

pica carregalmatge: 'aranya'.  
*"Carrega una altra imatge, i, com que el missatge aparentalmatge ja ha estat enviat, pica canvià la seva aparença per la de la nova imatge"*

berthe := Bot nou.  
berthe aparentalmatge.

*"Quan berthe es crea, la seva aparença és la imatge per defecte, i com que no s'ha carregat cap imatge a la classe, el missatge aparentalmatge no provoca cap canvi"*

A l'Script 6.8 veiem com indicar a la classe Bot que tots els robots que es creïn nous han de tenir una determinada aparença. En aquest cas, el missatge carregalmatge: 'nomFitxer' és enviat a la classe Bot i no a cap robot particular, com havíem fet a l'Script 6.7. De la mateixa manera que la paraula "lliure" evoca significats diferents en un pres i en un taxista, diferents objectes

i classes poden entendre de manera diferent el mateix missatge. Això és degut al fet que cada objecte o classe té el seu propi mètode per respondre a un missatge donat, i aquests mètodes pel mateix missatge poden ser diferents. En el cas que ens ocupa, carregalmatge: té comportaments diferents dependent de si és rebut per la classe Bot o per un robot, que és una *instància* de la classe. Quan és rebut per la classe Bot, el missatge carregalmatge: *nomFitxer* fa que la classe carregui el gràfic del fitxer i l'associï amb el procés de creació de robots, de manera que noves instàncies de robots poden fer servir el nou gràfic. Quan és un robot qui rep el missatge, només aquell robot en particular podrà fer servir el gràfic carregat.

#### **Script 6.8** Asociar un gràfic amb la classe Bot

```
| pica berthe daly yrtle |
Bot carregalmatge: 'aranya'.
berthe := Bot nou.
berthe aparentalmatge.
"berthe, com a instància de la classe Bot, sembla ara una aranya."
```

```
daly := Bot nou.
daly aparentalmatge.
"daly també sembla una aranya"
```

```
pica := Bot nou.
pica carregalmatge: 'luth'.
pica aparentalmatge.
"Però un robot particular pot encara canviar la seva aparença;
pica ara sembla una tortuga"
```

```
pica aconsegueixImatgeDeClasse.
"pica obté el seu gràfic de la classe Bot; ara sembla una aranya altre cop"
```

```
Bot carregalmatge: 'luth'.
Bot aparentalmatge.
yrtle := Bot nou.
"Ara la classe crearà robots que semblen tortugues marines"
```

L'Script 6.8 comença carregant un dibuix nou d'un fitxer i associant-lo amb la classe Bot. Aleshores es crea un nou robot, berthe, i li enviem el missatge per a que utilitzi el nou gràfic.

Crear un altre robot, daly, i enviar-li el missatge `aparentalmatge` també li fa adoptar la imatge associada amb la classe.

Tots els robots creats poden ser obligats a semblar una aranya. Un robot particular, però, pot adquirir la seva pròpia aparença enviant-li el missatge `carregalmatge`: '*nomFitxer*', com el robot pica de l'*script*. La imatge associada al robot substitueix a la imatge de la classe. El missatge `aconsegueixImatgeDeClasse` fa possible restaurar el gràfic associat a la classe. La darrera seqüència de missatges mostra que podem associar un nou dibuix a la classe, substituint el gràfic ja associat. Enviar el missatge `carregalmatge`: '*nomFitxer*' a la classe Bot associa el gràfic del fitxer *nomFitxer.frm* a Bot. Aleshores, enviar el missatge `aparentalmatge` assegura que els robots nous tindran per defecte l'aparença del gràfic ara associat a la classe. Per tant, el robot yertle tindrà l'aparença d'una tortuga.

## Resum

Mètode	Descripció	Exemple
aparentaCercle	La forma del receptor passa a ser un cercle.	Bot nou apparentaCercle
aparentaBot	La forma del receptor passa a ser un robot.	Bot nou apparentaBot
aparentaTriangle	La forma del receptor passa a ser un triangle.	Bot nou apparentaTriangle
aparentalmatge	La forma del receptor passa a ser un gràfic que hem dibuixat.	Bot nou apparentalmatge
aparentaCercle	Enviat a Bot, els nous robots es crearan amb forma de cercle.	Bot apparentaCercle
aparentaBot	Enviat a Bot, els nous robots es crearan amb forma de robot.	Bot apparentaBot
aparentaTriangle	Enviat a Bot, els nous robots es crearan amb forma de triangle.	Bot apparentaTriangle
aparentalmatge	Enviat a Bot, els nous robots tindran la forma del gràfic que haguem dibuixat o carregat.	Bot apparentalmatge
carregalmatge: ' <i>nomFitxer</i> '	Carrega el fitxer <i>nomFitxer.frm</i> a la classe o al robot.	Bot carregalmatge: 'aranya' o bé berthe carregalmatge: 'aranya'
carregalmatge	Demana a l'usuari pel nom del fitxer que hauria de ser carregat a una classe o un robot.	Bot carregalmatge o bé berthe carregalmatge
guardalmatge: ' <i>nomFitxer</i> '	Guarda el gràfic de la classe o del robot al fitxer anomenat <i>nomFitxer.frm</i> .	Bot guardalmatge: 'aranya' o bé berthe guardalmatge: 'aranya'
guardalmatge	Guarda el gràfic de la classe o del robot demanant a l'usuari el nom del fitxer	Bot guardalmatge o bé berthe guardalmatge
colorLlapis: <i>unColor</i>	Canvia el color del llapis.	berthe colorLlapis: Color blau
midaLlapis: <i>unNombre</i>	Canvia la mida del llapis. Per defecte és 1.	berthe midaLlapis: 3
color: <i>unColor</i>	Canvia el color del receptor al color especificat.	berthe color: Color groc
area: <i>unPunt</i>	Canvia la mida del receptor a les dimensions especificades per <i>unPunt</i> , on <i>unPunt</i> ve donat per <i>am@al</i> , on <i>am</i> és l'amplada i <i>al</i> és l'alçada.	berthe area: 80@100
passalmatgeAClasse	Passa el gràfic del receptor a la classe. Després d'aquest missatge, els robots creats tindran com a dibuix el gràfic del robot actual.	berthe passalmatgeAClasse
aconsegueixImatgeDeClasse	Aconsegueix el gràfic de la classe. Després d'aquest missatge, el receptor tindrà l'aparença dels robots que serien creats per la classe	berthe aconsegueixImatgeDeClasse

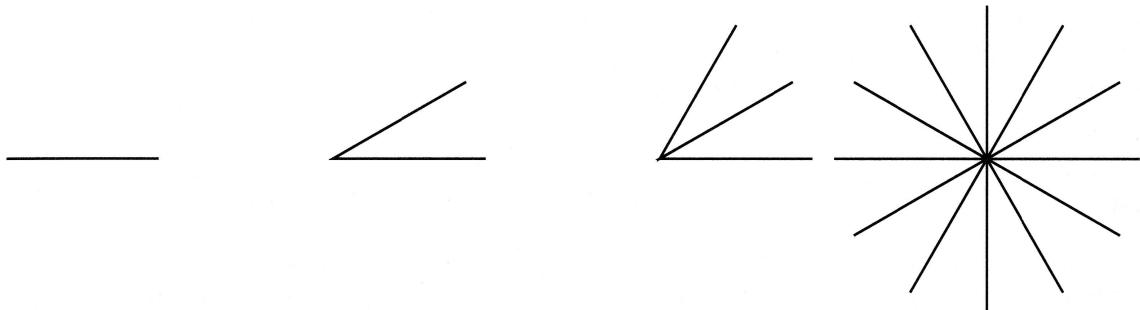
## **Part II**

# **Conceptes elementals de programació**



## Capítol 7

# Repetir



Ara per ara, segur que penseu que la feina del programador de robots és força tediosa. Probablement teniu bones idees per fer dibuixos interessants, però no us veieu amb cor d'escriure els *scripts* per fer-los realitat, ja que sembla que el nombre de línies que heu d'escriure creix a mida que la complexitat del dibuix augmenta. En aquest capítol aprendreu a utilitzar *repeticions* (també conegeudes com *bucles*) per reduir el nombre d'expressions que cal donar als robots. Les repeticions us permeten *repetir una seqüència d'expressions*. Amb un bucle, l'*script* per dibuixar un hexàgon o un octàgon no és pas més gran que l'*script* per dibuixar un quadrat.

### Ha nascut una estrella

Ens agradaria que un robot dibuixés una estrella, semblant a la que teniu dibuixada al començament d'aquest capítol. Ho farem de la seguent manera: començant des del que serà el centre

de l'estrella, el robot dibuixa una línia, retorna al centre, gira un determinat angle, dibuixa una altra línia, i així fins que l'estrella s'acabi. L'*Script 7.1* crea un robot que dibuixa una línia de 70 píxels de llarg i retorna al lloc d'on ha sortit. Fixeu-vos que després de tornar al lloc de partida, el robot fa mitja volta, i així apunta en la direcció original.

**Script 7.1** *Dibuixar una línia i tornar al lloc de partida.*

```
| pica |
pica := Bot nou.
pica ves: 70.
pica giraEsquerra: 180.
pica ves: 70.
pica giraEsquerra: 180.
```

Per dibuixar una estrella, hem de repetir part de l'*script 7.1* i després ordenar al robot que giri un angle determinat. Si volem dibuixar una estrella de sis puntes, l'angle ha de ser de 60 graus, ja que girant 60 graus cada vegada resultarà en  $360/60 = 6$  branques. L'*Script 7.2* ens mostra com hem de fer això per obtenir una estrella amb 6 branques sense utilitzar repeticions.

**Script 7.2** *Una estrella de sis puntes sense utilitzar repeticions.*

```
| pica |
pica := Bot nou.
pica ves: 70.
pica giraEsquerra: 180.
pica ves: 70.
pica giraEsquerra: 180.
pica giraEsquerra: 60.
pica ves: 70.
pica giraEsquerra: 180.
pica ves: 70.
pica giraEsquerra: 180.
pica giraEsquerra: 60.
pica ves: 70.
pica giraEsquerra: 180.
pica ves: 70.
pica giraEsquerra: 180.
pica giraEsquerra: 60.
pica ves: 70.
```

```

pica giraEsquerra: 180.
pica ves: 70.
pica giraEsquerra: 180.
pica giraEsquerra: 60.
pica ves: 70.
pica giraEsquerra: 180.
pica ves: 70.
pica giraEsquerra: 180.
pica giraEsquerra: 60.
pica ves: 70.
pica giraEsquerra: 180.
pica ves: 70.
pica giraEsquerra: 180.
pica giraEsquerra: 60.

```

Com podeu veure, després ser creat, pica repeteix les mateixes cinc línies de codi sis vegades (ho hem mostrat alternant el tipus normal amb lletra itàlica). Sembla una pèrdua de temps haver d'escriure el mateix tros de codi una vegada i una altra. Imagineu la mida del vostre *script* si volguéssiu una estrella amb 60 branques, com la que mostrem a l'Experiment 7-1. El que ens cal és una manera de repetir una seqüència d'expressions.

## Bucle al rescat

La solució del nostre problema és utilitzar un *bucle*. Hi ha diferents tipus de bucles, i el que introduirem ara us permet repetir una seqüència donada de missatges un nombre donat de vegades. El mètode<sup>1</sup> vegadesRepetir: repeteix una seqüència d'expressions un cert nombre de vegades, com veieu a l'*Script* 7.3. Aquest *script* defineix la mateixa estrella que l'*Script* 7.2, però amb molt menys codi. Fixeu-vos que les expressions que s'han de repetir estan col·locades entre claudàtors.

**Script 7.3** *Dibuixar una estrella de sis puntes utilitzant un bucle.*

```

| pica |
pica := Bot nou.
6 vegadesRepetir:
  [ pica ves: 70.
    pica giraEsquerra: 180.

```

---

<sup>1</sup>Nota del Traductor: timesRepeat: en la versió original de Smalltalk

```
pica ves: 70.
pica giraEsquerra: 180.
pica giraEsquerra: 60. ]
```

**Important!** *n* vegadesRepetir: [ seqüència d'expressions ] repeteix una seqüència d'expressions *n* vegades.

El mètode vegadesRepetir: permet repetir una seqüència d'expressions, i a Smalltalk, aquesta seqüència d'expressions delimitada per claudàtors s'anomena *bloc*.

El missatge vegadesRepetir: s'envia a un nombre enter, el nombre de vegades que la seqüència s'hauria de repetir. A l'*Script* 7.3 el missatge vegadesRepetir: [...] és enviat a l'enter 6. Res de nou aquí; ja vau veure un missatge enviat a un nombre enter quan vam parlar de la suma: el segon enter és enviat al primer, que retorna la suma dels dos.

Finalment, fixeu-vos que el nombre receptor del missatge vegadesRepetir: ha de ser un *nombre enter* ja que en un bucle, igual que a la vida real, no està clar què vol dir executar una seqüència d'expressions, diguem, 0.2785 vegades.

L'argument de vegadesRepetir: és un bloc, és a dir, una seqüència d'expressions envoltada per un parell de claudàtors. Recordeu del capítol 2 que l'argument d'un missatge consisteix en la informació que l'objecte receptor necessita per executar el missatge. Per exemple, [ pica ves: 70. pica giraEsquerra: 180. pica ves: 70. ] és un bloc compost de tres expressions: pica ves: 70, pica giraEsquerra: 180 i pica ves: 70.

**Important!** L'argument de vegadesRepetir: és un *bloc*, això és, una seqüència d'expressions entre claudàtors.

## Repeticions en marxa

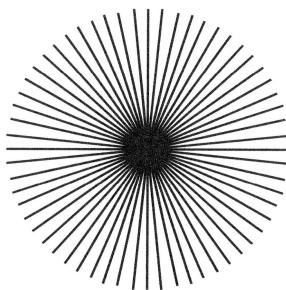
Si compareu l'*Script* 7.1 amb les expressions a la repetició de l'*Script* 7.3, veureu que hi ha una expressió més: pica giraEsquerra: 60, que crea l'angle entre branques adjacents. Hi ha una relació molt simple entre el nombre de branques i l'angle que el robot hauria de girar abans de dibuixar la propera branca: Per a una estrella completa, la relació entre l'angle i el nombre de repeticions ha de ser  $\text{angle} \times n = 360$ .

Si voleu adaptar l'*Script* 7.3 per dibuixar una estrella amb qualsevol nombre de branques, hauríeu de canviar el nombre de vegades que el bucle es repeteix substituint 6 pel nombre enter

apropiat. Fixeu-vos que l'angle 60 hauria de ser modificat si voleu generar una estrella completa.

### Experiment 7-1 (una estrella amb seixanta branques)

Feu un *script* que dibuixi una estrella amb 60 branques.



```
| pica |  
pica := Bot nou.  
6 vegadesRepetir: [ pica ves: 70.  
pica giraEsquerra: 180.  
pica ves: 70.  
pica giraEsquerra: 180.  
pica giraEsquerra: 60. ]
```

```
| pica |  
pica := Bot nou.  
6 vegadesRepetir: [ pica ves: 70.  
pica giraEsquerra: 180.  
pica ves: 70.  
pica giraEsquerra: 180.  
pica giraEsquerra: 60. ]
```

**Figura 7.1** — Fer sagnat de blocs fa molt més fàcil identificar repeticions. Esquerra: Sense sagnat. Dreta: Amb sagnat.

### Sagnat de codi

Els programes Smalltalk es poden escriure de moltes maneres diferents, i, en particular, el sagnat des del marge esquerre no té cap efecte en l'execució del codi. Diem que el sagnat no té cap

efecte en el “significat” sintàctic del programa. Tot i això, utilitzar un sagnat clar i consistent ajuda el lector a entendre el codi.

Us suggerim que seguiu la convenció que hem fet servir a l'*Script* 7.3 en donar format a l'expressió `vegadesRepetir:`. La idea és que el bloc d'expressions que es repeteixen, situades entre claudàtors, formi un rectangle textual i visual. Aquesta és la raó que el bloc comenci amb el claudàtor esquerre a la línia que segueix el misatge `vegadesRepetir:` i d'alinear totes les expressions dins del bloc amb un tabulador. El claudàtor dret al final indica que el bloc s'ha acabat. La figura 7.1 us hauria de convèncer que el codi sagnat és més fàcil de llegir que el codi sense sagnar.

La discussió sobre el format que caldria donar als programes podria allargar-se eternament, ja que gent diversa llegeix el codi de manera diferent. La convenció que proposem pretén ajudar a identificar expressions repetides.

## Dibuixar figures geomètriques regulars

Podeu dibuixar moltes figures diferents només repetint seqüències de missatges, com el quadrat del capítol 4 (que repetim aquí com a *Script* 7.4).

**Script 7.4** *El primer quadrat de pica.*

```
| pica |
pica := Bot nou.
pica ves: 100.
pica giraEsquerra: 90.
```

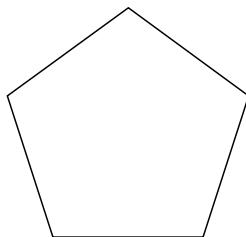
### Experiment 7-2 (un quadrat utilitzant un bucle)

Transformeu l'*Script* 7.4 perquè dibuixi el mateix quadrat utilitzant el mètode `vegadesRepetir:`. Ara hauríeu de ser capaços de dibuixar altres polígons regulars, fins i tot els que tenen un gran nombre de costats.

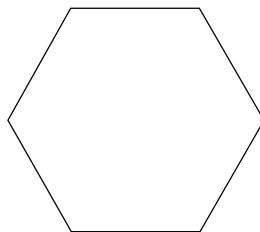
---

**Experiment 7-3 (pentàgon regular)**

Dibuixeu un pentàgon regular utilitzant el mètode vegadesRepetir::

**Experiment 7-4 (hexàgon regular)**

Dibuixeu un hexàgon regular utilitzant el mètode vegadesRepetir::

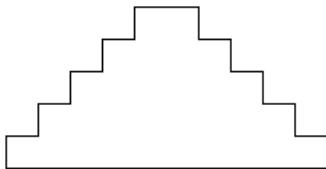


Un cop ho hagueu dominat, intenteu dibuixar un polígon regular amb un nombre molt gran de costats. És possible que hagueu de reduir la mida dels costats per fer que la figura pugui caber a la pantalla. Quan el nombre de costats sigui gran i la mida dels costats sigui petita, el polígon us semblarà un cercle.

## Redescobrir les piràmides

Recordeu com vau codificar el perfil de la piràmide de Saqqara a l'Experiment 3-5? Podeu simplificar el vostre codi utilitzant un bucle, com a l'*Script* 7.5.

**Script 7.5** Un script amb repeticions per dibuixar la piràmide.

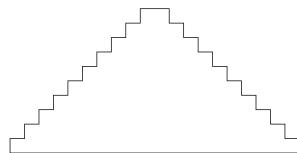


```
| pica |
pica := Bot nou.
5 vegadesRepetir:
  [ pica nord.
    pica ves: 20.
    pica est.
    pica ves: 20 ].
5 vegadesRepetir:
  [ pica ves: 20.
    pica sud.
    pica ves: 20.
    pica est ].
pica oest.
pica ves: 200.
```

Ara ja hauríeu de generar piràmides amb un nombre arbitrari d'esglaons utilitzant el mateix nombre d'expressions, només canviant els nombres de l'*script*.

### Experiment 7-5 (piràmide amb deu esglaons)

Dibuixeu una piràmide amb 10 esglaons utilitzant una variació de l'*Script 7.5* .




---

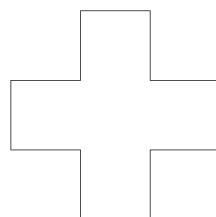
És possible que vulgueu generar piràmides amb un nombre encara més gran d'esglaons. S'hauria d'ajustar la mida dels esglaons si voleu que càpiguen a la pantalla.

## Més experiments amb repeticions

Com ja heu comprovat, generar una piràmide esglaonada implica la repetició d'un bloc de codi que dibuixa dos segments de línia. Un cop identifiqueu els elements que cal repetir, podeu produir dibuixos complexos a partir de la repetició de dibuixos elementals. Els experiments següents il·lustren aquest principi.

### Experiment 7-6 (una creu suïssa)

Dibuixe l'esbós de la creu suïssa mostrada a la figura utilitzant giraEsquerra: o bé giraDreta:, i vegadesRepetir:

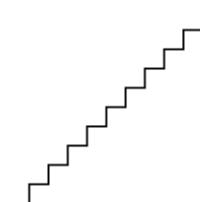


---

### Experiment 7-7 (una escala)

---

Dibuixe l'escala que veieu a la figura.

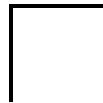


**Experiment 7-8 (escala sense alçadors)**

Dibuixeu l'escala estilitzada –sense alçadors– que veieu a la figura.

**Experiment 7-9 (una grapa)**

Dibuixeu l'element gràfic, semblant a una grapa, mostrat a la figura.

**Experiment 7-10 (una pinta)**

Transformeu l'element gràfic que heu fet a l'Experiment 7-9 per construir la pinta mostrada a la figura.



**Experiment 7-11 (una escala de mà)**

Transformeu l'element gràfic que heu fet a l'Experiment 7-9 per construir una escala de mà.

**Experiment 7-12 (quadrats que fan tombarelles)**

Ara que ja sou uns mestres en repeticions utilitzant vegadesRepetir:, feu un bucle per dibuixar els quadrats fent tombarelles que teniu al començament del capítol 4.

## Resum

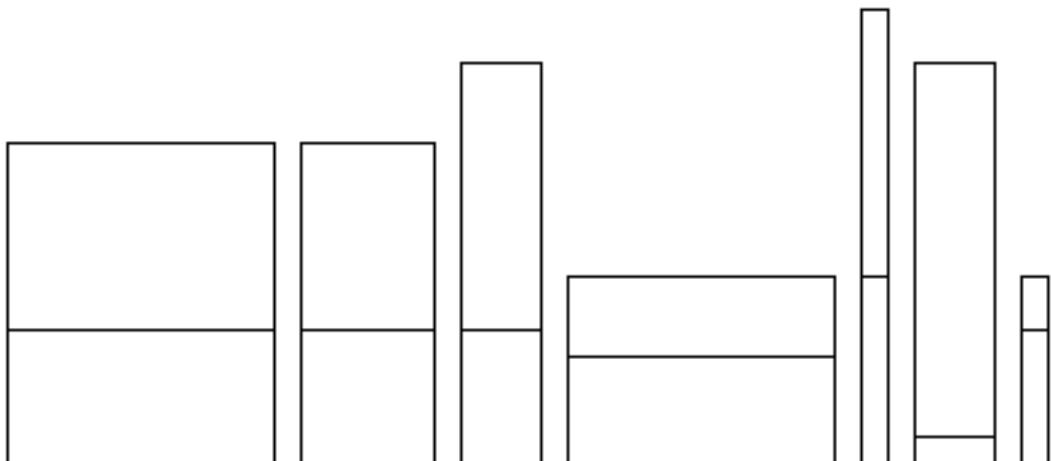
En aquest capítol heu après a programar repeticions utilitzant el mètode *n* vegadesRepetir::

Mètode	Sintaxi	Descripció	Exemple
vegadesRepetir:	<i>n</i> vegadesRepetir: [ seqüència d'expressions ]	Repteix una seqüència d'expressions <i>n</i> vegades	10 vegadesRepetir: [ pica ves: 10. pica salta: 10 ]



# Capítol 8

## Variables



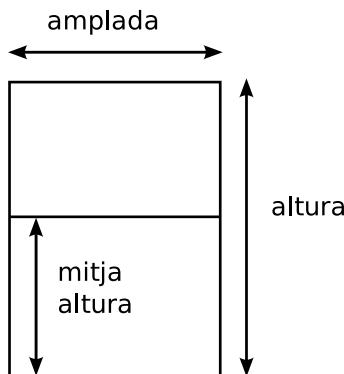
Hom sempre posa nom a les coses. Per exemple, donem nom a l'altra gent, als gossos i als cotxes. Quan fem això, estem *associant* algun objecte, ésser o idea amb una paraula o símbol. Un cop hem fet aquesta associació, podem fer servir aquesta paraula per *referir-nos* a o interactuar amb l'objecte associat a ella. Un nom pot durar tota una vida, o el podem descartar després de no gaire temps. Algunes vegades, els noms fan referència a d'altres noms. Per exemple, un actor té usualment diversos noms: el seu nom, el nom d'actor i el nom del personatge que està interpretant. En un llenguatge de programació, també ens cal poder donar nom a les coses, i per a això utilitzem *variables*.

En aquest capítol veureu les variables, que són receptacles per a objectes, i com les podeu

utilitzar per simplificar els vostres programes. De fet, les variables són sovint imprescindibles en els programes. Finalment, a mesura que la complexitat dels problemes que anem trobant s'incrementi, veureu que caldrà expressar dependències entre variables. Per exemple, l'amplada d'un rectangle pot ser dos terços de la seva altura. En aquest capítol us mostrarem com fer servir les variables per expressar dependències entre diverses quantitats.

## Per cortesia de la lletra A

Tal com vau fer al capítol 3, imagineu que voleu utilitzar un robot per escriure lletres de l'alfabet. La lletra A més aviat primitiva que dibuixarem tot seguit està caracteritzada per la seva *altura*, la seva *amplada* i la seva *mitja altura*, que és l'altura a què dibuixem la línia del mig de la lletra A (figura 8.1).



**Figura 8.1** — La forma de la lletra A es caracteritza per la seva *altura*, la seva *amplada* i la seva *mitja altura*.

### Experiment 8-1

Feu un *script* que dibuixi una lletra A d'altura 100 píxels, amplada 70 píxels i mitja altura 60 píxels

## Variacions sobre el tema de la A

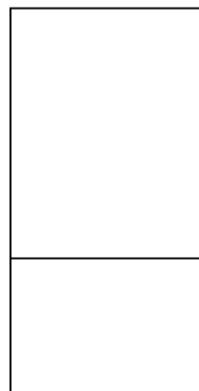
L'*script* que heu escrit per a l'Experiment 8-1 s'hauria d'assemblar a l'*Script 8.1*.

**Script 8.1 Una A per a l'Experiment 8-1**

```
| pica |
pica := Bot nou.
pica nord.
pica ves: 100.
pica est.
pica ves: 70.
pica sud.
pica ves: 100.
pica oest.
pica salta: 70.
pica nord.
pica ves: 60.
pica est.
pica ves: 70.
```

**Experiment 8-2 (frAnkenstein)**

Modifiqueu l'*Script* 8.1 per dibuixar una lletra A monstruosa d'altura 200 píxels, amplada 100 píxels i mitja altura 70 píxels, com veieu a la figura.



---

En modificar l'*Script* 8.1 per a l'Experiment 8-2, heu vist que per fer una A de mida diferent, heu hagut de canviar els valors que representen l'altura, l'amplada i la mitja altura de la lletra *a tot arreu* on aquests nombres apareixen, i a més ho heu hagut de fer *síncronament*. Per síncronament volem dir “en l'ordre correcte”; és a dir, el 100 hauria de ser un 200, el 70 hauria de ser un 100 i el 60 hauria de ser un 70, sense barrejar-los.

### Experiment 8-3

Modifiqueu l'*Script* 8.1 per dibuixar altres A de mides diferents. Proveu de reproduir algunes de les As que apareixen a la figura del començament d'aquest capítol.

---

Fent l'Experiment 8-3 heu arribat sens dubte a la conclusió que canviar els valors de les característiques de la lletra A a tot arreu és força avorrit. És més, hauria de ser obvi que fent aquests canvis us arrisqueu a confusions, a obrir algun valor o a fer un canvi de valor incorrecte. El resultat no serà gens semblant al que teniu al cap quan feu l'*script*. Podeu imaginar que en programes complexos, canviar els valors un per un tal com ho heu fet pot ser força problemàtic.

## Variables al rescat

Fer molts canvis per construir lletres de diverses mides i formes és molt pesat i pot introduir errors. Ens cal una solució que ens estalviï barrejar els nombres que representen les diferents característiques d'una lletra i que també ens permeti fer alteracions sense haver de canviar els valors a tot arreu. De fet, voldríem ser capaços de fer el següent:

- *Declarar* l'altura, l'amplada i la mitja altura de la lletra A només un cop a tot l'*script*.
- *Referir-nos* a aquests valors si els necessitem.
- *Canviar* els valors, si cal.

Aquestes tres coses són exactament el que una variable ens permet fer! Fantàstic, no? Una variable és un *nom* a què *associoem un valor*. Cal *declarar* la variable i *associar-li* un valor nou. Aleshores podrem *referir-nos* a la variable i obtenir-ne el valor. També és possible *modificar* el valor associat a la variable i *assignar-li* un valor nou. El valor d'una variable pot ser un nombre, una col·lecció d'objectes, o fins i tot un robot. Tot seguit mostrarem com declarar, associar un valor i utilitzar una variable.

---

**Important!** Una variable és un *nom* a què *associoem un valor*. Cal que *declarem* la variable i li *associoem* un valor nou. Aleshores podrem *referir-nos* a la variable i obtenir-ne el valor. També és possible *modificar* el valor associat a la variable i associar-hi un valor nou.

---

## Declarar una variable

Abans d'utilitzar una variable, l'hem de *declarar*; és a dir, li hem de dir a Squeak el nom de la variable que volem fer servir. Declarem variables posant-les entre barres verticals | |, com veieu en l'exemple, que declara tres variables altura, amplada i mitjaAltura:

```
| altura amplada mitjaAltura |
```

Per ser precís, les barres verticals | | declaren variables *temporals*, que són variables que existeixen només mentre s'executa l'*script*.

## Assignar un valor a una variable

Abans de fer servir una variable és quasi sempre necessari donar-li un valor. Associar un valor a una variable s'anomena *assignar* un valor a una variable. Smalltalk utilitza el símbol (compost per un parell de caràcters) := per assignar un valor a una variable. A l'*script* següent, després de declarar tres variables, assignem 100 a la variable altura, 70 a la variable amplada i 60 a la variable mitjaAltura. Quan assignem un valor a una variable per primera vegada, diem que la *inicialitzem*:

```
| altura amplada mitjaAltura |
altura := 100.
amplada := 70.
mitjaAltura := 60.
```

---

**Important!** El símbol := assigna un valor a una variable. Per exemple, altura := 120 assigna el valor 120 a la variable altura, mentre que longitud := 120 + 30 assigna el resultat de l'expressió 120 + 30, és a dir, 150, a la variable longitud

---

Quan assignem un valor a una variable per primer cop, diem que l'estem *inicialitzant*.

---

### Referir-nos a les variables

Per referir-nos al valor assignat a una variable –també parlem d'*utilitzar* una variable– senzillament escrivim el seu nom. Al següent *script*, després d'haver *declarat* les variables a la primera línia, la variable *altura* s'*inicialitza* amb el valor 100 a la línia 3 i s'*utilitza* a la línia 5 per dir-li al robot que vagi endavant el nombre de píxels associat amb la variable *altura*, que aquí és 100.

```
| pica altura |
pica := Bot nou.
altura := 100.
pica nord.
pica ves: altura.
```

**Important!** En general, cal *declarar* i *inicialitzar* una variable abans d'utilitzar-la.

## I què passa amb Pica?

Ho heu encertat! pica també és una variable. El que passa és que el seu valor és un robot. Per tant, `| pica |` declara una variable anomenada pica. L'expressió `pica := Bot nou` inicialitza la variable amb un valor, que aquí és un robot nou. Aleshores utilitzem aquest robot tot enviant-li missatges via la variable pica, per exemple, `pica ves: 100`.

## Utilitzar variables

Ara explorarem els beneficis d'utilitzar variables, i us mostrarem algunes propietats interessants de les variables. En particular, us ensenyarem la potència que aconseguim en expressar relacions entre variables.

Introduint variables a l'*Script* 8.1, obtenim l'*Script* 8.2

### **Script 8.2 Una A amb variables.**

```

| pica altura amplada mitjaAltura |
pica := Bot nou.
altura := 100.                      "inicialitzem les variables"
amplada := 70.
mitjaAltura := 60.
pica nord.
pica ves: altura.                  "després utilitzem les variables"

```

```

pica est.
pica ves: amplada.
pica sud.
pica ves: altura.
pica oest.
pica salta: amplada.
pica nord.
pica ves: mitjaAltura.
pica est.
pica ves: amplada.

```

Estareu d'acord que canviar els valors de les variables una vegada és més senzill que canviar els noms repartits per tot l'script. Canvieu alguns valors, així us n'acabareu de convèncer. Hauríeu de ser capaços de dibuixar totes les A de la figura del començament del capítol. Ara, si voleu canviar les característiques de la vostra A, només us cal re-initialitzar les variables tot canviant els valors a les línies 3, 4 i 5, com veieu a l'Script 8.3. El dibuix resultant us el mostrem a la figura 8.2.

**Script 8.3** Una lletra A modificada.

```

| pica altura amplada mitjaAltura |
pica := Bot nou.
altura := 30.                               "inicialitzem les variables"
amplada := 200.
mitjaAltura := 10.
...

```



**Figura 8.2** — Una A curta i ampla creada simplement inicialitzant `altura := 30`, `amplada := 200` i `mitjaAltura := 10`.

Utilitzant variables podeu crear fàcilment moltes lletres diferents, i en un futur podreu escriure programes per resoldre molts problemes interessants. Tornem un moment enrere i considerem la potència que ens proporcionen les variables.

## La potència de les variables

Els experiments que trobareu en el que queda de capítol il·lustren la potència de les variables. Les variables us permeten donar nom a una entitat, ja sigui un robot, una longitud o pràcticament qualsevol altra cosa. Després podeu utilitzar els noms en lloc de repetir els valors que heu associat amb aquests noms. Les variables fan que els vostres *scripts* siguin molt més senzills de canviar, ja que només cal re-inicialitzar les variables amb valors diferents.

A més a més, una variable pot contenir una gran varietat de tipus de valors. Fins ara, heu assignat robots i nombres a les variables, però també podeu assignar colors (per exemple, `robotColor := Color groc`), sons, o qualsevol objecte d'Squeak.

Fixeu-vos també que les variables fan els vostres *scripts* més llegibles i fàcils d'entendre. Per quedar-ne convençuts, compareu l'*Script* 8.1 i l'*Script* 8.2. El fet d'utilitzar variables amb noms com "altura" o "amplada" us ajuda a entendre com es dibuixa la lletra.

## Expressar les relacions entre variables

En els vostres experiments amb la lletra A, segurament heu trobat que algunes de les lletres són més reconeixibles que altres. Les lletres de l'alfabet haurien de mantenir certes proporcions per ser llegibles. En particular, les dimensions que descriuen una lletra concreta no es trien a l'atzar, sinó que existeixen unes certes proporcions.

Podem decidir, per a la nostra lletra A, que l'amplada hauria de ser dos terços de l'altura, i que la mitja altura hauria de ser tres cinquenes parts de l'altura. Podem expressar aquestes relacions utilitzant variables com ensenyem a l'*Script* 8.4. Com podeu veure, el valor d'una variable no ha de ser un simple nombre, sinó que pot ser el resultat d'un càlcul complex.

### **Script 8.4 Relacions entre variables: una primera aproximació.**

```
| pica altura amplada mitjaAltura |
pica := Bot nou.
altura := 120.
amplada := 120 * 2 / 3.
mitjaAltura := 120 * 3 / 5.
...
```

Mirant l'*Script* 8.4, aviat us adoneu que no és òptim. Les relacions entre les variables no s'expressen entre les variables, sinó en termes del valor 120 (a les línies 3, 4 i 5). Aquest valor hauria de ser canviat manualment si volguéssiu fer una altra lletra A amb les mateixes proporcions. Vosaltres voleu ser capaços de canviar el valor d'altura i que els valors d'amplada i mitjaAltura canviïn automàticament. La solució està a utilitzar la variable altura en lloc del valor 120, com veieu a l'*Script* 8.5. En aquest *script*, els valors de les variables amplada i mitjaAltura realment depenen del

valor d'altura. Això funciona ja que el valor d'una variable pot ser expressat en termes d'altres variables. L'expressió amplada := altura \* 2 / 3 expressa que l'amplada de la lletra és igual a dos terços de la seva altura.

**Script 8.5 Relacions entre variables:** Les variables amplada i mitjaAltura depenen d'altura.

```
| pica altura amplada mitjaAltura |
pica := Bot nou.
altura := 120.
amplada := altura * 2 / 3.
mitjaAltura := altura * 3 / 5.
pica nord.
...
```

Inicialitzeu abans d'utilitzar!

L'única restricció que heu de considerar en expressar relacions entre variables és que una variable utilitzada a la definició d'una altra variable hauria de tenir un valor. Per exemple, a l'*Script 8.5*, la variable altura té 120 com a valor d'inicialització, que és utilitzat per amplada i mitjaAltura en calcular els seus valors inicials. Per veure què pot anar malament, a l'*Script 8.6* la variable altura no ha estat inicialitzada, i per tant quan provem d'inicialitzar la variable amplada a altura \* 2 / 3 obtenim un error ja que altura no té cap valor. Parlarem més sobre els errors al capítol 15.

**Script 8.6 Inicialització problemàtica d'amplada.**

```
| altura amplada mitjaAltura |
amplada := altura * 2 / 3.
altura := 120.
mitjaAltura := altura * 3 / 5.
```

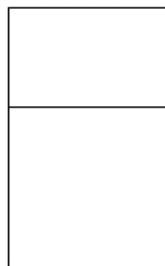
## Experimentar amb variables

Els experiments que segueixen us ajudaran a guanyar experiència amb les variables.

### Experiment 8-4 (el rectangle d'or)

Un rectangle d'or és un rectangle tal que un dels seus costats és aproximadament 1.6 vegades la longitud de l'altre. El nombre 1.6 és una aproximació de la "raó àuria": el nombre. Una propietat interessant d'aquests

rectangles és que si talleu un quadrat del rectangle, com veieu a la figura, el rectangle que queda també és un rectangle d'or. Ara podeu tallar un quadrat d'aquest rectangle més petit i obtenir un rectangle d'or encara més petit, i així *ad infinitum*. Les dimensions d'un rectangle d'or són agradables a la vista, i des de temps antics, artistes i arquitectes han utilitzat la raó àuria a la seva feina. Feu un *script* que dibuixi un rectangle d'or. Podeu expressar el nombre en Smalltalk de la forma  $1 + 5 \sqrt{5} / 2$



### Experiment 8-5 (*Scripts* que no funcionen)

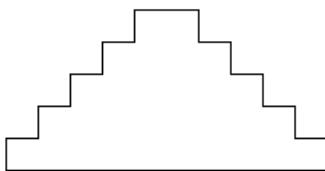
Expliqueu per quina raó cap dels *scripts* següents és capaç de dibuixar una lletra A d'altura 120 píxels.

pica altura	pica altura
pica := Bot nou.	pica := Bot nou.
altura := 120.	pica nord.
pica nord.	pica ves: altura.
pica ves: 100.	pica est.
pica est.	pica ves: 70.
pica ves: 70.	pica sud.
pica sud.	pica ves: altura.
pica ves: 100.	pica oest.
pica oest.	pica salta: 70.
pica salta: 70.	pica nord.
pica nord.	pica ves: 50.
pica ves: 50.	pica est.
pica est.	pica ves: 70.
pica ves: 70.	

## Les piràmides redescobertes

A l'*Script* 7.5, al capítol 7, vam definir el perfil de la piràmide de Saqqara com a l'*Script* 8.7.

**Script 8.7** *La piràmide de Saqqara.*



```
| pica |
pica := Bot nou.
5 vegadesRepetir:
  [ pica nord.
    pica ves: 20.
    pica est.
    pica ves: 20 ].
5 vegadesRepetir:
  [ pica ves: 20.
    pica sud.
    pica ves: 20.
    pica est ].
pica oest.
pica ves: 200.
```

### Experiment 8-6 (una piràmide amb un nombre variable d'esglaons)

Modifiqueu l'*Script* 7.5, introduint la variable `nombreEsglaons` per representar el nombre d'esglaons que hauria de tenir la piràmide.

---

### Experiment 8-7 (una piràmide amb una mida variable de l'esglaó)

Modifiqueu l'*Script* de l'Experiment 8-6 introduint la variable `midaEsglaons` per representar la mida dels esglaons.

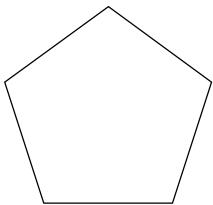
---

## Polígons automatitzats utilitzant variables

La utilització de variables simplifica enormement la definició d'*scripts* on alguna de les *variables* depèn d'altres variables. En aquesta secció, veureu com l'ús de variables proporciona força avantatges a l'hora de tractar amb bucles. El capítol 10 aprofundirà més en la potència que la combinació de variables i bucles proporciona als vostres programes.

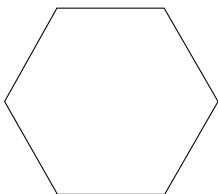
Tornem als Experiments 7-3 i 7-4, en què un Bot havia de dibuixar un pentàgon regular i un hexàgon regular. El codi necessari apareix aquí com a *Scripts* 8.8 i 8.9.

**Script 8.8** *Un pentàgon regular.*



```
| pica |
pica := Bot nou.
5 vegadesRepetir:
  [ pica ves: 100.
    pica giraEsquerra: 72 ].
```

**Script 8.9** *Un hexàgon regular.*



```
| pica |
pica := Bot nou.
6 vegadesRepetir:
  [ pica ves: 100.
    pica giraEsquerra: 60 ].
```

Per transformar el primer *script* en el segon només cal canviar el nombre de costats (diguem-ne *s*) i el gir (diguem-ne *T*) de manera que el producte  $s \times T$  sigui igual a 360. No estaria bé poder escriure un *script* on només calgués canviar un nombre, diguem-ne el nombre de costats, ja que aquest és el paràmetre més senzill? Això ho podem fer mitjançant les variables. Proveu de trobar-ne una solució.

L'*Script* 8.10 resol el problema. Fa possible dibuixar un polígon regular amb qualsevol nombre de costats, només canviant un nombre. Proveu-lo abans de continuar la discussió.

### **Script 8.10 Dibuixar un polígon regular.**

```
| pica costats angle |
pica := Bot nou.
costats := 6.
angle := 360 / costats.
costats vegadesRepetir:
  [ pica ves: 100.
    pica giraEsquerra: angle].
```

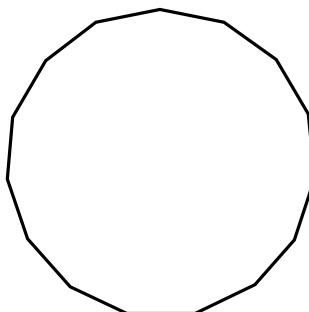
Aquest *script* introduceix dues noves variables, *costats* i *angle*, que s'utilitzen per guardar el nombre de costats i l'angle. Aleshores, l'expressió *costats* := 6 assigna el valor 6 a la variable *costats*, i l'expressió *angle* := 360 / *costats* assigna un valor a la variable *angle* que és el resultat de dividir 360 pel valor contingut a la variable *costats*. El valor de la variable *angle* s'utilitza com a argument del missatge *giraEsquerra:* enviat al robot dins del bloc que es repeteix.

## **Polígons regulars de mida fixa**

Segur que us heu adonat que si l'*Script* 8.10 s'executa amb un nombre molt gran de costats, la figura no cap a la pantalla. El proper experiment us demana que arregleu aquest problema reduint la longitud dels costats a mesura que creix el nombre de costats.

### **Experiment 8-8 (controlar els costats del polígon)**

Modifiqueu l'*Script* 8.10 de manera que la mida del polígon regular romangui més o menys constant mentre canvia el nombre de costats. Ajuda: Introduïu una variable *longitudTotal* que s'inicialitza a un valor determinat fix, i feu que cada costat del polígon tingui longitud igual a *longitudTotal* dividit pel nombre de costats.



## Resum

- Una variable és un *nom* a què podem *associar un valor*. Les variables han de ser *declarades* i cal que els *assignem* un valor. Aleshores, podem *referir-nos* a la variable per obtenir-ne el *valor associat*. També és possible *modificar* el valor associat a una variable i assignar-hi un nou valor.
- Una variable pot ser utilitzada a qualsevol lloc on el seu valor pugui ser utilitzat.
- El primer cop que assignem un valor a una variable, diem que la *inicialitzem*.
- El símbol `:=` assigna un valor a una variable. Per exemple, `altura := 120` assigna el valor 120 a la variable altura, mentre que `longitud := 120 + 30` assigna el resultat de l'expressió  $120 + 30$ , és a dir, 150, a la variable longitud.
- Una variable ha de ser, en general, *declarada* i *inicialitzada* abans de ser utilitzada.

Ús de la variable	Sintaxi	Descripció	Exemple
declaració de variables	<code>  nomVariable  </code>	Declarar nomVariable com a variable.	<code>  pica altura  </code>
assignació de variables	<code>nomVar := expressió</code>	Assignar el valor d'expressió a la variable nomVar Utilitzar el nom d'una variable en una expressió.	<code>longitud := 40</code> <code>longitud := 30 + 20</code> <code>pica ves: longitud</code>
		Canviar el valor d'una variable utilitzant el seu valor actual.	<code>long := long + 10</code>

## Capítol 9

# Aprofundir en les variables

En el capítol anterior vam introduir les variables. En aquest capítol aprofundirem en el tema de manera que pugueu aprendre més sobre com s'utilitzen les variables. Podeu optar per ometre aquest capítol en una primera lectura, ja que és una mica tècnic.

Abans d'il·lustrar amb detalls com funcionen les variables, voldríem posar èmfasi altre cop en la importància de triar noms adequats per a les variables.

### Anomenar les variables

En triar un nom per a una variable, podeu triar el nom que vulgueu. Tot i així, és molt important donar a les vostres variables noms amb un cert significat, ja que això us ajudarà a escriure programes i a entendre els programes que ja heu escrit. Per il·lustrar-ho, llegiu l'*Script 9.1*, que és l'*Script 8.10* escrit utilitzant noms de variables sense cap significat.

**Script 9.1** *Les variables sense cap significat fan que un programa sigui difícil d'entendre.*

```
| x y z |
x := Bot nou.
y := 6.
z := 360 / y.
y vegadesRepetir:
  [ x ves: 100.
    x giraEsquerra: z].
```

Com podeu descobrir tot provant-lo, l'*Script* 9.1 és perfectament correcte, i Squeak pot executar-lo sense cap problema. Però estem segurs que teniu clar quin dels dos *Scripts* 8.10 i 9.1 és més comprensible.

A Smalltalk el nom d'una variable pot ser qualsevol seqüència de caràcters alfabètics i numèrics (*alfanumèrics*) començant per una lletra minúscula. És habitual utilitzar noms de variable llargs que indiquen clarament quina funció té la variable dins del programa. Fent-ho així us ajuda, a vosaltres i a d'altres programadors, a entendre millor els vostres *scripts*.

Ser capaç d'entendre el que fa un programa és molt important, ja que, com veureu més tard, un programa usualment implica una combinació de diversos *scripts*<sup>1</sup>. Quan arribi el moment d'entendre un *script* escrit per algú altre, o fins i tot un d'escript per vosaltres mateixos fa temps, estareu agraiats d'haver agafat l'hàbit de triar noms de variables significatius.

Ara que ja us hem convençut de la importància de triar els noms de les variables amb significat, discutirem en detall les variables.

## Variabes com a contenidors

Les variables són receptacles que fan referència als objectes. Una manera habitual d'explicar la noció de variable és utilitzar una notació gràfica en què les variables es representen com a contenidors. Veiem aquesta idea a l'*Script* 9.2 i la figura 9.1.

A l'*Script* 9.2 (pas (a) a l'*script* i a la figura), dues variables han estat declarades, pica i pablo. Al pas (b) creem un robot i l'assignem a la variable pica; és a dir, la variable pica ara fa referència al robot que tot just hem creat. Després, a (c), assignem el valor de la variable pica a la variable pablo, i per tant pablo fa referència (també es diu *apunta*) al mateix objecte a què apunta la variable pica, el robot que acabem de crear. Quan enviem un missatge utilitzant qualsevol de les dues variables, estem enviant un missatge al mateix objecte, és a dir, al robot creat al pas (b), ja que les dues variables fan referència al mateix objecte. Per tant, a (d), el missatge enviat a pica fa que el robot es mogui 100 píxels, mentre que l'enviament de missatge (e), que va dirigit a pablo, fa que *el mateix* robot canviï el seu color a groc.

Una altra manera de pensar això és considerar que el robot té dos noms: pica i pablo. És tot just com si la mare de l'artista Pablo Picasso hagués dit "Picasso, vine aquí" (pica ves: 100), i després hagués dit "Pablo, aquí tens la teva camisa groga. Posa-te-la" (pablo color: Color groc).

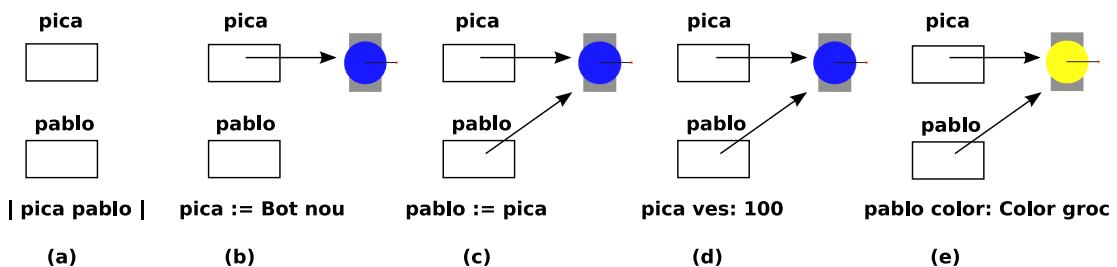
**Script 9.2** Dues variables apunten al mateix robot.

- (a) | pica pablo |
- (b) pica := Bot nou.
- (c) pablo := pica.

---

<sup>1</sup>Això és una simplificació. Aviat sabreu més coses sobre els mètodes, que són els autèntics fonaments de la programació amb objectes

- (d) pica ves: 100.  
 (e) pablo color: Color groc.



**Figura 9.1** — (a) Dues variables són declarades, pica i pablo. (b) Es crea un robot i la variable pica s'associa amb aquest objecte nou. (c) S'assigna a la variable pablo el valor de la variable pica; ara pablo apunta al mateix objecte que apunta pica. (d) Quan enviem el missatge ves: 100 utilitzant pica, el robot es mou. (e) Quan enviem el missatge color: Color groc a pablo, el mateix robot canvia de color. Resumint, si enviem un missatge a qualsevol de les dues variables, estem enviant aquest missatge al mateix objecte

## Assignació: L'esquerra i la dreta de :=

Hi ha dues maneres diferents d'utilitzar el nom d'una variable en un *script*. De vegades el nom s'utilitza per fer referència al valor, per exemple a expressions de la forma longitudCami + 100 i pica ves: 100. Altres vegades, el nom de la variable s'utilitza per fer referència a la variable mateixa, com a longitudCami := 100 o pica := Bot nou.

La clau per entendre les variables és: La utilització del nom d'una variable sempre fa referència al valor associat amb la variable, *excepte en el cas que la variable estigui a l'esquerra d'una assignació*, és a dir, a l'esquerra del símbol :=. En aquest cas, el nom de la variable representa el contenidor (la variable mateixa) i no el valor de la variable. Una altra manera de dir el mateix és que el valor d'una variable és sempre *llegit*, excepte quan apareix a l'esquerra de :=, en aquest cas és *escrit*, és a dir, canviat. L'*Script 9.3* en mostra un exemple.

**Script 9.3** *La variable longitudCami és escrita a la línia 3 i llegida a la línia 4.*

```
(1) | longitudCami pica |
(2) pica := Bot nou.
(3) longitudCami := 100.
(4) longitudCami + 150.
(5) pica ves: longitudCami.
```

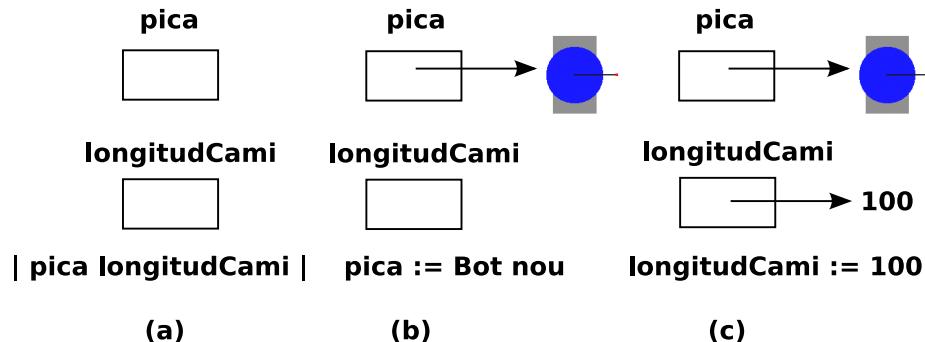
A la línia 3 de l'*Script 9.3*, el nom de variable *longitudCami* apareix a l'esquerra de `:=`, de manera que es refereix al contingut, i el valor 100 és assignat a la variable *longitudCami*. Després que la línia 3 s'hagi executat, la variable *longitudCami* es refereix al nombre 100. A la línia 4, *longitudCami* + 150 no és part d'una assignació, per tant el nom de la variable es refereix al valor de la variable (fixeu-vos que a la línia 4 es fa una suma però el resultat no s'utilitza. Per tant la línia no fa res i podríem eliminar-la). A la línia 5, les dues variables, *pica* i *longitudCami* s'utilitzen pels seus valors, és a dir, els objectes a què fan referència. Per tant la variable *longitudCami* es refereix al valor 100, mentre *pica* fa referència al robot creat més amunt dins l'*script*. Així, la línia 5 té com a efecte que el missatge *ves: 100* s'envia al robot creat a la línia 2.

**Important!** Una variable és un contingut on guardem una referència a un valor, és a dir, a un objecte. Utilitzar una variable retorna un valor excepte quan la variable és a l'esquerra d'una assignació, és a dir, a l'esquerra del símbol `:=`. En aquest cas, el valor de la variable és canviat pel valor de l'expressió a la dreta de l'assignació. Per exemple, *longitudCami* + 150 retorna 150 sumat al *valor* de la variable *longitudCami*, mentre que *longitudCami* `:= 100` *canvia* el valor de *longitudCami* a 100.

## Analitzar alguns *scripts* senzills

Per comprendre millor com manipular les variables, descriurem una sèrie d'*scripts*. Primer, llegiu-vos l'*script* i proveu d'endevinar què fa; aleshores mireu d'avaluar l'*script* amb molta cura per determinar-ne el resultat. Us suggerim que dibuixeu una representació dels continguts com a capsetes si penseu que us pot ajudar, i comproveu els vostres dibuxos amb el que mostrem a la figura. Com veureu, donarem una petita explicació de cada *script*. Fixeu-vos que les explicacions d'un *script* no es repetiran en els *scripts* següents, per tant estudieu-los en ordre, i torneu enrera a *scripts* anteriors per aclarir qualsevol cosa que us semblí confusa.

**Script 9.4** Es declaren i inicialitzen dues variables, i s'utilitzen els seus valors.



```
| pica longitudCami |
pica := Bot nou.
longitudCami := 100.
pica ves: longitudCami
```

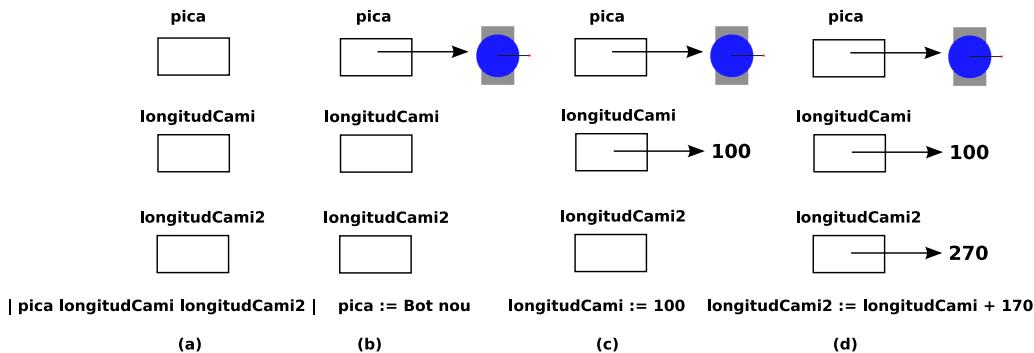
A l'*Script 9.4*, les variables **pica** i **longitudCami** són declarades. Es crea un robot nou i s'assigna a la variable **pica**. Aleshores 100 s'assigna a la variable **longitudCami**. El robot (valor assignat a la variable **pica**) rep el missatge **ves:** amb el valor de la variable **longitudCami** com a argument, que en aquest cas és 100.

**Script 9.5** Com a l'*Script 9.4*, però ara una variable s'utilitza com a part d'una expressió.

```
| pica longitudCami |
pica := Bot nou.
longitudCami := 100.
pica ves: longitudCami + 170
```

A l'*Script 9.5*, s'avalua l'expressió **longitudCami + 170** per determinar el nombre de píxels que el robot s'hauria de moure endavant. Com **longitudCami** ha estat inicialitzada al valor 100 i el seu valor no ha canviat des d'aleshores, el valor de **longitudCami** és 100. Per tant, l'expressió **longitudCami + 170** té valor 270, i el robot es mou endavant 270 píxels. L'expressió **pica ves: longitudCami + 170** a l'*Script 9.5* és equivalent a l'expressió **pica ves: longitudCami2** de l'*Script 9.6*. De fet, el valor de la variable **longitudCami2** és el valor de la variable **longitudCami** més 170.

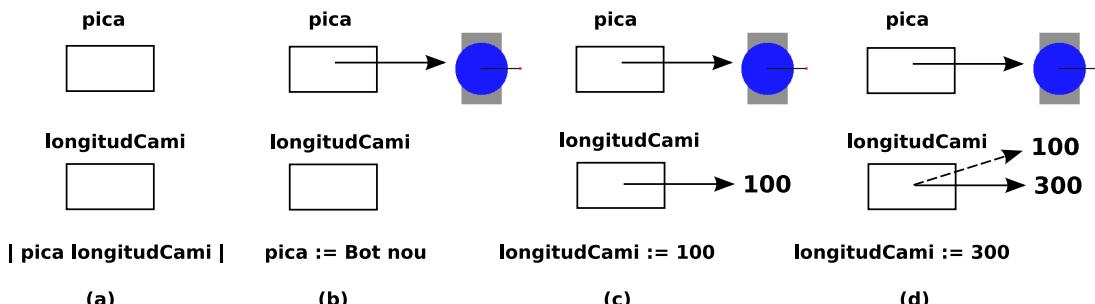
**Script 9.6 Utilitzar una nova variable per contenir el valor de la longitud del camí de pica.**



```
| pica longitudCami longitudCami2 |
pica := Bot nou.
longitudCami := 100.
longitudCami2 := longitudCami + 170.
pica ves: longitudCami2
```

El valor d'una variable pot canviar-se utilitzant `:=`. A l'*Script 9.7* primer declarem la variable `longitudCami` i després li assignem el valor 100. Immediatament després tornem a assignar-li un valor, en aquest cas 300 (la fletxa de guionets a la figura apuntant a 100 indica que la variable ja no apunta a 100). Quan la variable s'utilitza a la darrera expressió de l'*script*, el seu valor és 300. Com a resultat, el robot es mou endavant 300 píxels.

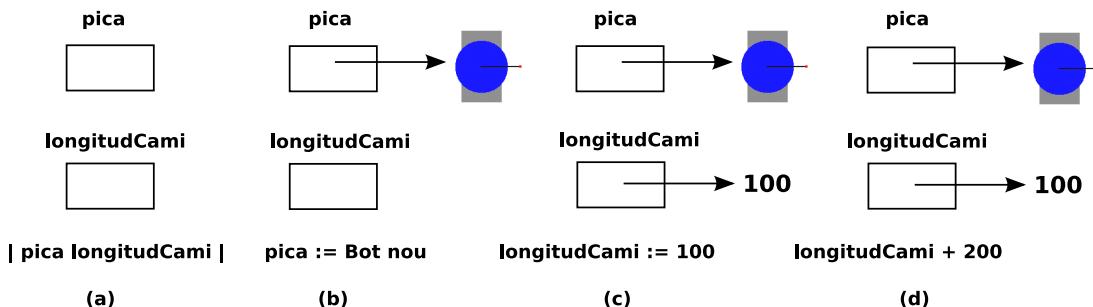
**Script 9.7 Canviar longitudCami dues vegades.**



```
| pica longitudCami |
pica := Bot nou.
longitudCami := 100.
longitudCami := 300.
pica ves: longitudCami
```

L'*Script* 9.8 demostra que utilitzar el valor d'una variable de qualsevol manera que no sigui assignant-hi un valor, no té cap efecte sobre el valor de la variable. L'única manera de canviar el valor d'una variable és amb una assignació. A l'*Script* 9.8, la variable `longitudCami` s'inicialitza amb el valor 100. Aleshores, el valor de `longitudCami`, aquí 100, s'afegeix a 200, però no hi ha cap assignació, de manera que el valor de la variable no es veu modificat. Per tant, quan el valor de `longitudCami`, que continua sent 100, s'utilitza en la darrera instrucció per especificar quants píxels s'hauria de moure el robot, el robot es mou endavant 100 píxels.

**Script 9.8 Utilitzar una variable sense assignar-hi un valor no té cap efecte sobre el valor de la variable.**



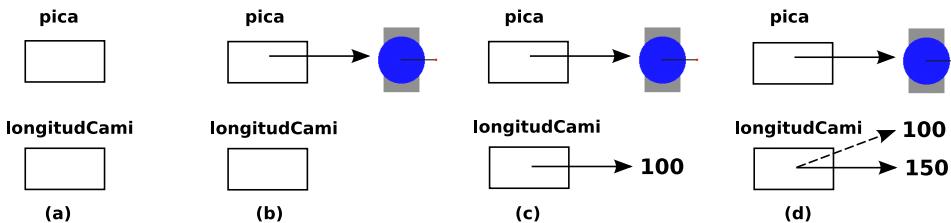
```
| pica longitudCami |
pica := Bot nou.
longitudCami := 100.
longitudCami + 200.
pica ves: longitudCami
```

A l'*Script* 9.9, la variable `longitudCami` s'inicialitza a 100. Després el seu valor és canviat pel valor de l'expressió `longitudCami + 50`. En aquest punt, el valor de `longitudCami` és 100, de manera que l'expressió `longitudCami + 50` retorna 150, i per tant el valor 150 és assignat a la variable `longitudCami`. Així doncs, el robot es mou endavant 150 píxels al darrer pas.

És important que us fixeu que a l'expressió `longitudCami := longitudCami + 150`, el nom de la variable `longitudCami` s'utilitza de *dues maneres diferents*: primer, s'avalua l'expressió a la dreta de `:=`, en la que `longitudCami` representa un valor, i el resultat d'aquesta avaluació s'assigna a la variable `longitudCami`, a l'esquerra de `:=`, on la variable es comporta com un contenidor on dipositar-hi un valor.

**Script 9.9** *El valor de la variable `longitudCami` s'utilitza per definir la mateixa variable.*

| pica longitudCami |      pica := Bot nou      longitudCami := 100      longitudCami := longitudCami + 150



```
| pica longitudCami |
pica := Bot nou.
longitudCami := 100.
longitudCami := longitudCami + 50.
pica ves: longitudCami
```

A l'*Script 9.10*, inicialitzem la variable `longitudCami` a 150. Després el valor de la variable `longitudCami` es reassigna per referir-se al valor de l'expressió `longitudCami + longitudCami`. En calcular el valor de l'expressió `longitudCami + longitudCami`, el valor de `longitudCami` és 150. Per tant l'expressió retorna 300, que passa a ser el nou valor de la variable `longitudCami`. Així, el robot es mou endavant 300 píxels.

**Script 9.10**

```
| pica longitudCami |
pica := Bot nou.
longitudCami := 150.
longitudCami := longitudCami + longitudCami.
pica ves: longitudCami
```

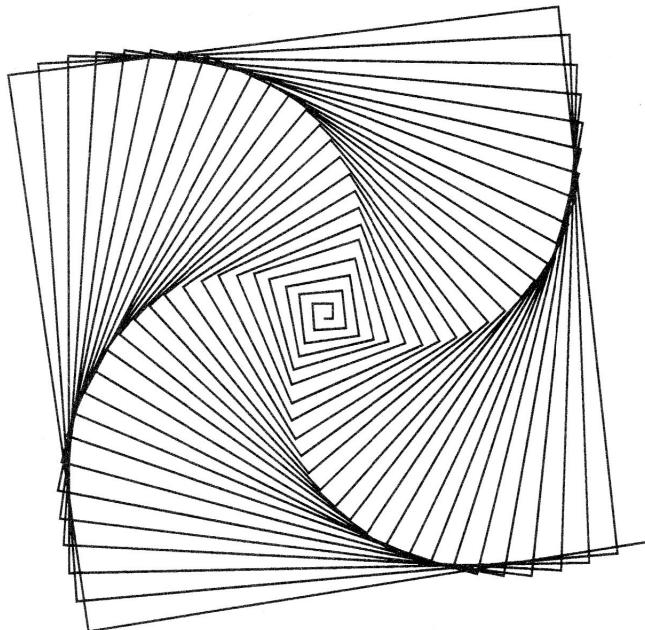
## Resum

- Una variable és un receptacle que serveix com a contenidor d'un valor. Podeu pensar en una variable com una capsà que es refereix a un objecte.
- Utilitzar una variable retorna el seu valor excepte quan la variable es troba a l'esquerra d'una assignació `:=`. En aquest cas, el valor de la variable canvia i aquesta queda associada al valor de l'expressió a la dreta de l'assignació `:=`. Per exemple, `longitudCami + 100` retorna 100 afegit al valor de la variable `longitudCami`. Per altra part, `longitudCami := 100` canvia el valor de `longitudCami` i aquest passa a ser 100.



## Capítol 10

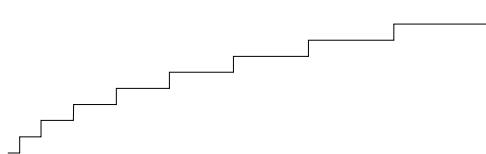
# Repeticions i variables



En aquest capítol us ensenyarem com fer servir les variables i els bucles conjuntament. Començarem analitzant un problema senzill que il·lustra la necessitat d'utilitzar variables als bucles. Després, experimentareu amb altres problemes.

## Una escala estranya

Intenteu programar un robot per dibuixar l'estranya escala de la figura 10.1. Tots els esglaons tenen la mateixa alçada, però la longitud de l'esglao va creixent i creixent a mida que anem pujant l'escala. Una manera de començar podria ser escriure un *script* per a una escala normal i modificar-lo. Hauríeu, però, de resoldre com fer que la longitud de l'esglao fos més gran que la de l'esglao anterior.



**Figura 10.1 — Pica dibuixa una escala molt estranya.**

Una solució senzilla la mostrem a l'*script* 10.1, on la longitud de cada esglao creix 10 píxels. Aquesta solució, però, no és satisfactòria per dues raons: Primera, heu de calcular la longitud de cada esglao manualment. I segona, heu d'escriure molts cops una seqüència de missatges que varia molt poc a cada repetició.

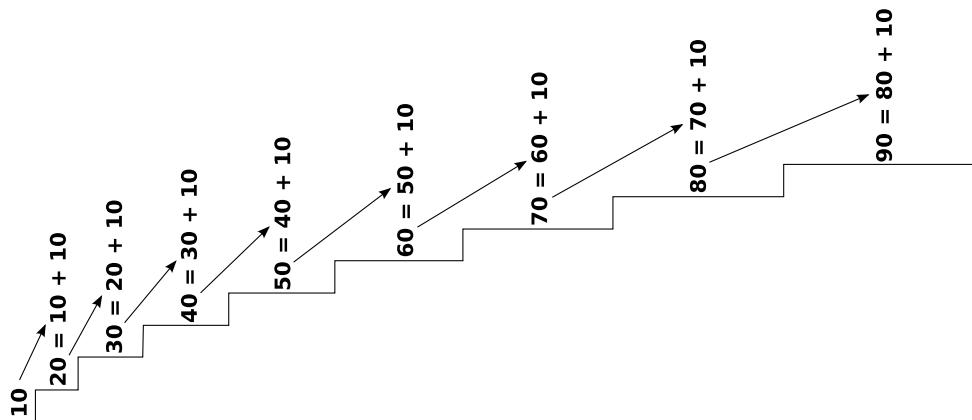
**Script 10.1** *Pica dibuixa l'escala estranya.*

```
| pica |
pica := Bot nou.
pica ves: 10.
pica giraEsquerra: 90.
pica ves: 5.
pica giraDreta: 90.
pica ves: 20.
pica giraEsquerra: 90.
pica ves: 5.
pica giraDreta: 90.
pica ves: 30.
pica giraEsquerra: 90.
pica ves: 5.
pica giraDreta: 90.
pica ves: 40.
pica giraEsquerra: 90.
pica ves: 5.
```

pica giraDreta: 90.

...

Ens agradaria ser capaços d'utilitzar totes les possibilitats que ens donen les variables (per automatitzar l'increment de la mida de l'esglao) combinades amb la potència dels bucles (de manera que no calgui repetir tant de codi). Per evitar repetir la seqüència d'enviaments de missatges podeu utilitzar el missatge vegadesRepetir:. I la clau de la utilització de les variables la podeu trobar en un examen atent de l'script 10.1, on podeu veure que la longitud de cada esglao posterior al primer és la longitud de l'esglao anterior més 10 píxels. És clar que  $20 = 10 + 10$ ,  $30 = 20 + 10$ ,  $40 = 30 + 10$  i així successivament, com mostrem a la figura 10.2.



**Figura 10.2 — La mida d'un esglao és la mida de l'esglao anterior més 10 píxels.**

Utilitzem la variable longitudEsglao per representar la longitud d'un esglao. Un cop longitudEsglao ha estat inicialitzada a la longitud del primer esglao, l'expressió longitudEsglao := longitudEsglao + 10 fixarà la longitud del *proper* esglao al valor del esglao *actual* incrementat en 10. El resultat és que si longitudEsglao s'inicialitza a 10 i dibuixem el primer esglao, aquest esglao tindrà obviament mida 10. Després que l'expressió longitudEsglao := longitudEsglao + 10 sigui executada, la propera vegada que es dibuixi un esglao tindrà mida 20. I després que l'expressió s'executi altre cop, el proper esglao tindrà mida 30, i així successivament.

Combinem-ho tot! Començarem amb l'script d'una escala normal (Script 10.2). Aleshores, a l'Script 10.3, obtenim la mateixa escala però utilitzant la variable longitudEsglao. A l'Script 10.4 afegim la línia longitudEsglao := longitudEsglao + 10 per canviar el valor de longitudEsglao a cada iteració, finalment aconseguim l'escala que volem.

**Script 10.2** Una escala amb esglaons normals.

```
| pica |
pica := Bot nou.
10 vegadesRepetir:
    [ pica ves: 10.
    pica giraEsquerra: 90.
    pica ves: 5.
    pica giraDreta: 90 ]
```

**Script 10.3** Una escala amb esglaons normals, utilitzant la variable longitudEsglao.

```
| pica longitudEsglao |
pica := Bot nou.
longitudEsglao := 10.
10 vegadesRepetir:
    [ pica ves: longitudEsglao.
    pica giraEsquerra: 90.
    pica ves: 5.
    pica giraDreta: 90 ]
```

**Script 10.4** La solució: incrementar la variable longitudEsglao cada cop dins del bucle produeix l'escala estranya.

```
| pica longitudEsglao |
pica := Bot nou.
longitudEsglao := 10.
10 vegadesRepetir:
    [ pica ves: longitudEsglao.
    pica giraEsquerra: 90.
    pica ves: 5.
    pica giraDreta: 90.
    longitudEsglao := longitudEsglao + 10. ]
```

Anem a examinar més detingudament la seqüència d'expressions dins del bucle de l'*Script 10.4*. La primera expressió dibuixa un esglaó fent que el robot es mogui endavant una distància que

ve donada pel valor de la variable longitudEsglao (que ha estat inicialitzada a 10 abans d'entrar dins del bucle). Aleshores el robot gira, dibuixa l'alçada de l'esglao i gira un altre cop. Llavors el valor de la variable longitudEsglao s'incrementa en 10 i la repetició torna a començar, tot i que ara la variable longitudEsglao té un valor nou, més gran que el que tenia abans. (20 a la segona repetició). Tot aquest procés s'executa 10 vegades.

L'expressió `longitudEsglao := longitudEsglao + 10` és absolutament necessària. Sense ella, el valor de la variable mai no canviaria.

### Experiment 10-1 (col·locació de l'increment dins del bucle)

Intenteu canviar la darrera línia del bucle; per exemple, `longitudEsglao := longitudEsglao + 15`. Aleshores proveu de moure aquesta línia a diferents llocs del bucle. Podeu explicar què passa quan moveu la darrera línia del bucle al començament del bucle?

---

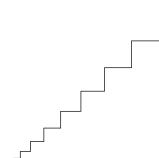
Si encara no esteu del tot segurs de què està passant a l'*Script* 10.4, us suggerim que penseu amb cura sobre el valor de la variable `longitudEsglao`, especialment al començament i al final del bucle. Intenteu endevinar el valor de `longitudEsglao` a cada una de les expressions del bucle durant tres iteracions. Si cal, llegiu el capítol 8 altre cop.

## Pràctiques amb repeticions i variables: laberints, espirals i altres.

Anem a veure com, combinant variables i repeticions, podem ajudar-vos a resoldre altres problemes.

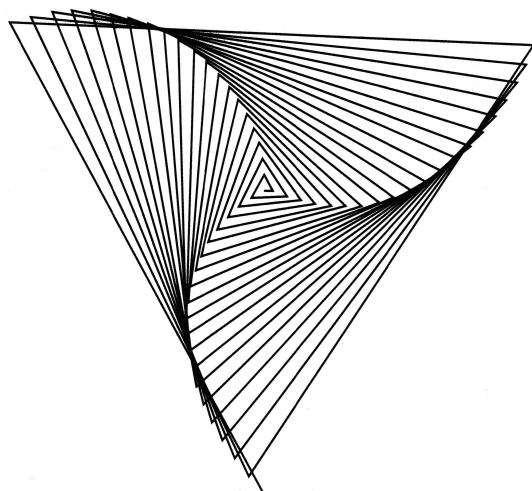
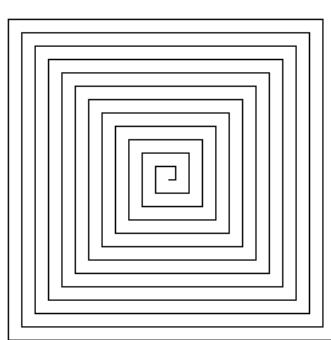
### Experiment 10-2 (una altra escala estranya)

Modifiqueu l'*Script* 10.4 per dibuixar la figura que veieu més avall, que representa una escala en què s'incrementen tant la longitud com l'alçada de l'esglao.



**Experiment 10-3 (un senzill laberint)**

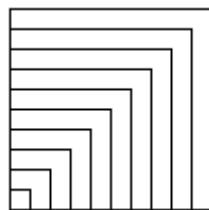
Escriviu un *script* que reproduueixi la figura mostrada més avall. A més, canviant l'angle que gira el robot, hauríeu de ser capaços de re-crear la figura que podeu veure al començament del capítol, així com l'espiral de la figura 10.3.



**Figura 10.3 — Una espiral bonica.**

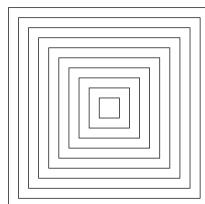
### Experiment 10-4 (quadrats russos)

Dibuixeu els quadrats encaixats de diferents mides com mostrem a la figura de més avall. Podríeu començar definint un bucle que dibuixi el mateix quadrat deu vegades. Aleshores introduïu una variable longitudCostat per representar la longitud del costat del quadrat, i finalment, feu que el costat s'incrementi a cada iteració del bucle. Com a repte addicional, podríeu provar d'escriure un nou *script* que dibuixi la mateixa figura sense que el robot salti o dibuixi cap línia dos cops.



### Experiment 10-5 (un passadís llarg)

Els quadrats concèntrics (que tenen el mateix centre) de diferents mides mostrats a la figura de més avall representen un llarg passadís, que sembla fer-se més petit a mida que l'anem veient de més lluny. Altre cop, comenceu dibuixant un quadrat, però aquest cop dibuixeu-lo començant des del mig (us caldrà un missatge salta:), de manera que quan canvieu la mida del quadrat, el proper quadrat serà dibuixat automàticament concèntric al primer. Ara definiu el vostre quadrat dins d'un bucle, i introduïu la variable longitudCostat representant la longitud del costat del quadrat. Finalment, incrementeu la longitud del costat del quadrat a cada iteració.



## Alguns aspectes importants d'utilitzar variables i repeticions

Ara que ja heu vist el procés complet de combinar repeticions i variables i heu experimentat una miqueta, ens agradaria emfatitzar alguns aspectes importants. L'*Script* 10.5 mostra la típica situació de forma esquemàtica: Primer declarem una variable. Després aquesta variable és inicialitzada. Dins del bucle, la variable és utilitzada per fer alguns càculs, i el seu valor és canviat per a la propera iteració.

**Script 10.5** *Un esquema d'script mostrant l'ús d'una variable dins d'un bucle.*

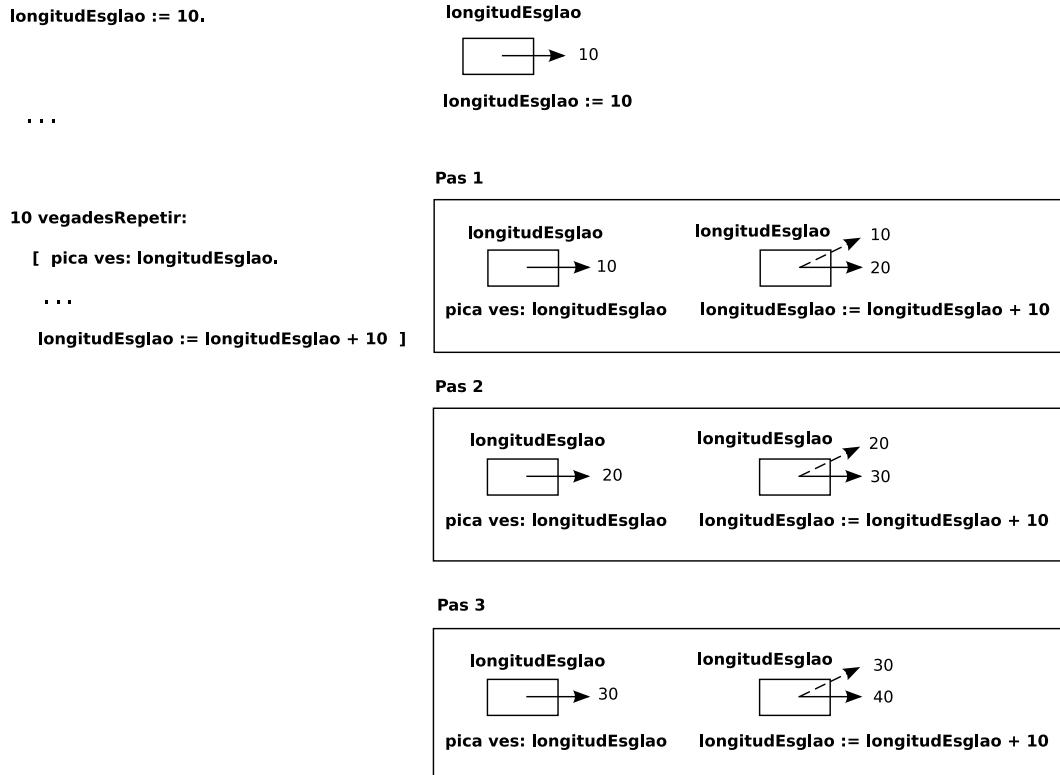
```
| longitudEsglao pica |           "declaració de variables"
...
longitudEsglao := 10.            "inicialització de la variable"
...
10 vegadesRepetir:
    [ pica ves: longitudEsglao.      "ús de la variable"
    ...
    longitudEsglao := longitudEsglao + 10 ]   "canvi de valor de la variable"
```

### Inicialització de variables

Quan introduïu una variable en un bucle, heu de posar especial atenció al valor inicial de la variable, és a dir, al valor assignat a la variable quan és inicialitzada. Recordeu que una variable no pot ser utilitzada fins que ha estat inicialitzada. Usualment, la inicialització de les variables es fa fora del bucle, ja que d'una altra manera el valor de la variable seria reinicialitzat a cada iteració, i aquest valor no canviaria mai.

### Utilitzar i canviar el valor d'una variable

Dins del bucle, el valor de la variable s'utilitza sovint per realitzar càlculs diversos, com calcular els valors d'altres variables o dir-li a un robot a quina distància s'ha de moure. Per tant, el valor d'una variable eventualment es canvia. A l'exemple de l'escala, l'expressió `longitudEsglao := longitudEsglao + 10` incrementa el valor de la longitud de l'esglao basant-se en el seu valor anterior. El que és important d'entendre és que el nou valor assignat a la variable serà el seu valor a la propera volta del bucle, com veieu a la figura 10.4.

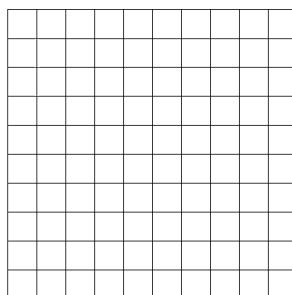


**Figura 10.4** — La longitud d'un esglao és la longitud de l'esglao anterior més 10 píxels. El darrer valor que una variable té dins del bucle és el valor que la variable tindrà al començament del bucle quan es repeteixi.

## Experiments avançats

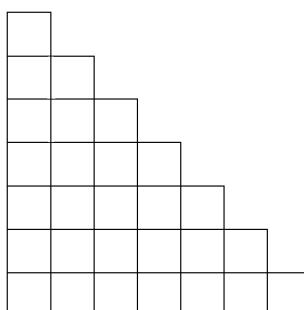
### Experiment 10-6 (quadrats)

Definiu un *script* que dibuixi el tauler que veieu més avall. Aquest experiment és una mica més complicat que els experiments anteriors, ja que és difícil veure com dibuixar la figura amb una sola repetició. Tot i així, hi ha diverses maneres de resoldre el problema utilitzant dos bucles. Per exemple, podríeu utilitzar un bucle per dibuixar totes les línies horizontals i un altre per dibuixar totes les línies verticals.



### Experiment 10-7 (piràmide)

Escriviu un *script* que dibuixi la figura que veieu més avall, representant els blocs de pedra que formen part de la piràmide esglaonada de Saqqara, que ja coneixeu de capítols anteriors. Podríeu modificar el vostre *script* de l'Experiment 10-7 canviant la variable corresponent a la longitud de la línia dibuixada a cada iteració.



## Resum

- Quan s'introdueix una variable dins d'un bucle, heu d'estar segurs d'haver-la inicialitzat correctament. Recordeu que cal inicialitzar una variable abans d'utilitzar-ne el valor. Usualment, la inicialització es fa fora del bucle, ja que si no fos així el valor de la variable no canviaria amb les repeticions del bucle.
- Recordeu que el darrer valor que rep una variable dins d'un bucle serà el valor de la variable al començament de la següent iteració del bucle.



## Capítol 11

# Compondre missatges

Smalltalk, com tots els altres llenguatges de programació, segueix algunes regles en executar els missatges enviats als objectes. Encara no us hem explicat aquestes regles, i és possible que us hagieu preguntat quines són quan estavez experimentant amb els *scripts* dels capítols anteriors. La vostra paciència es veurà ara recompensada. Aquest capítol explica com llegir i escriure missatges formulats correctament.

Aquest capítol pot semblar una mica més difícil o abstracte que els capítols anteriors. Tot i així, hem procurat presentar tan clarament com hem sabut les senzilles regles que governen l'escriptura de missatges. Entendre aquests detalls no és tan divertit com jugar amb els robots, però és el preu que heu de pagar si voleu ser capaços d'escriure programes més complicats. Les bones notícies són que Smalltalk no és un llenguatge complex: Hi ha només cinc regles que heu d'entendre. Si encara dubteu, podeu saltar-vos aquest capítol en una primera lectura i tornar-hi quan tingueu algun dubte sobre l'estructura dels vostres programes.

Tal com vam explicar al capítol 2, un missatge està compost d'un *selector de missatge* i els *arguments del missatge*, si n'hi ha. Un missatge és enviat a un *receptor del missatge*. La combinació del missatge i el receptor s'anomena un *enviament de missatge*.

Quan escriviu una *expressió* complexa, com pica ves: `100 + 20`, aquesta conté dos enviaments de missatge, amb els selectors ves: `i +`, i heu de conèixer en quin ordre són executats els missatges si voleu entendre el resultat de l'*expressió*. A Smalltalk, l'ordre en què els missatges s'executen ve determinat pel *tipus* de l'*enviament de missatge*. N'hi ha tres tipus: *unaris*, *binaris* i *paraula-clau*. Els missatges *unaris* sempre s'envien els primers, seguits dels missatges *binaris* i finalment els missatges de *paraula-clau*. Els missatges posats entre parèntesis s'executen abans que qualsevol altre tipus de missatge. Això vol dir que podeu canviar l'ordre en què els enviaments de missatges són executats mitjançant la col·locació de parèntesis. Aquestes regles són en gran part responsables que el codi Smalltalk sigui fàcil de llegir. I aviat descobrireu que la majoria de les

vegades ni tan sols us caldrà pensar-hi. Tot i així, heu de conèixer aquestes regles, ja que pot ser que en alguna ocasió sí que us calgui saber-les aplicar.

Tots els exemples que apareixen en aquest capítol són codi executable, tal com es mostra en el text. No dubteu a provar-los i entendre com funcionen. Començarem ensenyant-vos la manera d'identificar els diferents tipus de missatges, i després presentarem alguns exemples de cada tipus. Finalment, introduirem les regles per a la composició de missatges.

## Els tres tipus de missatges

Smalltalk defineix un petit nombre de regles senzilles per determinar l'ordre en què s'executen els missatges. Aquestes regles, que presentarem en detall més tard, es basen en la diferenciació dels missatges en tres tipus:

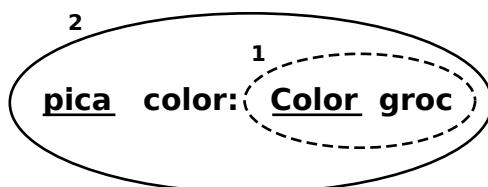
- *Missatges unaris* són missatges sense arguments. Són enviats a un objecte (el receptor del missatge) sense cap altra informació. Per exemple, a l'expressió `pica color`, el missatge `color` és un missatge unari. No envia cap informació addicional; això vol dir que no hi ha cap argument. Aquests missatges s'anomenen “unaris” del Llatí *unum*, que significa “un”, ja que un enviament de missatge amb un missatge unari involucra *només un objecte*, el receptor del missatge.
- *Missatges binaris*, són missatges que involucren *dos objectes*: el receptor del missatge i l'únic argument del selector del missatge (la paraula “binari” ve del Llatí *bis*, que significa “dues vegades”). Els missatges binaris usualment s'utilitzen en expressions matemàtiques. Per exemple, a l'enviament de missatge `10 + 20`, el missatge consta del selector de missatge `+` juntament amb el seu únic argument `20`. El missatge `+ 20` és enviat a l'objecte `10`, que és el receptor del missatge.
- *Missatges de paraula-clau*, són missatges el selector dels quals conté almenys una paraula clau amb un caràcter “dos punts” (`:`) al seu nom i que té un o més arguments. Per exemple, al missatge `pica ves: 100`, la paraula clau `ves:` conté el caràcter “dos punts” (`:`), hi ha un argument, `100`, i per tant hi ha dos objectes involucrats en l'enviament de missatge: el receptor del missatge `pica` i l'argument `100`.

**Nota** No us confongueu per la nomenclatura dels missatges unaris i binaris. La idea d’ “un” a la paraula *unari* i la idea de “dos” a la paraula *binari* es refereixen al nombre d’objectes involucrats en l’enviament del missatge, no al nombre d’arguments. Així, un missatge unari no té cap argument, però hi participa un objecte, el receptor del missatge. Un missatge binari té només un argument, i per tant un enviament de missatge d’un missatge binari implica dos objectes: l’argument del selector del missatge i el receptor del missatge.

## Identificar missatges

Per poder entendre l’estructura d’una expressió, el primer que us cal fer és identificar els missatges i els seus receptors. Per tal de fer això, us suggerim que utilitzeu una notació gràfica com la que veieu a la figura 11.1. En totes les figures d’aquest capítol, els receptors de missatge estan *subratllats*, cada missatge està envoltat d’una *el·lipse*, i els missatges que constitueixen l’expressió estan *numerats* en l’ordre en què s’executaràn. Quan hi ha més d’una el·lipse, la primera el·lipse a executar es dibuixa amb una línia interrompuda de manera que es pugui veure de seguida on començar.

La figura 11.1 mostra que l’expressió `pica color: Color groc` conté l’expressió `Color groc`. Per tant hi ha dues el·lipses, una per a l’expressió completa `pica color: Color groc` i una altra per a la subexpressió `Color groc`. Aprendreu una mica més endavant que l’expressió `Color groc` s’executa primer, de manera que l’el·lipse està dibuixada amb línia interrompuda i s’ha numerat amb el nombre 1. Fixeu-vos que cada missatge té un receptor. El robot `pica` rep el missatge `color: ...`, i `Color` rep el missatge `groc`. Per tant, aquests dos receptors de missatge estan subratllats (els tres punts al missatge `color: ...` indiquen l’argument del selector de missatge `color:`, que serà el resultat de l’enviament de missatge `Color groc`).



**Figura 11.1** — L’expressió `pica color: Color groc` conté la subexpressió `Color groc`. El missatge `groc` és enviat a `Color`, i després el missatge `color: ...` és enviat a `pica`.

Tal com hem dit, cada missatge és enviat a un objecte anomenat el receptor del missatge.

Un receptor no ha de ser un robot. Pot ser qualsevol cosa, des d'un nombre a una finestra. Un receptor de missatge pot aparèixer explícitament com a primer element d'una expressió, com pica a l'expressió pica ves: 100 o Color a l'expressió Color blau. Un receptor, però, també pot ser el resultat d'altres enviaments de missatges. Per exemple, al missatge Bot nou ves: 100, el receptor del missatge ves: 100 és l'objecte resultant del retorn de l'enviament de missatge Bot nou. Tot i així, un missatge sempre és enviat a algun objecte, i aquest objecte és anomenat receptor del missatge.

---

**Important!** En un *enviament de missatge*, un *missatge* és sempre enviat a un *objecte*, anomenat el *receptor del missatge*, que pot ser explícit o el resultat d'altres enviaments de missatge.

---

La taula 11.1 mostra alguns enviaments de missatge; us suggerim que en cada cas identifiqueu el tipus de missatge i dibuixe la representació gràfica de l'expressió.

**Taula 11.1 — Alguns exemples d'enviaments de missatge, simples i compostos.**

Expressió	Tipus	Acció
pica ves: 100	paraula-clau	El robot receptor es mou endavant 100 píxels.
100 + 20	binari	El nombre 100 rep el missatge + amb el nombre 20 com a argument.
pica est	unari	El robot receptor pica apunta a l'est
pica color: Color groc	paraula-clau, unari	El robot receptor pica canvia els seu color pel color groc
pica ves: 100 + 20	paraula-clau, binari	El robot receptor es mou endavant 120 píxels.
Bot nou ves: 100	unari, paraula-clau	El missatge nou és enviat a la classe Bot, que retorna un robot nou al qual el missatge ves: 100 és enviat, fent que el robot es mogui endavant 100 píxels.

A partir de la taula 11.1 hauríeu de parar atenció a les següents observacions:

- Alguns missatges tenen arguments, altres missatges no. El missatge est, sent unari, no té cap argument, mentre que ves: 100 i + 20 tenen cadascun un argument, el nombre 100 i el nombre 20 respectivament.

- S'envien diferents missatges a diversos objectes. A l'expressió pica est, el missatge est és enviat a un robot, i a 100 + 20, el missatge + 20 és enviat al nombre 100.
- Hi ha missatges simples i missatges compostos. Per exemple, Color groc i 100 + 20 són simples: Un missatge és enviat a un objecte. Per altra part, l'expressió pica ves: 100 + 20 conté dos missatges: + 20 és enviat a 100, i després ves: ... és enviat a pica, on ... representa el resultat de l'execució de l'enviament de missatge 100 + 20.
- El receptor de missatge pot ser el resultat d'una expressió que retorna un objecte. A l'expressió Bot nou ves: 100, el missatge ves: 100 és enviat a l'objecte (un robot) que resulta de l'avaluació de l'expressió Bot nou.

## Els tres tipus de missatges en detall

Ara que ja sou capaços d'identificar els receptors de missatge i els tres tipus de missatges, examinem en detall els diferents tipus de missatges.

### Missatges unaris

Entre els usos habituals dels missatges unaris hi ha l'obtenció d'un valor a partir d'un objecte, com la mida del llapis d'un robot (pica midaLlapis); obtenir un objecte a partir d'una classe (Color groc); i ordenar al receptor la realització d'alguna acció (pica tornarInvisible). Recordeu que un missatge unari no té arguments: és enviat al receptor sense cap més informació. Així, l'únic objecte implicat en un enviament de missatge unari és el receptor del missatge. Enviaments de missatge unaris són de la forma *receptor nomMissatge*. L'Script 11.1 presenta alguns exemples de missatges unaris, mostrats en negreta.

**Script 11.1** Exemples de missatges unaris.

```
| pica |
pica := Bot nou.
pica color.
pica midaLlapis.
pica est.
Color groc.
125 factorial.
```

---

**Important!** Els missatges unaris són missatges que no tenen cap argument. Un enviament de missatge unari té la forma *receptor nomMissatge*.

---

## Missatges binaris

Els missatges binaris són missatges que involucren dos objectes: el receptor i un únic argument. Tots els selectors de missatges dels missatges binaris consisteixen en un o dos caràcters de la llista següent: +, \*, /, |, &, =, >, <, ~, @. Per tant, +, = i \* són selectors de missatge, però també ho és =>, que està compost de dos símbols.

La taula 11.2 mostra alguns exemples d'enviaments de missatge binari i el seu significat. Ara mateix, no entrarem en els detalls d'aquests exemples, no us amoïneu si no esteu completament segurs del que fan tots i cadascun d'aquests selectors de missatge. Proveu, però, d'executar les expressions i altres de similars.

**Taula 11.2 — Exemples de missatges binaris numèrics.**

Expressió	Valor Retornat	Acció
1 + 2.5	3.5	Suma de dos nombres
3.4 * 5	17.0	Producte de dos nombres
8 / 2	4	Divisió de dos nombres
10 - 8.3	1.7	Resta de dos nombres
12 == 11	false	Test d'igualtat entre nombres
12 ~= 11	true	Test de desigualtat entre nombres
12 > 9	true	És el receptor més gran que l'argument?
12 >= 10	true	És el receptor més gran o igual que l'argument?
12 < 10	false	És el receptor més petit que l'argument?
100@10	100@10	Crea un punt amb coordenades (100,10)

---

**Important!** Els enviaments de missatge binari involucren dos objectes: el receptor i un únic argument. El selector de missatge d'un missatge binari està compost d'un o dos caràcters de la llista següent: +, \*, /, |, &, =, >, <, ~, @. Els enviaments de missatge binari tenen la forma *receptor nomMissatge argument*.

---

## Missatges de paraula-clau

Els missatges de paraula-clau són missatges que prenen com a mínim un argument i contenen com a mínim un caràcter “dos punts” (:). Fixeu-vos que els dos punts formen part del selector de missatge. Per tant, `ves:`, no `ves`, és el nom d'un missatge de paraula-clau. L'*Script 11.2* presenta alguns exemples de missatges de paraula-clau, mostrats en negreta.

**Script 11.2** *Exemples de missatges de paraula-clau.*

```
| pica |
pica := Bot nou.
pica ves: 100.
pica midaLlapis: 5.
pica color: Color groc.
pica gira: 90.
```

Ja hem dit que els missatges de paraula-clau tenen com a mínim un argument, però encara no hem vist cap exemple d'un missatge amb múltiples arguments. L'enviament de missatge<sup>1</sup> *unNumero between: fitaInferior and: fitaSuperior* mira si el nombre *unNumero* és a l'interval representat pels dos nombres *fitaInferior* i *fitaSuperior*. El missatge necessita dos arguments, els dos extrems de l'interval. En mostrem un exemple a la taula 11.3. Fixeu-vos que el selector de missatge és *between:and:*. Està compost de les dues paraules *between:* i *and:*.

**Taula 11.3 — Missatges de paraula-clau que necessiten més d'un argument.**

Expressió	Arguments	Valor Retornat	Acció
5 <b>between:</b> 2 <b>and:</b> 10	2, 10	true	Està 5 entre 2 i 10?
Color r: 0 g: 1 b: 0	0, 1, 0	un objecte-color verd	Crea un color amb els valors donats de vermell, verd i blau.

---

**Important!** Els missatges de paraula-clau necessiten com a mínim un argument, i el seu selector de missatge conté com a mínim un caràcter “dos punts” (:). Un enviament de missatge de paraula-clau amb dos arguments pren la forma *receptor primeraParaulaNomMissatge: primerArgument segonaParaulaNomMissatge: segonArgument*.

---

<sup>1</sup>*Nota del Traductor:* Altre cop, aquest missatge pertany a Smalltalk més que a l'entorn BotInc, per tant no l'hem traduït.

## Ordre d'execució

Ja sabeu que hi ha tres tipus de missatges: unaris, binaris i de paraula-clau. Ara us explicarem, com vam prometre, la manera de determinar l'ordre en què són executats els missatges. L'ordre d'execució dels missatges ve donat pel tipus de missatge, tal com està descrit en les tres regles següents:

**Regla 1:** Els enviaments de missatge unari s'executen primer, després els enviaments de missatge binari i finalment els enviaments de missatge de paraula-clau.

**Regla 2:** Tal com passa amb les expressions matemàtiques, la prioritat d'execució dels missatges pot ser invalidada mitjançant la col·locació de parèntesis: els enviaments de missatge entre parèntesis s'executen abans que qualsevol altre tipus d'enviament de missatge.

**Regla 3:** Els enviaments de missatge del mateix tipus s'executen d'esquerra a dreta.

Aquestes regles poden semblar complicades, però són força naturals, i un cop us hi acostumeu, pràcticament no hi haureu de tornar a pensar. En particular, la tercera regla senzillament diu que els missatges del mateix tipus s'executen en l'ordre en el qual són llegits.

Si mai teniu dubtes, i voleu assegurar-vos que els vostres missatges s'executen en l'ordre que cal, sempre podeu afegir parèntesis extra, com veieu a la figura 11.2. A la figura, s'analitza l'expressió `pica color: Color groc`. El selector de missatge `Color groc` és un missatge unari, mentre que el selector de missatge `color:` és un missatge de paraula-clau. Per tant, l'expressió `Color groc` s'executa primer. Si no esteu convençuts de l'ordre d'execució, podeu posar parèntesis al voltant de `Color groc` per assegurar-vos que aquesta expressió serà executada primer. És a dir, `pica color: Color groc i pica color: (Color groc)` tenen exactament el mateix efecte. La resta d'aquesta secció il·lustra cada un d'aquests aspectes.



**Figura 11.2 —** Els missatges unaris s'executen primer, de manera que `Color groc` és la primera expressió a executar-se. Aquesta execució retorna un objecte `color`, que es passa com a argument del missatge `color: ...`, que és enviat a `pica`.

## Regla 1: Unari > Binari > Paraula-Clau

Els enviaments de missatge unari s'executen primer, després els enviaments de missatge binari i finalment els enviaments de missatge de paraula-clau. En l'argot de la programació diem que els missatges unaris tenen *precedència* sobre els missatges binaris, i que els missatges binaris tenen precedència sobre els missatges de paraula-clau.

---

**Important!** Regla 1: Els enviaments de missatge unari s'executen abans que els enviaments de missatge binari, que s'executen abans que els enviaments de missatge de paraula-clau.

---

### Exemple 1

A l'enviament de missatge pica color: Color groc, hi ha un missatge unari, groc, enviat a la classe Color, i hi ha un missatge de paraula-clau, color: ..., enviat al robot pica. Els enviaments de missatge unari s'executen primer, de manera que el primer enviament de missatge per executar és Color groc. Aquesta execució retorna un objecte color, representat com a *unColor* (ja que ens cal un nom), que es passa a pica com a argument del missatge color: *unColor*. La figura 11.2 mostra gràficament l'ordre en què són executats els missatges.

Per ajudar-vos a entendre-ho millor, voldríem proposar una representació textual de l'execució pas a pas dels enviaments de missatge compostos. A Pas-a-pas 11.1, l'enviament de missatge per ser executat pas a pas és pica color: Color groc. La primera línia mostra l'enviament de missatge complet en negreta.

#### Pas-a-pas 11.1 Descomposició de l'execució de pica color: Color groc

**pica color: Color groc**

- |     |                            |                |
|-----|----------------------------|----------------|
| (1) | Color groc                 | “unari”        |
|     | -retorna> <i>unColor</i>   |                |
| (2) | pica color: <i>unColor</i> | “paraula-clau” |

Les línies de codi representen els diferents passos de l'execució enumerats en l'ordre que succeeixen. Així, Color groc és la primera expressió a ser executada. Fixeu-vos que les expressions han estat sagnades per alinear-se amb les seves expressions equivalents a l'enviament de missatge que hi ha a dalt de tot.

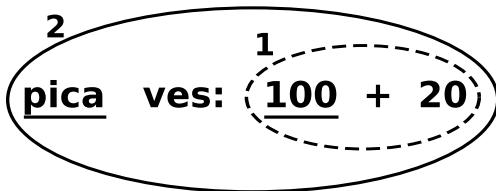
Quan l'execució d'un enviament de missatge retorna un resultat que s'utilitza en les següents execucions, la línia que segueix l'expressió executada mostra “-retorna>” seguit del resultat. Aquí l'expressió Color groc retorna un objecte color que hem anomenat *unColor* de manera que podem referir-nos-en més endavant. La segona expressió executada és pica color: *unColor*, on,

com acabem d'explicar, *unColor* és el resultat obtingut del pas d'execució previ. Per enfatitzar això, el valor retornat és escrit en itàlica. El tipus del missatge que s'està executant s'escriu com a comentari dins de cometes, per deixar-ho més clar. Per exemple, es diu explícitament que *Color* groc és un missatge unari.

### Exemple 2

L'enviament de missatge *pica ves*: 100 + 20, conté el selector de missatge binari + i el selector de missatge de paraula-clau *ves*:. Els missatges binaris s'executen abans que els missatges de paraula-clau, per tant 100 + 20 s'executa primer: El missatge + 20 és enviat a l'objecte 100, que retorna el nombre 120. Després el missatge *pica ves*: 120 s'executa amb 120 com a argument. Pas-a-pas 11.2 mostra com s'executa l'expressió.

**Pas-a-pas 11.2** Descomposició de l'expressió *pica ves*: 100 + 20



**pica ves: 100 + 20**

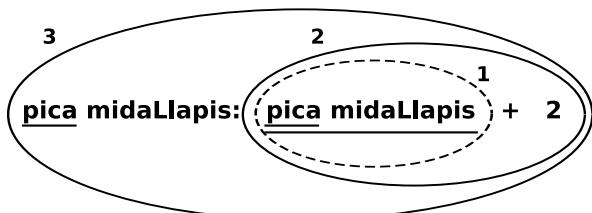
- (1)       $100 + 20$  "binari"  
-retorna > 120
- (2)    *pica ves: 120*        "paraula-clau"

### Exemple 3

El missatge *pica midaLlapis*: *pica midaLlapis* + 2 conté el missatge unari *midaLlapis*, el missatge binari amb selector +, i el missatge de paraula-clau amb selector *midaLlapis*:. Pas-a-pas 11.3 il·lustra la descomposició de l'execució del missatge. L'enviament de missatge unari *pica midaLlapis* s'executa primer (pas 1). Aquest missatge retorna un nombre, que anomenarem *unNumero*, representant la mida de llapis actual del receptor. Després s'executa l'enviament de missatge binari *unNumero* + 2 (pas 2). El nombre *unNumero* és el receptor del missatge + 2, que retorna un altre nombre, la suma, que anomenem *unAltNumero*. Finalment, el missatge de paraula-clau *midaLlapis*: *unAltNumero* és enviat a *pica*, que fa que *unAltNumero* sigui la seva mida de llapis actual.

Tot plegat, l'expressió sencera incrementa la mida del llapis del receptor en dos píxels. Ho fa demanant primer la mida del seu llapis a pica, afegint dos a aquest nombre (*unNumero + 2*), i després dient-li a pica que canviï la mida del seu llapis pel nou nombre (pica *midaLlapis*: *unAltNumero*). Fixeu-vos que *midaLlapis* i *midaLlapis*: són dos selectors de missatge diferents! El primer és unari i *demanar* al receptor la mida del seu llapis, i el segon és de paraula-clau i *ordena* al receptor que canviï la seva mida de llapis pel valor de l'argument.

**Pas-a-pas 11.3 Descomposició de l'expressió pica *midaLlapis*: pica *midaLlapis* + 2**

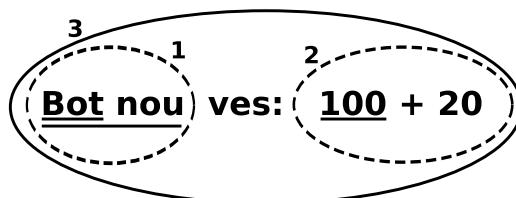


**pica *midaLlapis*: pica *midaLlapis* + 2**

- |     |   |                |
|-----|---|----------------|
| (1) | pica <i>midaLlapis</i>                      | "unari"        |
|     | -retorna> <i>unNumero</i>                   |                |
| (2) | <i>unNumero</i> + 2                         | "binari"       |
|     | -retorna> <i>unAltNumero</i>                |                |
| (3) | pica <i>midaLlapis</i> : <i>unAltNumero</i> | "paraula-clau" |

**Exemple 4**

Com a exercici, us deixarem descompondre l'execució del missatge Bot nou ves: 100 + 20, que està compost d'un missatge unari, un missatge de paraula-clau i un missatge binari (figura 11.3).

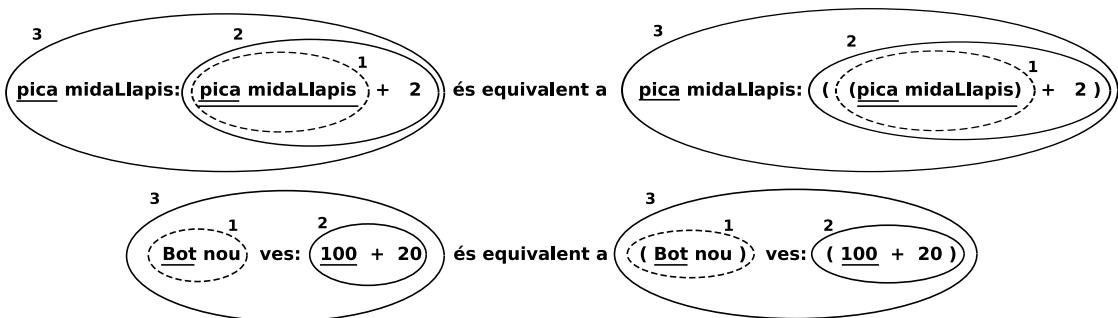


**Figura 11.3 — Descompondre Bot nou ves: 100 + 20**

## Regla 2: Primer els parèntesis

L'ordenació per defecte de l'execució de missatges pot no ser l'adequada pel que voleu fer amb una expressió, per tant haurieu de poder canviar aquesta ordenació. Per poder fer això, Smalltalk us dóna la possibilitat d'utilitzar parèntesis ( i ). Igual que en matemàtiques, les expressions entre parèntesis tenen la precedència màxima, i s'executen abans que qualsevol altra cosa.

Recordeu que si trobeu les regles per a l'ordre d'execució complicades o si senzillament voleu deixar més clara l'estructura d'una expressió, utilitzeu parèntesis per assegurar que l'execució dels missatges es fa en l'ordre que vosaltres voleu. La figura 11.4 mostra algunes de les expressions que hem vist anteriorment juntament amb les expressions equivalents utilitzant parèntesis.



**Figura 11.4 — Missatges equivalents utilitzant parèntesis.**

---

**Important!** Regla 2: Igual que en matemàtiques, els missatges entre parèntesis s'executen abans que qualsevol altre missatge. Tenen la *prioritat* més alta.

---

### Exemple 5

El missatge<sup>2</sup> (65@325 extent: 134@100) center retorna el centre d'un rectangle amb (65,325) com a cantonada superior esquerra, l'amplada del qual és 134 píxels i l'altura és 100 píxels. Pass-a-pas 11.4 ens mostra com es descompon i executa el missatge. Primer, el missatge dins dels parèntesis és executat. És una expressió composta consistent en tres enviaments de missatge:

<sup>2</sup>Nota del Traductor: Els punts de la forma  $a@b$  són propis de l'entorn general Squeak, per tant els seus missatges no han estat traduïts, per exemple extent: que retorna un rectangle. Amb els rectangles passa el mateix, per tant els missatges corresponents tampoc no han estat traduïts, per exemple, center.

dos enviaments de missatge binari, 65 @ 325 i 134 @ 100, que s'executen primer i retornen punts; i el missatge de paraula-clau extent: ..., que és enviat al punt (65,325) i que retorna un rectangle amb la cantonada superior esquerra, l'amplada i l'altura que ja hem dit. Finalment, el missatge unari center és enviat a aquest rectangle, i un punt, el centre del rectangle, és retornat. Provar d'avaluar el missatge sense parèntesis donaria un error, ja que en aquest cas, el missatge unari center s'hauria d'executar primer enviant-lo a l'objecte 100, que, com és un nombre, no entén el missatge center.

**Pas-a-pas 11.4 Descomposició d'una expressió composta amb parèntesis prioritaris.**

**(65 @ 325 extent: 134 @ 100) center**

- (1) 65 @ 325 "binari"  
-retorna> unPunt
- (2) 134 @ 100 "binari"  
-retorna> unAltrePunt
- (3) unPunt extent: unAltrePunt "paraula-clau"  
-retorna> unRectangle
- (4) unRectangle center "unari"  
-retorna> 132 @ 375

### Regla 3: D'esquerra a dreta

Ara que ja sabeu com categoritzar els missatges d'acord amb la prioritat d'execució, la darrera pregunta que queda pendent és en quin ordre s'executen els missatges amb la mateixa prioritat. La regla 3 ens diu que s'executen d'esquerra a dreta. Ja hem utilitzat aquesta regla, a Pas-a-pas 11.4, on el missatge binari @ 325 s'ha executat abans que el missatge @ 100, que està més a la dreta.

---

**Important!** Regla 3: Missatges del mateix tipus s'executen en ordre d'esquerra a dreta.

---

### Exemple 6

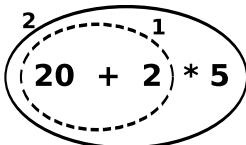
A l'expressió Bot nou est, els dos enviaments de missatge envien missatges unaris, de manera que el missatge que està més a l'esquerra, Bot nou, s'executa primer. Retorna un robot nou, anomenat *unBot* a Pas-a-pas 11.5, al que s'envia el segon missatge, est. Pas-a-pas 11.5 mostra l'ordre d'execució.

**Pas-a-pas 11.5 Descomposició de l'expressió Bot nou est****Bot nou est**

- |     |                 |         |
|-----|-----------------|---------|
| (1) | Bot nou         | "unari" |
|     | -retorna> unBot |         |
| (2) | unBot est       | "unari" |

**Exemple 7**

A l'expressió  $20 + 2 * 5$ , hi ha només dos selectors de missatge binari  $+ i *$ . Seguint la regla 3, com  $+$  és a l'esquerra de  $*$ ,  $+ 2$  hauria d'executar-se primer. La notació matemàtica habitual i molts altres llenguatges de programació ens han acostumat que la multiplicació tingui més precedència que la suma, sense importar l'ordre en què apareixen els operadors aritmètics. A Smalltalk, però, no hi ha cap *prioritat específica* per a operacions matemàtiques. Els descriptors de missatge  $+ i *$  són només missatges binaris, per tant tenen el mateix estatus. El selector de missatge  $*$  no té precedència sobre  $+$ , i el selector de missatge de més a l'esquerra  $+$  és enviat primer. Després,  $*$  és enviat al resultat, com mostrem a Pas-a-pas 11.6.

**Pas-a-pas 11.6 Descomposició de  $20 + 2 * 5$**  **$20 + 2 * 5$** 

- |     |               |          |
|-----|---------------|----------|
| (1) | $20 + 2$      | "binari" |
|     | -retorna> 22  |          |
| (2) | $22 * 5$      | "binari" |
|     | -retorna> 110 |          |

---

**Nota** No hi ha prioritat entre missatges binaris. A l'expressió  $20 + 2 * 5$ , el missatge situat més a l'esquerra,  $+$ , s'avalua primer, malgrat que en notació matemàtica habitual, el producte té més precedència que la suma.

---

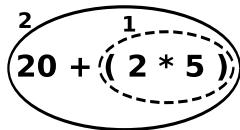
Podeu veure, com mostrem a Pas-a-pas 11.6, que el resultat d'aquesta expressió no és 30, que és el que obtindríeu si féssim primer el producte, sinó 110. Aquest comportament és d'entrada sorprenent, però deriva de les tres senzilles regles per executar missatges. Aquest ordre anti-intuïtiu de les operacions matemàtiques és el preu que heu de pagar per la simplicitat del model Smalltalk, que només té mètodes. Si voleu que la vostra expressió obereixi la prioritat habitual de les operacions matemàtiques, hauríeu d'utilitzar parèntesis, ja que quan els enviaments de missatge estan tancats entre parèntesis, s'executen primer. Per tant, l'expressió  $20 + (2 * 5)$  retorna el resultat 30, tal i com es veu a Pas-a-pas 11.7.

---

**Important!** Enviaments de missatge envoltats de parèntesis s'executen primer. Per tant, a l'expressió  $20 + (2 * 5)$ , el missatge amb  $*$  s'executa abans que el missatge amb  $+$ , que és l'ordre habitual de les operacions en matemàtiques.

---

**Pas-a-pas 11.7 Descomposició de  $20 + (2 * 5)$**



- |   |          |
|---|----------|
| $20 + (2 * 5)$<br>(1) $2 * 5$<br>-retorna> 10 | "binari" |
| (2) $20 + 10$<br>-retorna> 30                 | "binari" |

---

**Nota A** Smalltalk, els selectors matemàtics de missatge com `+` i `*` tenen tots la mateixa prioritat. Els símbols `+` i `*` són senzillament selectors per a missatges binaris. Per tant, `*` no té prioritat sobre `+`. Si voleu forçar que una operació tingui precedència sobre una altra, hauríeu d'utilitzar parèntesis.

El fet que Smalltalk no segueix la precedència matemàtica pot ser confús al començament. Per tant, si teniu múltiples missatges binaris representant una expressió matemàtica, feu la vostra vida (i la de qualsevol que hagi de llegir el vostre codi) més fàcil i poseu parèntesis per expressar com s'hauria de dur a terme el càlcul. Quan estigueu més acostumats a la manera d'executar-se els missatges, probablement deixareu poc a poc de fer servir els parèntesis.

---

Una conseqüència de la regla 1 –que diu en quin ordre s'han d'executar els diferents tipus de missatges, amb els missatges unaris executant-se abans que els missatges binaris, i els missatges binaris abans que els missatges de paraula-clau– és que no heu d'utilitzar parèntesis gaire sovint. És a dir, la majoria de vegades no us heu d'amoïnar sobre l'ordre d'execució. La taula 11.4 mostra expressions escrites per ser executades d'acord amb les regles d'Smalltalk i expressions equivalents utilitzant parèntesis, que serien necessaris si l'ordre de precedència no existís.

**Taula 11.4 — Algunes expressions i els seus equivalents amb parèntesis.**

Sense Parèntesis	Expressions Equivalents amb Parèntesis
<code>pica color: Color groc</code>	<code>pica color: (Color groc)</code>
<code>pica ves: 100 + 20</code>	<code>pica ves: (100 + 20)</code>
<code>pica midaLlapis: pica midaLlapis + 2</code>	<code>pica midaLlapis: ((pica midaLlapis) + 2)</code>
<code>2 factorial + 4</code>	<code>(2 factorial) + 4</code>

## Resum

- Un missatge sempre s'envia a un objecte, anomenat el receptor del missatge, que pot ser el resultat d'altres missatges.
- Els missatges unaris són missatges que no tenen cap argument. Un enviament de missatge unari té la forma *receptor nomMissatge*.
- Els missatges binaris són missatges que involucren dos objectes: el receptor del missatge i un únic argument. El selector de missatge d'un missatge binari consisteix d'un o dos caràcters de la llista següent: +, \*, /, |, &, =, >, <, ~, @. Els enviaments de missatge binari tenen la forma *receptor nomMissatge argument*.
- Els missatges de paraula-clau són missatges que prenen un o més arguments i utilitzen una paraula-clau amb com a mínim un caràcter “dos punts” (:). Un enviament de missatge de paraula-clau amb dos arguments pren la forma *receptor primeraParaulaNomMissatge: primerArgument segonaParaulaNomMissatge: segonArgument*
- Regla 1: Els missatges unaris s'executen primer, després els missatges binaris, i finalment els missatges de paraula-clau.
- Regla 2: Igual que en matemàtiques, els missatges entre parèntesis s'executen abans que qualsevol altre missatge.
- Regla 3: Quan els missatges són del mateix tipus, l'ordre d'execució és d'esquerra a dreta.
- A Smalltalk, els selectors de missatge matemàtics, com + i \*, tenen tots la mateixa prioritat, i per tant, \* no té prioritat sobre +. Hauríeu d'utilitzar parèntesis si voleu que les vostres expressions matemàtiques s'executin en l'ordre adequat.



## Part III

# Posar en joc l'abstracció

En aquesta part del llibre aprendreu a definir els vostres propis mètodes. Això us permetrà reutilitzar seqüències de missatges, i sereu capaços de definir mètodes complexos a partir de mètodes més senzills.



## Capítol 12

# Mètodes: seqüències de missatges amb nom

Fins ara, heu estat utilitzant *scripts* per crear robots i enviar-los seqüències de missatges. Utilitzar *scripts* té l'avantatge de ser l'aproximació més immediata, però té limitacions importants. Una de les principals és que un *script* no pot ser invocat per un altre *script*. Aquest és un problema seriós, ja que un *script* no pot ser reutilitzat per altres *scripts*. Heu de reescriure la mateixa seqüència de missatges una vegada i una altra.

No estaria bé que poguéssim definir una mena d'*script* la seqüència de missatges del qual pogués ser enviada a qualsevol robot? De fet, això és possible i aquestes seqüències d'*scripts* es diuen *mètodes* (en el context d'aquest llibre no entrarem a estudiar tota la potència que poden proporcionar els *mètodes*, ja que això ens introduiria de ple en el complicat món de la programació orientada a objectes). Un *mètode* és un *script* amb nom. El nom del *mètode* pot utilitzar-se en un *script*, o fins i tot en un altre *mètode*, per invocar el *mètode*. Si us fixeu, no hi ha gairebé res de nou en això: tots els missatges per a robots que heu utilitzat fins ara representen *mètodes* que podeu fer servir amb qualsevol robot!

En aquest capítol aprendreu com definir *mètodes*. Ja sabeu gairebé tot el que cal saber per escriure un *mètode*. Tot i així, un *mètode* s'ha de definir amb un editor especial anomenat *explorador* (*o navegador*) de codi. Començarem comparant un *script* i un *mètode*. Després definirem un *mètode*, i finalment, entrarem en detall a veure què hem aconseguit.

## **Scripts versus mètodes**

Anem a retrobar un dels *scripts* que ja heu escrit, per exemple l'*Script 12.1*, que crea un robot i li diu com dibuixar un quadrat amb costat de mida 100 píxels.

**Script 12.1** *Pica dibuixa un quadrat senzill.*

```
| pica |
pica := Bot nou.
4 vegadesRepetir:
  [ pica giraEsquerra: 90.
    pica ves: 100 ]
```

El problema amb aquest *script* és que cada cop que necessiteu dibuixar un quadrat de costat 100 píxels heu de *copiar* les darreres tres línies de l'*Script 12.1*. És més, si voleu que un altre robot (per exemple, daly) dibuixi el quadrat, heu de canviar el nom pica a daly a tot arreu. Això ho podeu veure a l'*Script 12.2*.

**Script 12.2** *Pica i daly dibuixen cadascú un quadrat senzill.*

```
| pica daly |
pica := Bot nou.
daly := Bot nou.
daly salta: 200.
daly color: Color vermell.

4 vegadesRepetir:
  [ pica giraEsquerra: 90.
    pica ves: 100 ].
```

**4 vegadesRepetir:**

```
[ daly giraEsquerra: 90.
  daly ves: 100 ].
```

Per aquestes raons, treballar amb *scripts* no és fàcil. De fet, sospitem que les tres afirmacions següents reflecteixen part de la vostra experiència personal amb els *scripts*:

- Escriure *scripts* llargs és una tasca penosa.
- Repetir *scripts* llargs és avorrit i susceptible d'introduir errors.

- Quan un copia *scripts* complexos, la probabilitat de cometre un error de programació, com ometre una línia, és alta (un error de programació és un error dins la lògica del programa. Per contrast amb els errors de sintaxi, que són trobats ràpidament per l'ordinador ja que són errors en l'estructura del programa, els errors de programació poden ser bastant difícils de trobar).

Per superar aquestes dificultats, ens agradarà *definir* una seqüència de missatges una vegada, donar-li un *nom*, i després poder *enviar la seqüència amb nom* com un sol missatge a qualsevol robot, tal i com hem fet enviant missatges predefinitos com `ves:`, `nord` o `salta:`.

D'aquesta manera, podríem definir un *mètode* nou, anomenat `quadrat`, i aleshores escriure l'*Script* 12.3. No executeu l'*script* de moment, perquè el mètode `quadrat` encara no ha estat definit. Un cop tingueu el mètode `quadrat`, ja no us caldrà copiar ni adaptar la seqüència de missatges definint un quadrat. Senzillament podeu utilitzar-la dues vegades. L'enviament de missatge `pica quadrat` li dirà a pica que dugui a terme les instruccions codificades dins el mètode `quadrat`.

Esperem haver-vos convençut que aprendre a definir mètodes valdrà la pena.

### **Script 12.3 Pica i daly dibuixen quadrats utilitzant el mètode quadrat.**

```
| pica daly |
pica := Bot nou.
daly := Bot nou.
daly salta: 200.
daly color: Color vermell.
pica quadrat.
daly quadrat.
```

## Com definir un mètode?

En aquesta secció us donarem una recepta per crear mètodes. A Squeak podeu definir mètodes en qualsevol objecte, en aquest llibre, però, definireu mètodes només pels robots. Per ajudar-vos, hem desenvolupat un explorador de codi anomenat Explorador de la classe Bot, només per definir mètodes pels vostres robots. Podeu aconseguir un Explorador de la classe Bot arrossegant la seva icona des de la solapa blau fosc o via el menu **obre...** i després triant **explorador Bots**.

Utilitzar un Explorador de la classe Bot per definir un mètode requereix que (1) trieu o creeu una categoria nova de mètodes, que és una mena de carpeta de mètodes; (2) escriviu el mètode; i (3) el compileu. Descriurem aquests passos en les properes seccions. Primer, deixeu-nos explorar amb detall les diferents parts de l'Explorador de la classe Bot.

## Explorador de la classe Bot

Definir mètodes requereix una nova eina: l'editor mostrat a la figura 12.1. Aquest explorador és una versió simplificada de l'explorador que utilitzen els programadors en Smalltalk. L'explorador consisteix en tres parts, o *subfinestres*.



**Figura 12.1 — Un Explorador de la classe Bot mostrant la definició (subfinestra inferior) del mètode gira: (seleccionat a la subfinestra superior dreta) pertanyent a la categoria girar (seleccionada a la subfinestra superior esquerra).**

**Categories.** La subfinestra superior esquerra conté la *llista de categories*. Mostra les diferents categories de mètodes. Les categories de mètodes són noms per a grups de mètodes que ajuntem per trobar més de pressa la informació. A la figura 12.1, la categoria *girar* s'ha seleccionat; agrupa totes les operacions que tenen a veure amb els canvis de direcció dels robots. Podeu veure també altres categories que agrupen altres mètodes per als robots.

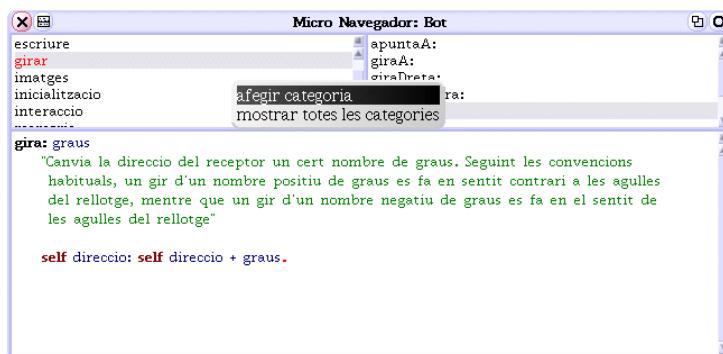
**Mètodes.** La subfinestra superior dreta conté la *llista de mètodes*. Aquesta llista mostra els noms dels mètodes dins la categoria seleccionada. A la figura 12.1, veiem llistats cinc mètodes apuntaA:, giraA:, giraDreta:, giraEsquerra: i gira:. El mètode *gira:* està seleccionat.

**Definició de Mètodes.** La subfinestra inferior conté l'*editor de codi*. Mostra la definició del mètode el nom del qual està seleccionat, juntament amb un comentari opcional. Aquesta subfinestra és també el lloc on podeu escriure el codi d'un mètode nou.

## Crear una nova categoria de mètodes

Agrupem els mètodes per categories. Una categoria es defineix donant-li un nom. Per definir un mètode, o bé definim una categoria nova pel mètode o bé seleccionem una categoria ja existent. Crearem una nova categoria de nom polígons regulars. Aquí teniu els detalls:

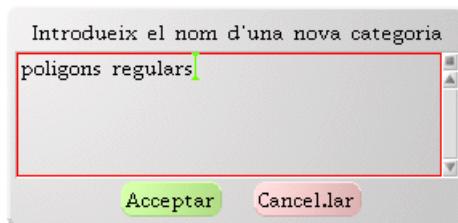
1. Feu clic amb el botó dret del ratolí (*Alt*-clic o bé *Option*-clic) a la llista de categories. Apareixerà un menú com el de la figura 12.2.



**Figura 12.2 —** Per crear una nova categoria de mètodes, obriu el menú de categories i seleccioneu afegir categoria.

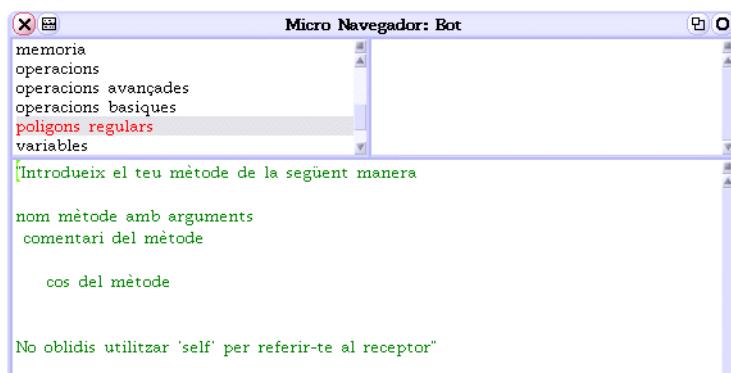
2. Seleccioneu l'opció afegir categoria d'aquest menú.
3. Escriviu el nom de la categoria al quadre de diàleg que apareixerà tot seguit, com podeu veure a la figura 12.3. Podeu triar qualsevol nom per a la categoria. Naturalment, noms significatius són millors que noms que no ho són quan voleu compartir la vostra feina amb altra gent o trobar el vostre mètode hores, dies o mesos més tard.
4. Feu clic al botó Acceptar per validar la vostra tria.

Com veieu a la figura 12.4, el nom de la nova categoria apareix a la subfinestra de categories i està seleccionat automàticament. L'editor està preparat per acceptar una nova definició de



**Figura 12.3** — Escriviu un nou nom de categoria al quadre de diàleg i feu clic al botó Acceptar.

mètode. Us mostra un recordatori de com cal definir un mètode, que haureu d'eliminar quan comenceu a escriure el vostre mètode. Ara ja esteu preparats per definir el vostre primer mètode.



**Figura 12.4** — La nova categoria està preparada.

## Definir el vostre primer mètode

Si la categoria on voleu afegir el vostre mètode no està seleccionada, seleccioneu-la. Aleshores escriviu el contingut del Mètode 12.1 (després d'aquest paràgraf) dins la subfinestra de l'editor de codi. Per fer això, seleccioneu tot el text dins l'editor de codi i comenceu a escriure el mètode.

**Mètode 12.1** *Un nou mètode per dibuixar un quadrat amb costats de mida 100 píxels.*

### quadrat

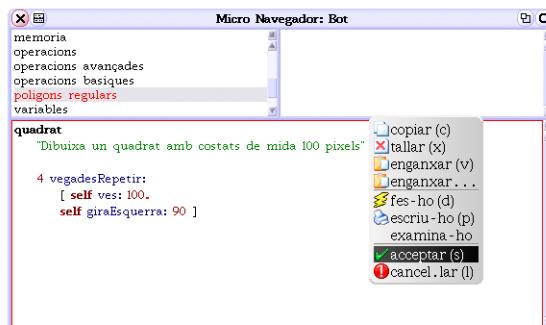
*"Dibuixa un quadrat amb costats de mida 100 píxels"*

4 vegadesRepetir:

```
[ self ves: 100.
  self giraEsquerra: 90 ]
```

Definir un mètode és un procés en tres passos:

- Escriure el mètode.** Escriure codi dins l'editor de codi funciona exactament igual que amb l'editor d'*scripts*. Primer elimineu el text que hi hagi a la subfinestra de l'editor de codi. La manera més senzilla de fer això és apuntar el ratolí al començament de la subfinestra de l'editor i fer clic abans del primer caràcter. Això seleccionarà tot el text de l'editor de codi. Un cop acabeu d'escriure el nou mètode, el codi dins la subfinestra de l'editor de codi hauria de semblar el que teniu a la figura 12.5.



**Figura 12.5 —** Després d'escriure el mètode *quadrat*, l'heu de compilar utilitzant el menú de l'editor de codi.

2. **Compilar el mètode.** Feu clic per fer aparèixer el menú de l'editor de codi, com veieu a la figura, i seleccioneu l'opció **acceptar**. En fer això la definició del vostre mètode es *compilarà*, és a dir, serà transformada a una representació que l'ordinador pugui entendre i executar. Un mètode nou anomenat quadrat ara apareix a la llista de mètodes. Si haguéssiu comès cap error mentre escrivíeu el mètode, Squeak es queixaria igual que ho feia amb els *scripts*. Si heu definit el mètode correctament, hauríeu de poder compilar el mètode sense que Squeak es queixés. L'explorador reflectirà el fet que la compilació és completa i que els robots ara poden entendre missatges que continguin el nou mètode mostrant el nom del nou mètode a la llista de mètodes (figura 12.6).
3. **Comprovar el mètode.** Segons la dita, la manera de comprovar si un pastís està ben fet és tastant-lo. Aquí igual, no heu acabat de crear el vostre mètode nou fins que l'heu comprovat, ja que el mètode que heu definit pot no fer el que teníeu al cap. Ara ja podeu executar l'*Script* 12.3. Hauríeu d'obtenir un quadrat negre i un altre de vermell.

Observeu que un mètode pot utilitzar-se molts cops, com queda demostrat a l'*Script* 12.3. Això ja ho sabíem. De fet, ho hem utilitzat des que vam començar el llibre: selectors de missatge com `ves:` o `giraEsquerra:` són els noms de mètodes definits de la mateixa manera que el mètode `quadrat`.

## Què hi ha en un mètode?

Us hem demanat que escriviu un mètode sense gaire explicacions. Ara és el moment d'analitzar l'estrucció d'un mètode.

Un mètode està compost d'un *nom*, un *comentari* opcional i el *cos del mètode* (una seqüència d'*expressions*), com es veu a la figura 12.6. El nom del mètode també pot contenir paràmetres (veure capítol 14) i el cos del mètode pot també definir variables locals utilitzant les barres verticals `| |`.

**Nom del mètode.** El nom d'un mètode hauria de representar el que fa el mètode, no com ho fa. Quan voleu que algú obri una porta, oi que no li expliqueu tota la física i les matemàtiques que hi ha relacionades? Doncs amb els mètodes és el mateix.

---

**Important!** El nom d'un mètode ha de representar sempre què fa el mètode, no com ho fa

---

Els noms de mètodes sense paràmetres, com `quadrat`, segueixen la mateixa sintaxi que els noms de les variables. Estan compostos de caràcters alfanumèrics (lletres i dígits) i comencen amb un caràcter en minúscules. En el nostre cas, el nom del mètode és `quadrat`.



**Figura 12.6** — Un mètode està format per un nom, un comentari opcional i el cos del mètode.

**Comentari del mètode.** Un comentari és un text entre cometes dobles (“Això és un comentari”). El text no pot contenir cometes dobles. El text pot ser tan llarg com vulgueu, fins i tot pot ocupar diverses línies.

En general un comentari explica el propòsit i l’efecte d’un mètode. Explica com podem utilitzar-lo, però no com fa la seva feina. Qui vulgui saber com funciona el mètode pot llegir-ne el cos.

Si el nom del mètode és prou clar, el comentari es pot ometre. En el nostre cas, el comentari del mètode és “Dibuixa un quadrat amb costats de mida 100 píxels”.

**Cos del mètode.** Després del comentari ve la definició del mètode, que és la seqüència de missatges que s’executaràn com a resposta al missatge. En el nostre cas, el cos del mètode és:

```
4 vegadesRepetir:
  [ self ves: 100.
    self giraEsquerra: 90 ]
```

**Important!** Un mètode és una seqüència d’expressions amb nom. Es compon d’un nom, un comentari opcional i una seqüència d’expressions. Un cop s’ha definit un mètode per a robots, qualsevol robot pot executar-lo en resposta a un missatge amb el mateix nom.

## **Scripts versus mètodes: Anàlisi**

Comparem el Mètode 12.2 amb l'*Script* 12.4. Podeu veure-hi tres diferències significatives: (1) La línia de l'*script* que declara la variable pica no és al mètode; (2) la línia que crea el robot tampoc hi és; (3) a la resta del mètode, la variable pica ha estat substituïda per self.

**Script 12.4** *Pica dibuixa un quadrat senzill.*

```
| pica |
pica := Bot nou.
4 vegadesRepetir:
    [ pica giraEsquerra: 90.
      pica ves: 100 ]
```

**Mètode 12.2** *Instruccions perquè qualsevol robot dibuixi un quadrat senzill.*

**quadrat**  
*“Dibuixa un quadrat amb costats de mida 100 píxels”*  
 4 vegadesRepetir:  
 [ **self** ves: 100.  
**self** giraEsquerra: 90 ]

Recordeu que un mètode per a un robot representa una seqüència d'expressions que pot ser enviada a *qualsevol* robot: El robot referenciat per la variable pica dins l'*script* no serà necessàriament el receptor del missatge quadrat. El robot daly, o qualsevol altre robot, podrien ser també receptors del missatge quadrat, com vam veure a l'*Script* 12.3.

Per tant, en definir el mètode quadrat és important no fer referència a cap robot *en particular*, ja que el missatge quadrat s'enviarà a robots *diferents* en moments diferents. Així doncs, ens cal un nom que representi al robot receptor del missatge quadrat, sigui quin sigui aquest robot. Aquest és el propòsit de self. Dins d'un mètode, self representa l'objecte receptor del missatge, ja que l'*objecte mateix* serà el que executi els missatges com ves: i giraEsquerra:.

## La Variable “self”

Al capítol 8 vam explicar que una variable és només un receptacle amb nom per a un objecte. En particular, vam posar èmfasi a que la mateixa variable podia ser utilitzada per apuntar a diferents objectes en diversos moments.

En el cas d'un mètode, la variable self apunta a l'objecte que ha rebut el missatge, sigui quin sigui aquest objecte: quan l'expressió pica quadrat és executada, la variable self dins del mètode

quadrat es refereix al robot anomenat pica, i quan l'expressió daly quadrat és executada, self es refereix al robot anomenat daly.

---

**Important!** Dins d'un mètode, la variable self representa l'objecte que ha rebut el missatge desencadenant de l'execució del mètode. Per exemple, quan l'expressió pica quadrat s'executa, pica rep el missatge quadrat i executa el mètode del mateix nom. La paraula self dins del mètode fa referència al robot anomenat pica, ja que pica està executant el mètode; quan l'expressió daly quadrat s'executa, self es refereix al robot daly.

---

La paraula self dins d'un mètode és una variable especial, ja que no en podeu canviar el valor. Només Squeak pot assignar un valor a self. Aquesta és la raó per la qual self no és declarada entre barres verticals | |. Encara més, self només es pot fer servir dins de la definició d'un mètode.

---

**Important!** Quan el codi d'un mètode necessita enviar un missatge al receptor, el missatge s'envia a self. Per exemple, al mètode quadrat, el robot que executa el mètode ha de girar, per tant el missatge gira: 90 és enviat a self.

---

## Mètode o no mètode: Aquesta és la qüestió

Ara per ara, potser esteu temptats de tornar enrere i convertir tots els *scripts* que heu escrit en mètodes. Això no és aconsellable, ja que no tots els *scripts* mereixen la conversió a mètodes. En general, hauríeu de definir un mètode quan teniu una seqüència de missatges prou general com per ser utilitzada diverses vegades.

## Retornar un valor

Un mètode també pot retornar un valor, utilitzant el caràcter ^, és a dir, l'accent circumflex. Quan escriviu un accent circumflex, Squeak escriu una fletxa apuntant cap a dalt (↑) dins l'entorn. Imagineu que voleu un mètode que retorni la distància més gran que li hauria d'estar permès recórrer a un robot d'un sol cop. Podríeu definir el mètode distanciaMaxima, mostrat com a Mètode 12.3. En aquest exemple, el mètode simplement retorna un nombre, però podria donar-se el cas que el mètode retornés el resultat d'una expressió complexa, i involucrés la posició actual del robot a la pantalla.

**Mètode 12.3** *Aquest mètode retorna un valor.*

distanciaMaxima

“retorna la distància més gran que li hauria d'estar permès recórrer a un robot d'un sol cop”

^100

Si un mètode no retorna explícitament un valor, aleshores retorna per defecte el receptor del missatge. El Mètode 12.4 és equivalent al mètode quadrat definit prèviament. De fet, al final de cada mètode hi ha una expressió implícita `^self` si no hi ha una expressió explícita de retorn. En aquest llibre, però, no us heu d'amoïnar per això.

**Mètode 12.4** *Aquesta versió equivalent del mètode quadrat retorna explícitament el receptor del missatge.*

quadratEquivalent

“Dibuixa un quadrat amb costats de mida 100 píxels”

4 vegadesRepetir:

```
[ self ves: 100.
  self giraEsquerra: 90 ]
```

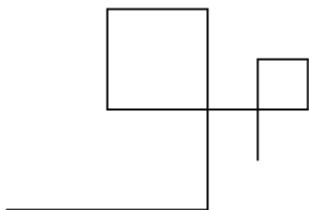
`^self`

En aquest llibre no utilitzareu gaire aquesta propietat, però és important saber que un mètode sempre retorna un valor.

## Dibuixar figures

Ara és hora de practicar. Com heu vist, és força senzill transformar un *script* en un mètode. Molts programadors experimentats utilitzen *scripts* per provar idees. Quan han comprovat, utilitzant *scripts*, que una idea és factible, mouen el codi a un mètode per poder reutilitzar-lo més tard. El proper exercici us servirà d'entrenament per fer això mateix. Considerem l'*Script 12.5*, que dibuixa un disseny abstracte “art nouveau”.

**Script 12.5** Pica dibuixa una senzilla figura abstracta.



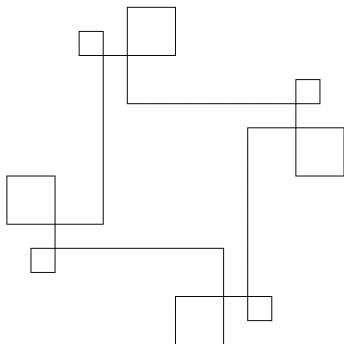
```
| pica |
pica := Bot nou.
pica ves: 100 ;
    giraEsquerra: 90 ;
    ves: 100 ;
    giraEsquerra: 90 ;
    ves: 50 ;
    giraEsquerra: 90 ;
    ves: 50 ;
    giraEsquerra: 90 ;
    ves: 100 ;
    giraEsquerra: 90 ;
    ves: 25 ;
    giraEsquerra: 90 ;
    ves: 25 ;
    giraEsquerra: 90 ;
    ves: 50
```

### Experiment 12-1 (un disseny abstracte senzill)

Creeu un mètode anomenat patro que generi la figura de l'*Script* 12.5

Ara podeu utilitzar aquest mètode dins un *script* per dibuixar un disseny més elaborat que podríeu fer servir com a marc “art nouveau”.

**Script 12.6** *Un marc art nouveau.*



```
| pica |
pica := Bot nou.
4 vegadesRepetir: [ pica patro ; ves: 50 ]
```

Ara el lector astut podria preguntar, “Per què no creem un mètode, anomenat marc50, per exemple, corresponent a l’*Script 12.6?*” Això és possible, ja que *qualsevol mètode creat per a un robot pot ser reutilitzat per qualsevol altre mètode de robots*. Crear aquests mètodes és el tema del proper capítol.

### Experiment 12-2 (un mètode pel marc art nouveau)

Creeu un mètode anomenat marc50 que generi el disseny de l’*Script 12.6*.

## Resum

- Un *mètode* és una seqüència d’expressions amb nom. Està compost d’un nom, un comentari i una seqüència d’expressions. Un cop s’ha definit un mètode per a robots, qualsevol robot pot executar-lo en resposta a un missatge amb el mateix nom.
- El nom d’un mètode sempre hauria de representar què fa el mètode, no com ho fa.
- Un mètode nou per a robots es crea utilitzant l’Explorador de la classe Bot, que és un editor especial per definir mètodes.

- Dins d'un mètode la variable `self` representa l'objecte que rep el missatge. Quan el codi del mètode necessita enviar un missatge al receptor, el missatge s'hauria d'enviar a `self`.

## Glossari

**Categories de Mètodes** Una categoria de mètodes és una carpeta on els mètodes es classifiquen. Les categories us ajuden a trobar els mètodes més ràpidament.

**Mètode** Un mètode representa una seqüència d'expressions que un objecte pot executar. Un mètode té un nom. S'executa quan l'objecte rep un missatge amb el mateix nom.

**Explorador de la Classe Bot** Un Explorador de la Classe Bot és una eina especial per veure i editar mètodes.

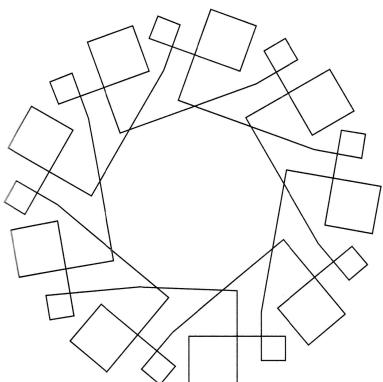
**Comentari** Un comentari és un text entre cometes dobles que explica el propòsit d'un mètode.

**self** La variable `self` està predefinida a Smalltalk. Sempre representa el receptor del missatge dins de la definició d'un mètode.



## Capítol 13

# Combinar mètodes



Heu après al capítol 12 com definir mètodes. Vam ensenyar-vos que definir mètodes és útil i interessant ja que (1) els mètodes us estalvien haver de reescriure *scripts*, la qual cosa requereix temps i és susceptible d'introduir errors, i (2) els mètodes poden ser utilitzats i reutilitzats per robots diferents. L'altre gran avantatge d'utilitzar mètodes és la possibilitat d'emprar mètodes dins d'altres mètodes, és a dir, cridar un o més mètodes ja existents com a part de la definició d'un mètode nou. La reutilització de mètodes és el que explorarem en aquest capítol.

És extremadament important que siguem capaços de reutilitzar mètodes, ja que podem definir un mètode en termes d'un altre sense haver de conèixer tots els detalls de la definició d'aquest altre mètode. Ens podem limitar a cridar-lo i demanar que faci allò pel que ha estat dissenyat.

## Res de nou: Revisitar el mètode quadrat

Tenir mètodes que cridin altres mètodes (cosa que anomenem *compondre mètodes*) és natural i no és realment nou. De fet, és el que vau fer al capítol 12 quan vau definir un mètode! El mètode quadrat inclou a la seva definició crides als mètodes `giraEsquerra()`, `ves()` i `vegadesRepetir()`: (com es pot veure al Mètode 13.1). Així, fins i tot el simple mètode quadrat està definit en termes d'altres mètodes, i no ens ha calgut saber com estan definitos `giraEsquerra()`, `ves()` i `vegadesRepetir()`. Només ens calia conèixer què feien. Per tant, ja hem acabat aquest capítol, sense res més a fer que divertir-nos una mica.

### Mètode 13.1

#### **quadrat**

*"Dibuixa un quadrat amb costats de mida 100 píxels"*

4 `vegadesRepetir:`

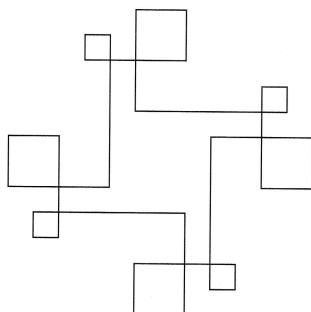
```
[ self ves: 100 ;
  giraEsquerra: 90 ]
```

## Altres patrons gràfics

Al capítol 12 us vam demanar de definir el mètode `patro`, que dibuixa un senzill patró abstracte (veure *Script 12.5*). Ara us demanarem de fer alguns experiments per generar més dibuixos definint més mètodes.

### Experiment 13-1

Definiu un mètode `patro4` que cridi a `patro` quatre vegades per generar la figura de més avall. Utilitzareu aquest mètode més tard, en un altre *script*. Després que hagueu creat el mètode `patro4`, utilitzeu l'*script* de tres línies següent per fer que pica dibuixi la figura



```
| pica |
pica := Bot nou.
pica patro4
```

---

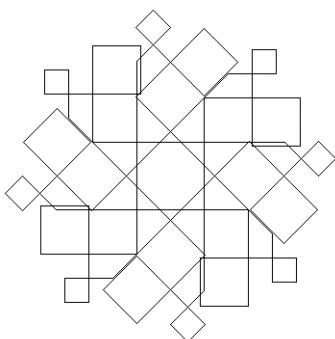
### Experiment 13-2 (una roda)

Definiu un mètode anomenat patroInclinat que generi la figura que podeu veure al començament d'aquest capítol, que sembla lleugerament una sínia. Pista: haureu de cridar a patro nou vegades, i l'angle que haureu de girar entre crides és de 10 graus.

---

### Experiment 13-3 (duplicar el marc)

Definiu el mètode dobleMarc, que podeu trobar més avall, per dibuixar la figura que veieu abans de la definició del mètode



**dobleMarc**

```
8 vegadesRepetir:
  [ self patro.
    self giraEsquerra: 45.
    self ves: 100 ]
```

---

## Què us diuen aquests experiments?

Què podeu aprendre dels experiments que acabeu de fer?. Com podeu veure en els mètodes patro4, patrolInclinat i dobleMarc, el mètode patro ha estat definit una sola vegada, i *reutilitzat diverses vegades* a mètodes diferents. Definir patro com a mètode us permet (1) definir-lo només una vegada, (2) reutilitzar-lo en contextos diversos, i (3) no introduir errors a causa de copiar una vegada i una altra aquest mètode.

Si pareu atenció a la definició del mètode dobleMarc, veureu que està definit en termes del mètode patro, definit ell mateix en termes d'altres mètodes, com ves: i giraEsquerra:. De fet, un mètode complex sovint es defineix en termes de mètodes més simples, que alhora estan definits en termes de mètodes encara més senzills, que alhora estan definits en termes de mètodes encara més senzills, que alhora... L'avantatge d'això és que els mètodes senzills són més fàcils d'entendre i definir que els mètodes complexos, i la tècnica de definir mètodes en termes de mètodes més simples limita el grau de complexitat de qualsevol mètode. Al capítol 16 us ensenyarem que per resoldre un problema, és avantatjós descompondre el problema en subproblemes més senzills, resoldre aquests subproblems, i després utilitzar les solucions als subproblems per resoldre el problema principal.

És essencial entendre que per definir el mètode dobleMarc, no us cal conèixer com està definit patro. Només heu de saber què fa i com utilitzar-lo! Quan definim un mètode estem donant un sol nom a una seqüència de missatges, la qual cosa redueix el nombre de detalls que hem de tenir en compte. Només ens cal recordar què fa el mètode, no com ho fa. Direm que estem construint una *abstracció* sobre els detalls de la definició. Per deixar-ho clar, hem escrit el mètode dobleMarc sense cridar el mètode patro, copiant-ne directament la definició (mostrada en itàlica). Compareu dobleMarcSenseCridarPatro (Mètode 13.2) amb el mètode dobleMarc. La nova versió sense patro no només és més llarga, sinó que per a la majoria de la gent és més confusa i difícil d'entendre.

Ara imagineu què passaria si féssim el mateix amb el codi de giraDreta:, giraEsquerra: i ves: -ja que aquests són també mètodes. Seria un malson! Hi hauria tants detalls que estaríem perduts tota l'estona.

**Mètode 13.2** *Crear el doble marc sense l'abstracció del mètode patro.*

#### dobleMarcSenseCridarPatro

8 vegadesRepetir:

```
[ self ves: 100.  
self giraDreta: 90.  
self ves: 100.  
self giraDreta: 90.  
self ves: 50.  
self giraDreta: 90.  
self ves: 50.  
self giraDreta: 90.  
self ves: 100.  
self giraDreta: 90.  
self ves: 25.  
self giraDreta: 90.  
self ves: 25.  
self giraDreta: 90.  
self ves: 50.  
self giraEsquerra: 45.  
self ves: 100 ]
```

---

**Important!** Quan escriviu un nou mètode, aquest pot cridar altres mètodes. Podeu utilitzar un mètode sense conèixer com està definit. Després d'acabar d'escriure un mètode, podeu cridar-lo quan definiu un altre mètode.

---

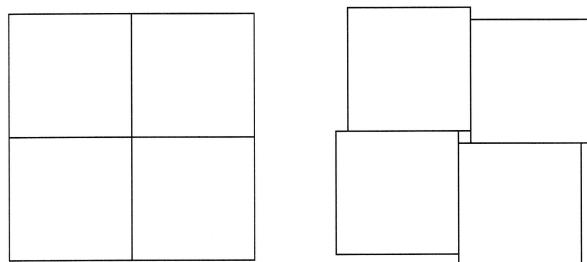
## Quadrats a tot arreu

Ara és el moment de practicar. Definiu els següents mètodes utilitzant el mètode quadrat.

### Experiment 13-4 (algunes capses)

Definiu els mètodes `capsa` i `capsaSeparada` que generi els dibuixos de la figura 13.1

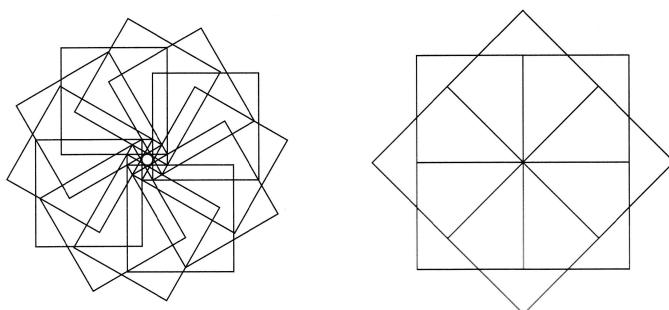
---



**Figura 13.1 — Capses**

### Experiment 13-5 (trieu vosaltres)

Utilitzeu els mètodes que ja heu definit per generar diverses figures. Vosaltres trieu quines figures. Divertiu-vos!



**Figura 13.2 — Estrelles**

### Experiment 13-6 (una estrella)

Utilitzant el mètode `capsa`, experimenteu i definiu un mètode `estrella` per generar el dibuix de la dreta de la figura 13.2

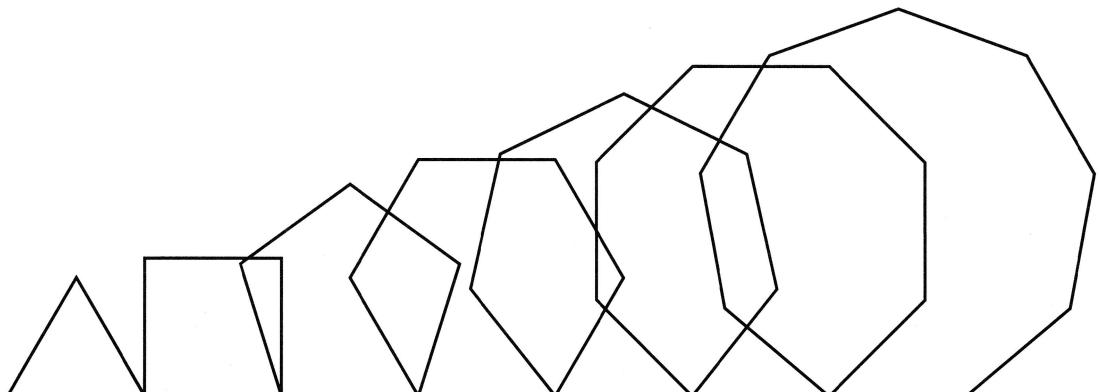
## Resum

- Quan definiu un mètode nou, pot cridar a altres mètodes.
- Podeu utilitzar un mètode sense conèixer com està definit; només us cal saber què fa.
- Després de definir un mètode, el podeu cridar quan definiu altres mètodes.
- Amagar els detalls d'un mètode tot donant-li un nom és el que anomenem *abstracció*.



## Capítol 14

# Paràmetres i arguments



En molts dels *scripts* que ja heu escrit, heu enviat missatges amb *arguments*. Per exemple, al missatge ves: 100 vau especificar que un robot s'hauria de moure una distància de 100 pixels, i l'argument del missatge és per tant 100. Malgrat que heu après com definir mètodes, no heu après encara com definir mètodes que requereixin un o més arguments.

En aquest capítol aprendreu com definir mètodes el comportament dels quals depengui dels valors dels arguments del missatge. Direm que aquests mètodes tenen *paràmetres*, i que el seu comportament està parametritzat. Els paràmetres d'un mètode actuen com a receptacles a la definició del mètode, i aquests receptacles s'omplen amb els arguments del missatge quan el missatge és enviat. Primer definirem un mètode amb un paràmetre, i l'invocarem. Després l'analiitzarem.

## Què és un paràmetre?

El mètode quadrat definit al capítol 12 és una mica limitat, ja que la mida del quadrat es fixa una vegada i ja no es pot modificar. Us podeu haver preguntat, “Què podríem fer per poder dibuixar un quadrat de costats de mida 300 píxels, o 175, o 225, o fins i tot 23 píxels?” No hi ha res que us impedeixi definir els mètodes quadrat300, quadrat175, quadrat225, quadrat23 i així successivament.

Crear múltiples mètodes quadrat, però, no resol el problema de manera satisfactòria. Seria molt incòmode si haguéssim de definir un mètode nou cada cop que volguéssim dibuixar un quadrat de mida diferent. El que realment volem és tenir un mètode general per dibuixar quadrats, que permeti a l’usuari especificar la mida del costat del quadrat en el moment de cridar al mètode. D’aquesta manera, no hauríem de definir un mètode nou per a cada quadrat de mida diferent.

El que necessitem és substituir la mida fixa del costat del quadrat per una mena de variable el valor de la qual serà assignat en el moment d’enviar el missatge, no abans. Aquesta mena de variable existeix en molts llenguatges de programació. S’anomena un *paràmetre*. Un paràmetre d’un mètode és una variable especial que pot prendre qualsevol valor *en el moment de l’enviament del missatge*. Així doncs, serveix com a receptacle quan es defineix el mètode i per tant no rep cap valor en la definició del missatge.

Això us hauria de sonar. Després de tot, ja sabeu que mètodes com ves: i giraEsquerra: necessiten un argument (una distància o un angle) en el moment de l’enviament del missatge. Cada vegada que escriviu una expressió com pica giraEsquerra: 90, pica giraEsquerra: 32 o daly giraEsquerra: 65 afegiu en el missatge el valor d’un angle com a argument, i aleshores el mètode giraEsquerra: utilitza aquest valor per decidir quin angle ha de girar el robot. De fet, en totes i cada una d’aquestes expressions, és el mateix mètode giraEsquerra: el que s’executa, cada cop amb un *valor diferent* per a l’angle que pica o daly ha de girar. Ser capaç d’especificar diferents angles en enviaments de missatge diferents utilitzant el mateix selector de missatge giraEsquerra: és una valuosa propietat del mètode giraEsquerra:. Vosaltres voldríeu que el mètode per dibuixar quadrats tingués aquestes propietats, igualment amb altres mètodes que pugueu definir. I els vostres mètodes les tindran aquestes propietats, tan aviat com expliquem com definir mètodes que, com el mètode giraEsquerra:, puguin tenir un o més arguments en temps d’execució.

## Un mètode per dibuixar quadrats

Ja heu vist que en Smalltalk, el nom d’un selector de missatge acaba en “dos punts” (:) per indicar que el missatge necessita un argument. Igualment, el nom del mètode corresponent a aquest selector de missatge també acaba en “dos punts” (:), i a la definició del mètode, un paràmetre és utilitzat com a receptacle per a l’argument del missatge. Així, si voleu crear un mètode per dibuixar un quadrat de mida arbitrària, podríeu utilitzar quadrat: com a nom i midaCostat com a paràmetre. Podeu definir aleshores el mètode quadrat: com veieu al Mètode 14.1. Aquest mètode

s'utilitza en l'*Script* 14.1, en el qual pica dibuixa dos quadrats, un de costat de mida 10 i l'altre de costat de mida 20.

**Mètode 14.1** *El mètode quadrat: utilitza el paràmetre midaCostat per dibuixar un quadrat de mida arbitrària.*

quadrat: **midaCostat**

*“Dibuixa un quadrat amb costats de la mida donada”*

4 vegadesRepetir:

```
[ self ves: midaCostat .
  self giraEsquerra: 90 ]
```

**Script 14.1** *El mètode quadrat: és utilitzat per dibuixar quadrats de diferents mides.*

```
| pica |
pica := Bot nou.
pica quadrat: 10.
pica ves: 300.
pica quadrat: 20.
```

Ara analitzem la definició del mètode quadrat: al Mètode 14.1. Per definir un mètode que requereix un argument, cal que el nom del mètode acabi amb “dos punts” i sigui seguit del nom del paràmetre, que aquí és midaCostat.

El paràmetre representa una variable el valor de la qual es defineix quan el missatge és enviat (no quan el mètode és definit). A l'*Script* 14.1, quan s'envia el missatge quadrat: 10, el paràmetre midaCostat pren el valor 10. Després, quan s'enviï el missatge quadrat: 20, midaCostat prendrà el valor 20. Els paràmetres no es declaren explícitament utilitzant les barres verticals | |, com les variables normals dels *scripts*.

Al Mètode 14.1, el nom del paràmetre és midaCostat. El nom del paràmetre hauria de triar-se per indicar per a què s'utilitza. Podrieu haver-lo anomenat longitud, com en el Mètode 14.2 (o mida, o qualsevol altra cosa), sempre i quan substituïssiu totes les aparicions de midaCostat dins del cos del mètode per longitud. Fixeu-vos que el Mètode 14.1 i el Mètode 14.2 produueixen *exactament* els mateixos resultats. Això és degut al fet que midaCostat i longitud són noms per a exactament la mateixa cosa: el paràmetre del mètode quadrat::

**Mètode 14.2****quadrat:** *longitud**"Dibuixa un quadrat amb costats de la mida donada"*

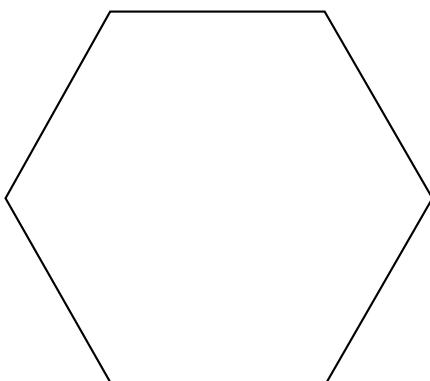
4 vegadesRepetir:

[ self ves: **longitud** .  
self giraEsquerra: 90 ]**Practiqueu amb els paràmetres**

Ara és moment de practicar una mica. Comencem amb un exercici senzill.

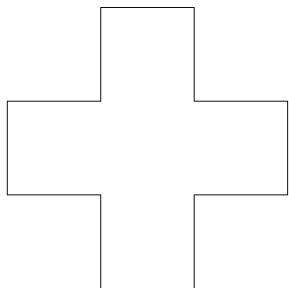
**Experiment 14-1 (un mètode per dibuixar un hexàgon)**

Definiu un mètode hexagon:, que dibuixa un hexàgon amb la mida dels costats com a argument



### Experiment 14-2 (un mètode per dibuixar una creu)

Transformeu l'*script* que us donem més avall en un mètode anomenat *creu*: que dibuixa una creu amb la mida d'un dels seus braços com a argument. Després, hauríeu d'executar l'expressió *pica creu: 100*. Pista: fixeu-vos que  $50 = 100/2$ . Un bon nom per al paràmetre seria *longitudBraç*.



```
| pica |
pica := Bot nou.
4 vegadesRepetir:
  [ pica ves: 50.
    pica giraEsquerra: 90.
    pica ves: 100.
    pica giraDreta: 90.
    pica ves: 100.
    pica giraDreta: 90.
    pica ves: 50 ]
```

---

## Variabes en mètodes

Tal com hem fet servir variables en els nostres *scripts* per anomenar algunes quantitats, també podem utilitzar variables en mètodes pel mateix propòsit. Si volguéssim dir a *pica* que dibuixés un polígon amb costats de mida 100, podríem escriure alguna cosa similar a l'*Script 14.2*. Com que el valor de l'angle que *pica* ha de girar depèn del nombre de costats del polígon, hem introduït les variables *nombreDeCostats* i *angle*. Assignem a *nombreDeCostats* un cert valor (*nombreDeCostats := 6* en el nostre exemple) i després assignem a *angle* un valor que és calculat en funció del valor de *nombreDeCostats*. Ara, si volem canviar el nombre de costats del polígon de 6, tal i com està definit a l'*script*, per qualsevol altre valor, només cal canviar el valor assignat a *nombreDeCostats* a la tercera línia de l'*script*, sense haver de preocupar-nos de l'*angle*.

**Script 14.2** *Dibuixar un polígon en un script utilitzant variables.*

```
| pica nombreDeCostats angle |
pica := Bot nou.
nombreDeCostats := 6.
angle := 360 / nombreDeCostats.
nombreDeCostats vegadesRepetir:
[ pica ves: 100.
  pica giraEsquerra: angle ]
```

Per convertir l'*Script 14.2* en un mètode, podem definir un paràmetre pel nombre de costats. Això ho fem al Mètode 14.3, que defineix el mètode `poligon100`: per dibuixar un polígon amb un nombre arbitrari de costats, tots de mida 100 píxels.

**Mètode 14.3** *Dibuixar un polígon en un mètode utilitzant una variable i un paràmetre.*

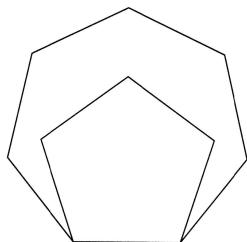
`poligon100: nombreDeCostats`  
 "Dibuixa un polígon amb un nombre arbitrari de costats;  
 la mida de cada costat és de 100 píxels"

```
| angle |
angle := 360 / nombreDeCostats.
nombreDeCostats vegadesRepetir:
[ self ves: 100.
  self giraEsquerra: angle ]
```

El mètode té un argument, `nombreDeCostats`, i una variable, `angle`. Tots dos són utilitzats dins del cos del mètode. Com que `nombreDeCostats` és un *paràmetre*, el seu valor s'especifica a l'argument de qualsevol missatge que cridi el mètode, per exemple, `pica poligon100: 7` per a un polígon regular de 7 costats (heptàgon). La *variable* `angle` s'inicialitza dins del text del mètode assignant-hi l'angle adequat, que depèn del valor del paràmetre *en el moment de l'enviament del missatge* (a l'exemple serà  $360 / 7$ ). Per qualsevol valor de `nombreDeCostats`, la variable `angle` tindrà el valor correcte per a un polígon regular amb aquell nombre de costats.

Ara que ja teniu el mètode `poligon100:`, podeu utilitzar-lo per dibuixar polígons, com veieu a l'*Script 14.3*.

**Script 14.3** Utilitzar el mètode poligon100: per dibuixar un heptàgon i un pentàgon.



```
| pica berthe |
berthe := Bot nou.
pica := Bot nou.
berthe poligon100: 5.
pica poligon100: 7.
```

## Experimentar amb múltiples arguments

Per quina raó hauríem de limitar-nos a polígons amb costats de mida 100? No seria millor tenir un mètode que dibuixés un polígon regular arbitrari, on tant el nombre de costats com la seva mida fossin determinats en el moment d'enviar el missatge? Per aconseguir això, caldrien dos paràmetres: nombreDeCostats i midaCostat. Així, com creem un mètode amb dos paràmetres? Podeu crear un mètode amb dos paràmetres escrivint el nom del mètode amb dos "dos punts" i posant un argument darrera de cada "dos punts".

---

**Nota** Per definir un mètode amb múltiples paràmetres, acabeu cada paraula del nom del mètode (una paraula per a cada paràmetre) amb "dos punts", i poseu cada paràmetre darrera la seva paraula corresponent en el nom del mètode. El mètode anomenat poligon:mida: requereix dos arguments. La definició del mètode poligon: nombreDeCostats mida: valorMida defineix dos paràmetres, nombreDeCostats i valorMida. El primer paràmetre representa, com queda clar pel nom, el nombre de costats. El segon paràmetre està relacionat amb la mida del polígon. Ho explicarem tot seguit després de la definició del mètode.

---

La definició del mètode poligon:mida: la teniu com a Mètode 14.4. Després de definir el mètode, podeu simplement enviar un missatge com ara pica poligon: 7 mida: 100.

### Mètode 14.4

poligon: nombreDeCostats mida: valorMida

“Dibuixa un polígon amb un nombre de costats i una mida per especificar”

```
| angle midaCostat |
angle := 360 / nombreDeCostats.
midaCostat := 4 * valorMida / nombreDeCostats.
nombreDeCostats vegadesRepetir:
[ self ves: midaCostat.
self giraEsquerra: angle ]
```

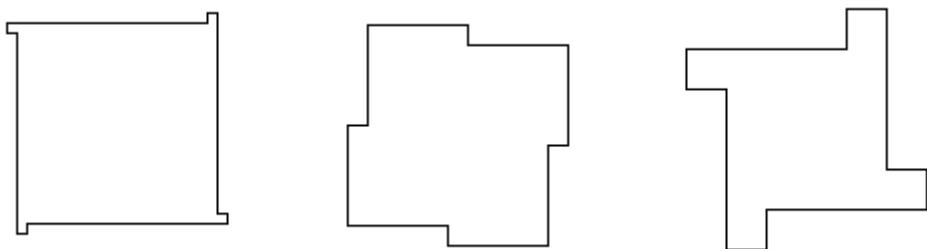
Us podeu preguntar per quina raó el paràmetre `valorMida` no especifica la mida del costat del polígon, i en canvi hem decidit definir la mida del costat (donada per la variable `midaCostat` en el mètode) com a  $4 * \text{valorMida} / \text{nombreDeCostats}$ . Per fer que tots els polígons amb el mateix `valorMida` tinguin aproximadament la mateixa mida, hem fet que tots els polígons tinguin el perímetre igual al perímetre d'un quadrat amb costats de mida `valorMida`. El perímetre d'aquest quadrat és  $4 * \text{valorMida}$ . El resultat és que fent que `midaCostat` sigui igual a  $4 * \text{valorMida} / \text{nombreDeCostats}$ , quan el robot dibuixa `nombreDeCostats` costats cada un de mida `midaCostat`, acaba fent un recorregut de distància igual al perímetre d'un quadrat amb costats de mida `valorMida`. Així doncs, qualsevol polígon dibuixat amb 100 de segon argument tindrà un perímetre de valor 400 pixels, i tots els polígons ocuparan la mateixa fracció de la pantalla en ser dibuixats.

Podeu pensar que el nom del paràmetre `nombreDeCostats` és una mica llarg i incòmode. Tot i així, és un bon nom per a un paràmetre, ja que pot ser comprès fàcilment per qualsevol persona que llegeixi el mètode. Com ja vam comentar en el capítol 9, és molt important que el vostre codi sigui llegible, quasi bé com una narració, per a tothom. I això us inclou a vosaltres: el nom d'una variable o d'un paràmetre que no està clar us pot deixar aturats quan torneu al vostre codi al cap d'uns mesos d'haver-lo escrit.

### Experiment 14-3

Definiu un mètode anomenat `rectangleAmplada:altura:` que dibuixa un rectangle amb la seva amplada i la seva altura com a arguments.

---



**Figura 14.1** — Tres creus estilitzades generades pel mètode `creuRecorregut1:recorregut2:`. La creu de l'esquerra és el resultat de pica `creuRecorregut1: 5 recorregut2: 50`; la creu del mig és el resultat de pica `creuRecorregut1: 50 recorregut2: 5`; la creu de la dreta és el resultat de pica `creuRecorregut1: 20 recorregut2: 40`.

### Experiment 14-4

Modificant lleugerament el mètode `creu:` que vau escriure a l'Experiment 14-2, definiu un mètode `creuRecorregut1:recorregut2:` que pugui dibuixar les creus estilitzades mostrades a la figura 14.1. Ordeneu els paràmetres de manera que una creu normal com la que dibuixaríeu amb `creu:` tingui el primer paràmetre igual al doble del segon, i per tant es pugui dibuixar amb expressions com pica `creuRecorregut1: 60 recorregut2: 30`.

## Paràmetres i variables

Ara que ja heu practicat una mica, és el moment per investigar amb una mica de cura la diferència entre les variables normals i els paràmetres. Comparem l'*script* i el mètode que ja hem definit per dibuixar un quadrat amb costats de mida arbitrària. Els hem tornat a reproduir com a *Script* 14.4 i Mètode 14.5.

A l'*Script* 14.4, la variable `midaCostat` és declarada (línia 1), després rep un valor (línia 3), i finalment s'utilitza com a argument del mètode `ves:` (línia 5).

**Script 14.4** L'*script* del quadrat que utilitza una variable.

- (1) | pica midaCostat |
- (2) pica := Bot nou.

- (3) midaCostat := 10.
- (4) 4 vegadesRepetir:
- (5) [ pica ves: midaCostat.
- (6) pica giraEsquerra: 90 ]

El Mètode 14.5 mostra exemples de dues propietats dels paràmetres. Primer el paràmetre midaCostat és declarat posant-lo darrera de “dos punts” en el nom del mètode (línia 1). Segon, és utilitzat com a argument del missatge ves: (línia 5). Un paràmetre no s’inicialitza en la definició del mètode ja que sempre rep el seu valor de l’argument corresponent en qualsevol missatge que invoqui el mètode. Per exemple, quan el missatge quadrat: 20 s’envia a pica, aleshores el paràmetre midaCostat del Mètode 14.5 rep el valor 20.

**Mètode 14.5** *El mètode que dibuixa quadrats utilitzant un paràmetre*

- (1) quadrat: midaCostat
- (2) “Dibuixa un quadrat amb costats de la mida donada”
- (3)
- (4) 4 vegadesRepetir:
- (5) [ self ves: midaCostat.
- (6) self giraEsquerra: 90 ]

Així doncs, hi ha tres clares diferències entre paràmetres i variables:

**Els paràmetres no es declaren explícitament.** Els paràmetres no es declaren entre barres verticals | |, com cal fer amb les variables normals. Un paràmetre és declarat quan apareix darrera els “dos punts” a la primera línia de la definició d’un mètode.

**Als paràmetres no se’ls assigna valors.** Els paràmetres no es poden modificar de la mateixa manera que modifiquem les variables. No podem assignar nous valors als paràmetres dins del cos de la definició d’un mètode. Per exemple, l’expressió midaCostat := 100 és impossible dins del Mètode 14.5. No podem modificar els valors d’un paràmetre ja que són una mena especial de variable. Són receptacles pels arguments utilitzats quan un missatge invoca el mètode corresponent, i els seus valors són assignats per Squeak quan un missatge és enviat, per tant no es poden assignar explícitament valors als paràmetres utilitzant :=.

**Inicialització de variables.** Les variables normals i els paràmetres obtenen els seus valors de maneres molt diferents. El valor d’una variable és canviat utilitzant una assignació explícita amb :=. El valor d’un paràmetre és assignat quan el mètode és invocat per un missatge.

Per exemple, l'enviament de missatge pica quadrat: 10 fa que el paràmetre `midaCostat` rebi el valor 10. Per tant, un paràmetre és una variable, però d'un tipus especial el valor de la qual és assignat només en el moment en què un missatge és enviat i el mètode corresponent és executat.

---

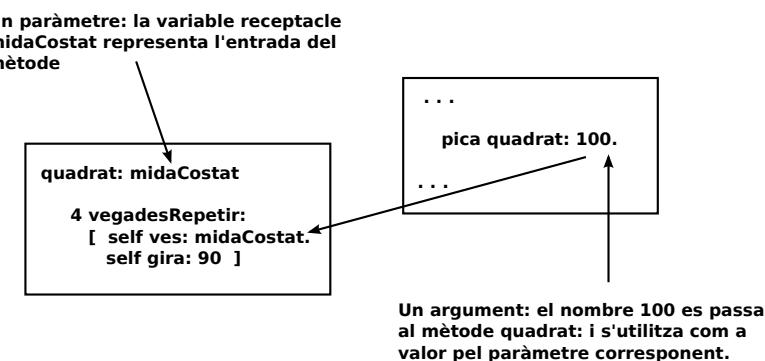
**Nota** A part de les tres diferències entre paràmetres i variables normals, un paràmetre pot ser utilitzat dins el codi de la definició d'un mètode de la mateixa manera que qualsevol altra variable.

---

## Arguments i paràmetres

Hem introduït els dos termes *argument* i *paràmetre* per a dues idees relacionades, tot i que diferents. Un argument és un objecte específic passat en un missatge. Un paràmetre és una variable receptacle utilitzada en la definició d'un mètode el valor de la qual és desconeguda quan el mètode és definit. Un paràmetre pren el seu valor de l'argument corresponent<sup>1</sup>

A la figura 14.2, en el missatge `quadrat: 100`, el nombre 100 és l'argument del missatge. Quan el mètode `quadrat:` s'executa, el seu paràmetre `midaCostat` és col·locat a 100, el valor de l'argument.

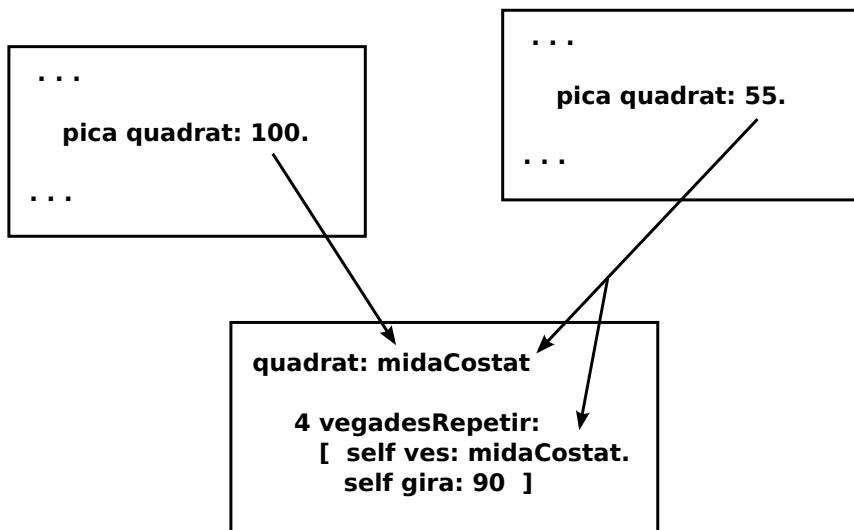


**Figura 14.2 — La relació entre argument (un objecte) i paràmetre (una variable receptacle).**

---

<sup>1</sup>Molts autors defineixen aquests termes de manera diferent. Alguns utilitzen “paràmetre real” pel que nosaltres anomenem “argument” i “paràmetre formal” pel que nosaltres anomenem “paràmetre”. D’altres utilitzen els termes “paràmetre” i “argument” de manera intercanviable.

Una altra manera d'entendre la diferència entre un argument i un paràmetre és que un paràmetre és un receptacle dins del mètode que representa una dada d'entrada pel mètode, mentre que un argument és el valor real que és passat com a entrada. Aquesta idea s'il·lustra a la figura 14.3.



**Figura 14.3** — El valor d'un argument queda lligat al paràmetre mentre el mètode s'executa.

Fixeu-vos que un paràmetre es pot utilitzar també com a argument en altres enviaments de missatges. Per exemple, a la definició del mètode quadrat: (Mètode 14.5), el paràmetre midacostat s'utilitza com a argument del missatge ves: midacostat.

L'argument d'un missatge pot ser també una variable. Per exemple, a l'*Script 14.5*, que utilitza el mètode quadrat:, l'argument del primer missatge quadrat: és el valor de la variable midacostat, que és 100. L'argument del segon missatge quadrat: és el valor de l'expressió midacostat + 200, que és 300. El paràmetre midacostat del mètode quadrat: pren el valor 100 al primer missatge quadrat:, i després el valor 300 del segon missatge quadrat.

#### **Script 14.5** Una variable com a argument.

```
| pica midaQuadrat |
pica := Bot nou.
midaQuadrat := 100.
pica quadrat: midaQuadrat.
```

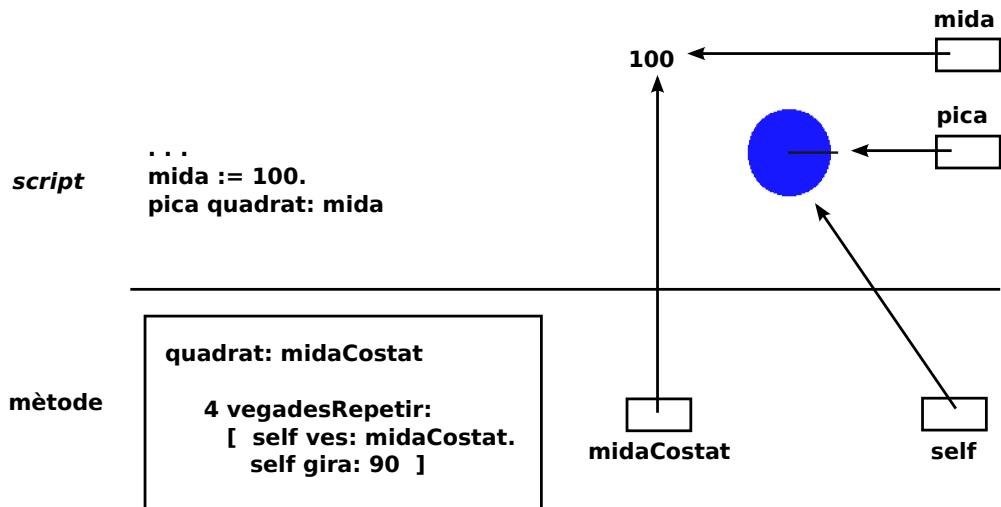
pica ves: 300.

pica quadrat: midaQuadrat.

## Sobre l'execució dels mètodes

En una primera lectura podeu saltar-vos aquesta secció, ja que entra en detalls que potser als programadors novells no els cal conèixer. L'hem escrit ja que volem respondre les preguntes dels lectors més curiosos, però ho podríem haver omès sense perdre continuitat.

Quan s'executa un mètode, es creen algunes variables. Aquestes variables són `self`, el receptor del missatge, i els paràmetres del mètode (que fan referència als arguments del mètode), com `midaCostat` a la figura 14.4, que mostra l'efecte d'enviar el missatge quadrat: `mida` a un robot a què ens referim amb la variable `pica`, on la variable `mida` fa referència al nombre 100.



**Figura 14.4** — Quan s'envia un missatge i s'executa un mètode, es creen noves variables que fan referència als arguments i al receptor del missatge.

Quan el mètode quadrat: s'executa, la variable self fa referència al receptor del missatge, que en el nostre exemple és el robot apuntat per la variable pica; i el paràmetre midacostat fa referència al valor de la variable midat, que aquí és el nombre 100. El mateix procés succeeix per a cada enviament de missatge. Per exemple, l'execució de l'expressió daly quadrat: 200 assigna a self el robot referenciat per la variable daly i assigna a midacostat el nombre 200.

Això pot semblar complicat, però no us heu d'amoïnar. Aquest són els passos amagats que Squeak realitza per assegurar-se que els paràmetres prenen els valors dels arguments dels missatges.

## Resum

- Un paràmetre és una mena especial de variable que actua com a receptacle per als arguments dels missatges. El paràmetre d'un mètode es declara tot just després dels "dos punts" en el nom del mètode, indicant la posició del paràmetre. Un paràmetre no s'ha de declarar com una variable, i no se li poden assignar valors dins del cos de la definició del mètode. Els paràmetres reben els seus valors dels arguments del missatge que invoca el mètode.
- Per definir un mètode amb múltiples arguments, cal acabar cada paraula del nom del mètode amb "dos punts", i col·locar cada paràmetre després de la corresponent paraula del nom del mètode. Per exemple, el mètode anomenat poligon:mida: requereix dos arguments. La definició del mètode poligon: nombreDeCostats mida: valorMida defineix dos paràmetres, nombreDeCostats i valorMida.

# Capítol 15

## Errors i depuració

Ara que ja sabeu com definir mètodes que criden altres mètodes, podeu escriure programes més i més complexos. Més tard o més d' hora cometreu algun error i no aconseguireu trobar què és el que està malament. Els errors en els programes d'ordinador s'anomenen *bugs*<sup>1</sup>, i us ensenyarem com utilitzar una eina molt potent que us ajudarà a trobar aquests errors, i corregir-los: el depurador d'Squeak (l'*Squeak debugger*)<sup>2</sup>.

Un *depurador* és una eina que mostra l'execució d'un programa. Us deixa inspeccionar i canviar els valors de les variables i editar els mètodes d'un programa. En aquest capítol us ensenyarem alguns exemples típics d'errors i us explicarem què és el depurador i com utilitzar-lo. Començarem presentant alguns errors comuns amb variables. Després us mostrarem com utilitzar el depurador per identificar i arreglar altres tipus de problemes.

### El valor per defecte d'una variable

Les variables són molt útils per programar, però requereixen que hi poseu una mica d'atenció. Per exemple, podeu introduir fàcilment un error en un programa utilitzant una variable que no ha estat declarada o a la qual s'han assignat valors incorrectes. Fins i tot els programadors amb experiència cometren errors. Aquests, però, saben com trobar-los i eliminar-los. Squeak us ajuda una mica comprovant, per exemple, si les variables que utilitzeu han estat declarades. Aquests errors estructurals, o *sintàctics*, són fàcils de localitzar, i Squeak els localitzarà. Tot i així, Squeak no té cap manera de detectar errors *lògics*, com ara assignar a una variable el valor 1000 quan li

---

<sup>1</sup>Nota del Traductor: D'on ve l'expressió *debugging* que aquí hem traduït com a "depuració", seguint els consells de softcatalà.

<sup>2</sup>Nota del Traductor: Com ja sabeu, el depurador d'Squeak és una eina de l'entorn general, i no pròpiament de l'entorn BotsInc, per tant no ha estat traduïda, tal i com hem fet amb algunes eines que ja hem trobat en aquest llibre.

hauríeu d'haver assignat el valor 100. Aquests errors els haureu de trobar vosaltres mateixos, i utilitzar un depurador us pot ajudar a entendre, localitzar i eliminar els vostres errors.

Primer, experimentem una mica. Escriviu, seleccioneu i demaneu el resultat de l'*Script 15.1*, que declara la variable *mida* i prova de fer-la servir abans de ser inicialitzada. Hauríeu d'obtenir la figura 15.1, que mostra com Squeak us avisa quan una variable no ha estat inicialitzada. Si trieu l'opció *si* quan us pregunti, hauríeu d'obtenir *nil* escrit com veieu a la dreta de la figura 15.1. El valor *nil* que tot just heu obtingut és un objecte especial assignat a qualsevol variable declarada. Començarem per mirar el valor per defecte d'una variable.



**Figura 15.1** — Esquerra: *Squeak us avisa quan proveu d'utilitzar una variable que no ha estat inicialitzada*. Dreta: *El valor nil s'escriu si decidiu procedir respondent si*.

**Script 15.1** *Provar d'utilitzar una variable abans d'inicialitzar-la.*

```
| mida |
mida
```

La raó de la queixa d'Squeak quan utilitzeu una variable que no ha estat inicialitzada és que si no inicialitzeu una o més variables, és pràcticament segur que el vostre programa és incorrecte, ja que estareu utilitzant una variable que té un valor incorrecte o fins i tot cap valor. El valor de les variables a Smalltalk és per defecte l'objecte *nil*, que representa un valor indefinit. Tot i que *nil* és un objecte com qualsevol altre, no entén gaires missatges. En particular, no entén cap dels missatges que s'envien als objectes nombre o als objectes robot. Per tant, obtindreu un error en provar d'enviar un d'aquests missatges a l'objecte *nil*. L'objecte *nil* és important per determinar si una variable té o no un valor assignat.

---

**Important!** A Smalltalk, el valor d'una variable no inicialitzada és per defecte l'objecte nil, que s'utilitza per representar un valor indefinit.

---

L'error que succeeix mentre provem d'executar l'*Script* 15.1 genera el missatge que podem veure a la figura 15.1. Normalment, és suficient respondre **no** i tornar a l'*script* per inicialitzar la variable. Tot i així, malgrat Squeak és capaç d'analitzar els *scripts* i trobar aquests errors estructurals, no pot detectar, abans d'executar un *script*, si heu utilitzat una valor *no vàlid*. En aquest cas, Squeak obre un *depurador* (*debugger* en anglès) que podeu utilitzar per accedir l'estat de l'execució, que inclou el receptor del missatge, el missatge mateix, les variables involucrades, etc. Això és el que ara us explicarem.

## Examinar l'execució d'un missatge

Per si de cas us heu oblidat de les definicions dels mètodes patro i patro4, els hem tornat a escriure aquí (Mètodes 15.1 i 15.2). El que és important és adonar-se que el mètode patro4 utilitza el mètode patro, mentre que el mètode patro invoca els mètodes ves: i giraDreta:.

### Mètode 15.1

#### patro

"dibuixa un patró"

```
self ves: 100.  
self giraDreta: 90.  
self ves: 100.  
self giraDreta: 90.  
self ves: 50.  
self giraDreta: 90.  
self ves: 50.  
self giraDreta: 90.  
self ves: 100.  
self giraDreta: 90.  
self ves: 25.  
self giraDreta: 90.  
self ves: 25.  
self giraDreta: 90.  
self ves: 50.
```

## Mètode 15.2

### **patro4**

“dibuixa quatre patrons”

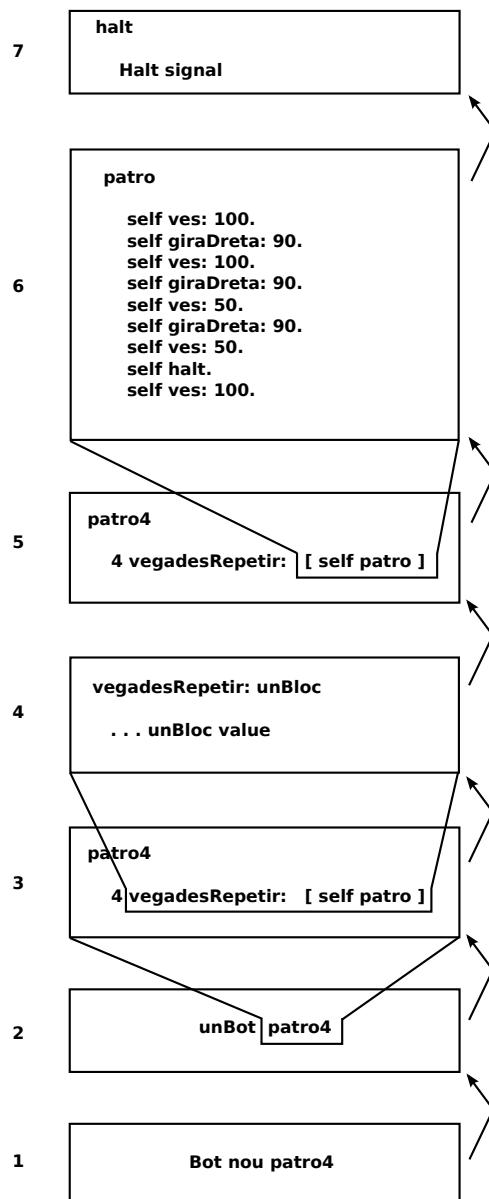
4 vegadesRepetir: [ self patro ]

Després de definir aquests mètodes, si Squeak executa l'expressió Bot nou patro4, diversos missatges són invocats en una reacció en cadena el resultat final de la qual és que el robot dibuixa quatre patrons a la pantalla. Examinem aquesta cadena de missatges: Primer, Bot nou crea un robot nou, al que el missatge patro4 s'envia. Com a resultat, el mètode patro4 és executat. Aquesta execució envia el missatge vegadesRepetir:, el qual envia el missatge patro. L'execució del mètode patro envia diversos missatges, en concret ves: i giraDreta:. En l'argot dels llenguatges de programació, aquesta cadena de missatges s'anomena la pila d'execució: la pila conté tots els mètodes executats com a reacció a un missatge inicial i la cadena de missatges que aquests mètodes han executat, després tots els mètodes executats com a reacció a aquest segon conjunt de missatges i la cadena de missatges que aquests mètodes han executat i així successivament fins que no queden més missatges.

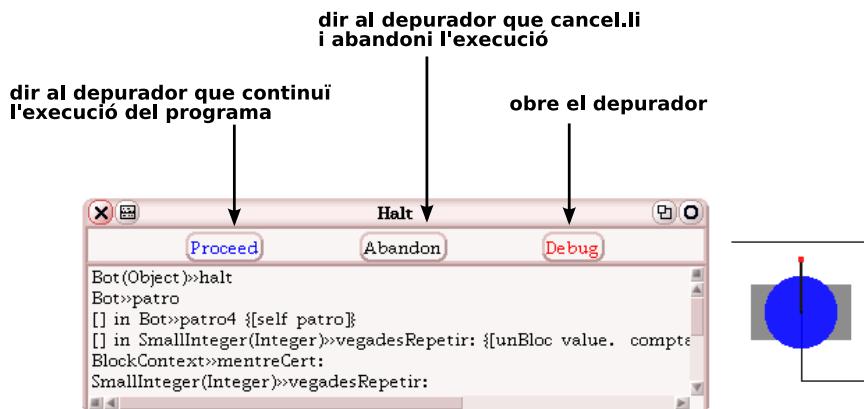
Una representació possible la podeu veure a la figura 15.2, on el darrer mètode executat és a sobre de tot, i així successivament cap avall. Un mètode que crida un altre mètode apareix sota el mètode cridat. La part de cada expressió que porta a la invocació del mètode sobre seu apareix en negreta. Mirem què passa detalladament:

1. A la capsa de baix de tot, el missatge nou és enviat a la classe Bot, que crea un robot nou.
2. El missatge patro4 és enviat al robot creat.
3. L'execució del mètode patro4 envia el missatge vegadesRepetir: [ self patro ]. El missatge vegadesRepetir: està en negreta, ja que és el primer en ser enviat.
4. L'execució del mètode vegadesRepetir: porta a l'execució del bloc. Aquesta es fa mitjançant l'enviament el missatge value a l'argument del missatge vegadesRepetir:.
5. Dins del mètode patro4, el missatge patro és enviat pel mètode vegadesRepetir: com a resultat de l'execució del bloc.
6. El procés continua de manera similar, amb els missatges del mètode patro executant-se un darrera l'altre.

La figura 15.2 conté una altra capsa que explicarem de seguida. El que hauríeu d'entendre ara és que un mètode crida un altre i que el mètode cridat és col·locat a la pila a sobre del mètode que el crida.



**Figura 15.2 — La pila d'execució, que conté tots els missatges i els mètodes invocats com a resultat de l'execució de Bot nou patro4.**



**Figura 15.3** — El robot comença a dibuixar el començament del primer patró, però després de dibuixar quatre línies, s'atura i podeu obrir el depurador.

## Una primera ullada al depurador

La figura 15.2 mostra la manera en la que podeu imaginar la seqüència de missatges enviats i mètodes executats com a resultat de l'execució d'un missatge. El depurador d'Squeak us deixa veure, explorar i canviar la cadena de missatges. El depurador s'invoca automàticament quan un objecte no entén un missatge que li han enviat, com discutirem més tard, però també podeu invocar-lo explícitament amb l'expressió `self halt` dins del cos d'un mètode. Introduir aquesta expressió és útil quan voleu entendre com s'executa una expressió o localitzar un error en un programa.

---

**Important!** El depurador d'Squeak és una eina que us permet explorar una seqüència de mètodes executats. Utilitzant el depurador, podeu conèixer els valors dels arguments dels mètodes, modificar la definició d'un mètode, canviar i veure els valors de variables i arguments, i continuar l'execució.

---

Per obrir el depurador, afegiu l'expressió `self halt` dins del mètode `patro`, com veieu al Mètode 15.3. Després executeu l'expressió `Bot nou patro4`. Hauríeu d'obtenir la situació mostrada a la figura 15.3. Primer es crea un robot nou, després comença a dibuixar el primer patró, i després de dibuixar quatre línies s'atura i apareix una finestra oferint-vos la possibilitat d'obrir el depurador.

**Mètode 15.3** *Introduir l'expressió self halt dins d'un mètode obre una finestra que us ofereix la possibilitat d'obrir el depurador.*

**patro**

"dibuixa un patró"

```
self ves: 100.  
self giraDreta: 90.  
self ves: 100.  
self giraDreta: 90.  
self ves: 50.  
self giraDreta: 90.  
self ves: 50.  
self halt.  
self giraDreta: 90.  
self ves: 100.  
self giraDreta: 90.  
self ves: 25.  
self giraDreta: 90.  
self ves: 25.  
self giraDreta: 90.  
self ves: 50.
```

---

**Important!** Per invocar el depurador, inseriu l'expressió `self halt` dins del cos d'un mètode. L'execució de l'expressió `self halt` obre una finestra oferint-vos la possibilitat d'obrir el depuador.

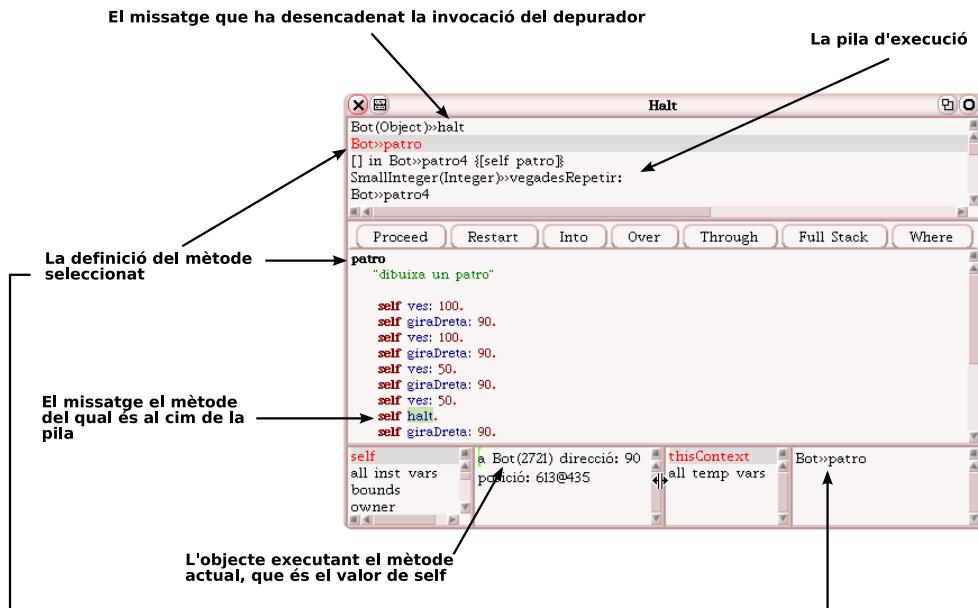
---

La finestra del depurador mostrada a la figura 15.3 ofereix tres botons: **Proceed** (Procedir), **Abandon** (Abandonar), i **Debug** (Depurar).

**Proceed.** Aquest botó li diu al depurador que continuï l'execució del mètode, ignorant el missatge `self halt`. Fixeu-vos que és possible procedir només si heu obert el depurador utilitzant `self halt`. Si teniu un error de debò que obre el depurador, utilitzar **Proceed** no és de cap ajuda, ja que Squeak no pot continuar.

**Abandon.** Aquest botó li diu al depurador que es tanqui, i l'execució del mètode és aturada.

**Debug.** Aquest botó li diu al depurador que s'obri. A la figura 15.4 podeu veure una finestra oberta del depurador.



**Figura 15.4 — La finestra del depurador. El mètode patro està seleccionat.**

Si premeu **Debug** i seleccioneu la segona línia de la subfinestra de dalt de tot, obteniu la finestra mostrada a la figura 15.4. Com podeu veure, el depurador es compon de diverses subfinestres. La subfinestra de dalt de tot representa la pila d'execució, que ja vau veure esquemitzada a la figura 15.2. Tots els missatges que han estat enviats fins l'`halt` són mostrats, amb els missatges més recents al cim. El missatge més recent és, naturalment, `halt`, i per tant és al cim de la pila. Aquest és el missatge que ha desencadenat l'obertura de la capsa de diàleg mostrada a la figura 15.3.

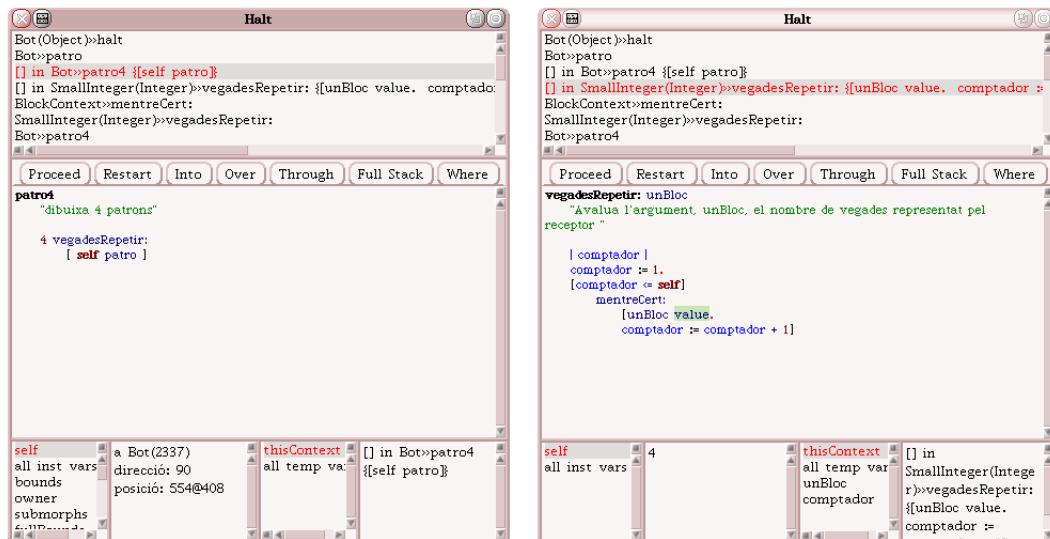
Seleccionar una de les línies de la subfinestra superior fa que la definició del mètode apareixi a la subfinestra del mig. A la figura 15.4 es veu que hem seleccionat a la pila el mètode `patro`, i la seva definició la podeu veure a la segona subfinestra gran. L'objecte que ha rebut el missatge `patro` (que és l'objecte al que la variable `self` es refereix) i que l'ha executat és mostrat a la subfinestra de baix de tot.

Al cos del mètode seleccionat a la pila, que en el nostre exemple és `patro`, el depurador ressalta en verd el mètode l'execució del qual està ara mateix aturada, i que, a la pila, està al damunt del mètode seleccionat. Aquí `self halt` està aturat, i està al damunt de `patro` a la pila. Podeu també fixar-vos que totes les expressions per sobre de `self halt` dins el cos del mètode ja han estat

executades, mentre que les expressions de sota no ho han estat encara. En aquest exemple, self ves: 100. self giraDreta: 90. self ves: 100. self giraDreta: 90. self ves: 50. self giraDreta: 90. self ves: 50. ja han estat executades.

Si seleccioneu la tercera línia de la subfinestra superior, és a dir, el mètode patro4, veureu, com es mostra a l'esquerra de la figura 15.5, que la pila de l'execució del mètode patro4 a dalt, està relacionada amb l'execució de l'expressió self patro del mètode patro4. Ara, si seleccioneu la cinquena línia, veieu un altre cop el mètode patro4, però ara abans que s'executi el bucle vegadesRepetir::.

Si seleccioneu la quarta línia, com podeu veure a la figura 15.5, veieu la definició del mètode vegadesRepetir::. Podeu veure a la subfinestra superior que l'objecte que rep el missatge vegadesRepetir: no és un robot, sinó un nombre enter. En el nostre exemple, el receptor del bucle és l'objecte 4, a l'expressió 4 vegadesRepetir: [ self patro ]. Això significa que dins d'aquest mètode, la variable self està lligada a un objecte enter. Això no és inusual, ja que la variable self *sempre* representa l'objecte que ha rebut el missatge actual.



**Figura 15.5 —** Esquerra: El mètode patro4 s'ha seleccionat. Dreta: El mètode vegadesRepetir: s'ha seleccionat.

## Anar pas a pas per la pila

Amb el depurador d'Squeak no només podeu explorar la pila i identificar els receptors de diversos missatges, sinó que també us deixa executar un mètode pas a pas. Podeu dir-li al depurador de realitzar diferents accions utilitzant els botons que hi ha entre la primera i la segona subfinestra. Aquí teniu una descripció de les més útils, ordenades de dreta a esquerra.

**Proceed.** Prémereu aquest botó té el mateix efecte que prémereu el botó **Proceed** a la capsa de diàleg de la figura 15.3. El depurador es tanca i l'execució del mètode continua si és possible.

**Restart.** Podeu demanar-li al depurador que torni a començar l'execució del mètode actual. Fixeu-vos que de vegades fer això pot portar problemes, ja que podeu estar modificant el mateix objecte dues vegades, obtenint resultats inesperats o fins i tot errors.

**Into.** Aquest botó i el proper són els botons més útils del depurador, i els que s'utilitzen més sovint. Prémereu aquest botó us porta dins del mètode seleccionat sense executar-lo. És a dir, podeu examinar el codi del mètode seleccionat, tal i com apareix a la segona subfinestra del depurador, sense que tingui lloc cap execució del mètode. Veieu la figura 15.6 com a exemple, en què el mètode `giraDreta:` del mètode `patro` s'ha seleccionat i es pot examinar.

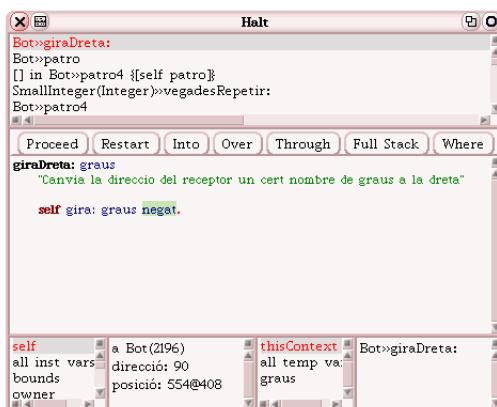
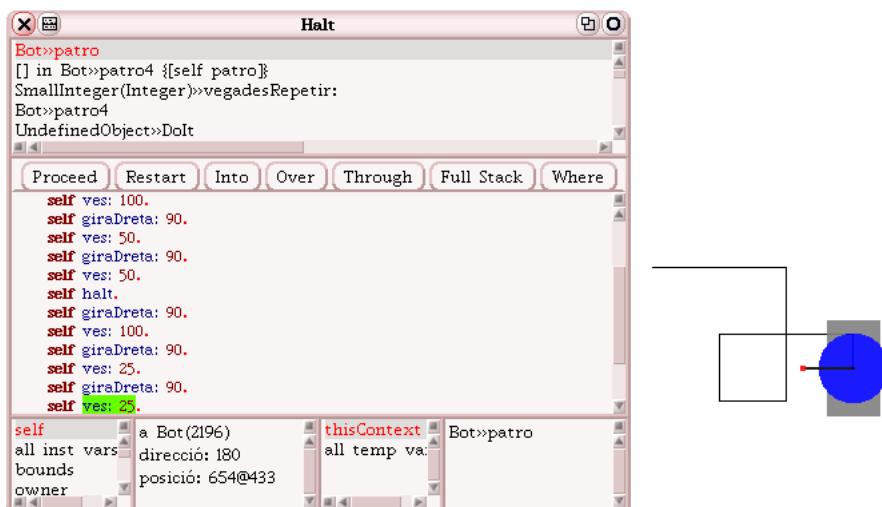


Figura 15.6 — Entrant dins del mètode `giraDreta:`.

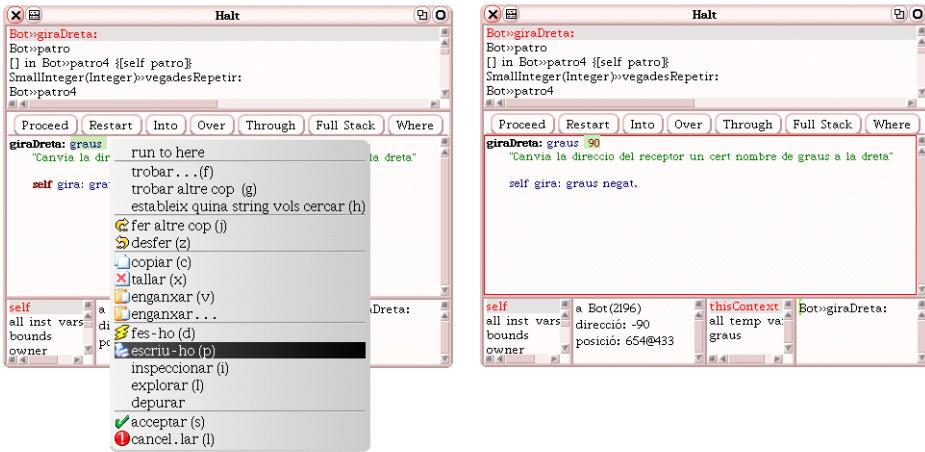
**Over.** Prémereu aquest botó us permet executar el missatge seleccionat sense examinar-lo. Només s'executa l'expressió i s'atura. Veieu la figura 15.7. En aquesta figura hem *examinat*

el mètode patro, i després he *executat* algunes expressions dins del mètode patro. Podeu veure a la figura que ens hem aturat a l'expressió ves: 25. Encara som dins del mètode patro ja que no hem volgut examinar cap de les expressions que componen patro. A mida que es van executant les expressions, hauríeu de veure el robot realitzant cada acció, una darrera l'altra. Fixeu-vos que quan sigueu al final del mètode, havent executat totes les expressions, el botó **Over** us retorna al mètode que ha invocat el mètode actual, i us moveu cap avall a la pila. Per exemple, després d'acabar d'executar patro, tornareu a patro4.

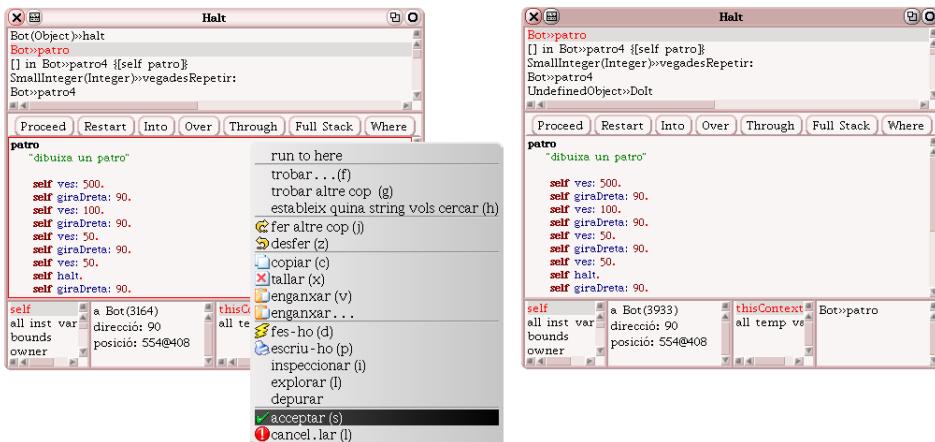


**Figura 15.7 — Examinant el mètode patro i executant cada missatge, podeu veure com el robot executa cada expressió una a una.**

Quan premeu el botó **Into**, esteu demanant al depurador d'*examinar* el mètode sense executar-lo. Per exemple, si l'expressió seleccionada dins el mètode patro és `self giraDreta: 90`, podeu simplement executar l'expressió prement el botó **Over** o podeu prémer el botó **Into**, que li diu al depurador d'anar dins del mètode `giraDreta:`, aturant-se a la primera expressió, com es mostra a la figura 15.6, on el primer missatge a enviar és el missatge negat. Aquí també teniu l'opció d'*examinar* l'expressió negat o executar-la. I un cop més, quan arribeu al final del mètode, retorneu al mètode que crida el mètode on sou ara. També podeu veure els valors dels arguments passats al mètode seleccionant el nom de l'argument al costat del mètode i triant **escriu-ho** (figura 15.8).



**Figura 15.8 —** Esquerra: Seleccionant un argument i escrivint-lo. Dreta: El valor és escrit.



**Figura 15.9 —** Esquerra: Editant la definició del mètode i recomplint-la. Dreta: El mètode patro ha estat recompliat.

Finalment, podeu *modificar* la definició d'un mètode des del depurador editant el codi i acceptant-lo via l'opció del menú contextual **aceptar** (veure figura 15.9). Per exemple, a la figura 15.9, hem tornat a començar el mètode prement el botó **Restart**; després hem editat el mètode en el mateix depurador, substituint el valor 100 del primer ves: pel valor 500. Aleshores hem recompilat el mètode amb l'opció **accept** del menú. Ara el robot es mourà 500 píxels si premem el botó **Proceed**.

Això és una mica complicat, i en veure-ho per primera vegada ho podeu trobar una mica confús. Recordeu que el depurador us deixa executar totes les expressions d'un mètode pas a pas. Així doncs, si us plau, experimenteu amb el depurador i proveu tots els botons. Estigueu tranquil·s, que cap robot prendrà mal en el procés.

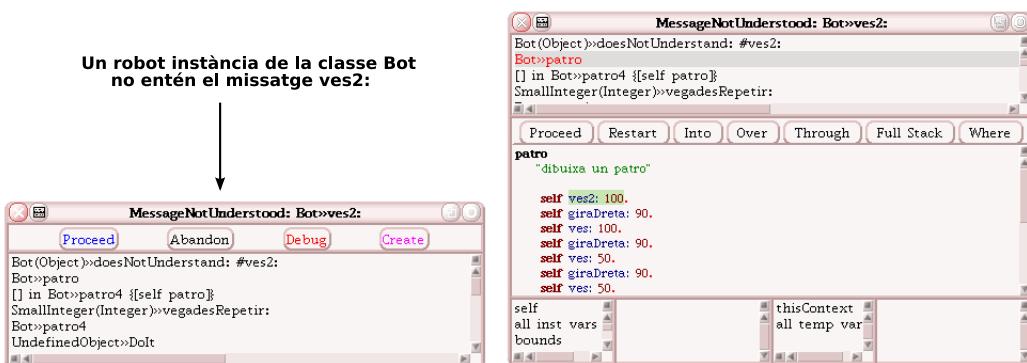
## Corregir els errors

Us hem mostrat com obrir el depurador inserint l'expressió `self halt` en un mètode. També us hem ensenyat com podeu editar el codi d'un mètode en el depurador. Aquesta tècnica us permet utilitzar el depurador per corregir els vostres errors, és a dir, *depurar* el vostre codi. Quan un objecte rep un missatge que no entén, Squeak us dóna l'oportunitat d'obrir el depurador. Quan un objecte no entén un missatge, Squeak envia a aquest objecte el missatge `doesNotUnderstand:` (això vol dir “no entén” literalment) juntament amb una representació del missatge. Per defecte, el mètode `doesNotUnderstand:` obre una capsà de diàleg per preguntar-vos si voleu obrir el depurador. Aleshores podeu utilitzar el depurador per explorar la pila dels mètodes executats i intentar entendre què ha anat malament.

### Exemple 1

Per il·lustrar el procés, canvieu la primera línia del mètode `patro` a `self ves2: 100` i executeu l'expressió `Bot nou patro4`. Hauria d'aparèixer la capsà de diàleg del depurador (figura 15.10). La capsà de diàleg del depurador indica que el receptor, un robot creat per la classe `Bot`, no entén el missatge `ves2::`. Quan premeu el botó **Debug** a la capsà de diàleg, obriu el depurador, com podeu veure a l'esquerra de la figura 15.10.

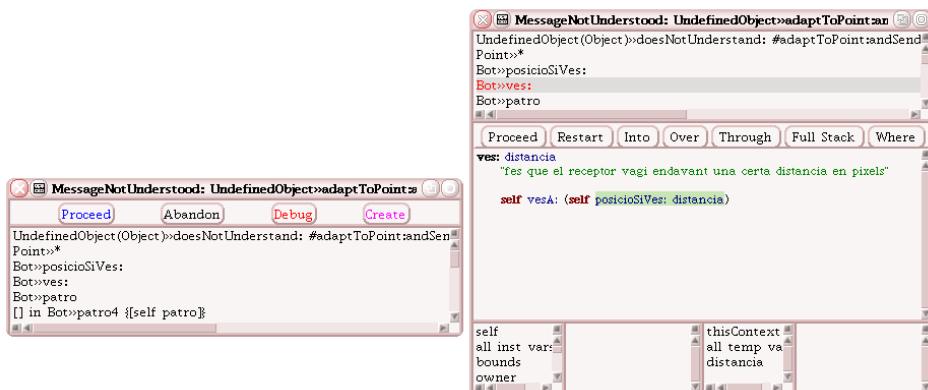
El mètode al cim de la pila és el mètode `doesNotUnderstand:`, que és enviat al receptor d'un missatge quan el receptor no entén aquest missatge. El mètode directament davall és el mètode que conté el missatge que ha desencadenat l'error i per tant la crida al missatge `doesNotUnderstand::`. Aquí el mètode `patro` conté el missatge `ves2::`, que no és entès pel robot, com veieu a la figura 15.10.



**Figura 15.10** — Esquerra: *Un error doesNotUnderstand: ha succeït.* Dreta: *Trobar el problema.*

## Exemple 2

Ara canviem la primera línia del mètode patro a self ves: nil i executeu l'expressió Bot nou patro4. Hauríeu d'obtenir la capsa de diàleg del depurador mostrada a la figura 15.11. L'error és una mica difícil de trobar. Aquí, el títol de la capsa de diàleg MessageNotUnderstood: UndefinedObject indica que hi ha un missatge no entès enviat a nil, que és una instància de la classe UndefinedObject.



**Figura 15.11** — Esquerra: *nil rep un missatge que no entén.* Dreta: *Anar cap avall per la pila d'execució.*

El fet que nil sigui passat com a valor ha desencadenat un error després que s'executessin altres mètodes. Així doncs, heu d'anar avall a la pila, al punt on podeu entendre i corregir el vostre error. Per exemple, a la dreta de la figura 15.11, el segon mètode des de dalt deixa clar que alguna cosa no ha anat bé amb \*, però el problema no ve d'aquest mètode. El mateix passa amb el següent mètode ves:. Si seleccioneu aquest mètode, podeu veure que el mètode ves: només passa l'argument que rep del mètode patro al mètode posicioSiVes:. Si seleccioneu el paràmetre distància i escriviu el seu valor, obtenuïs nil. Això indica que l'error ve de més avall de la pila.

Finalment, podeu veure a l'esquerra de la figura 15.12 que nil es passa com a argument, i no un nombre, que és el que espera el mètode ves:. Ara podeu corregir l'error editant el mètode patro i canviar self ves: nil per self ves: 100, acceptant els canvis mitjançant el menú i prement el botó **Proceed** per continuar l'execució.

Aquest exemple utilitzant nil com a argument d'un missatge ves: provoca un error fàcil d'identificar. És possible que cometeu ocasionalment errors triviais com aquest, però el que generalment haureu d'afrontar és una gran varietat de situacions inesperades que provoquen errors. Tot i així, el procés és el mateix: Heu d'utilitzar el depurador per explorar la pila d'execució, i comprovar els valors dels arguments fins que pugueu entendre què és el que ha anat malament. Us podeu trobar amb la situació que també calgui modificar el codi, no només els arguments dels missatges.



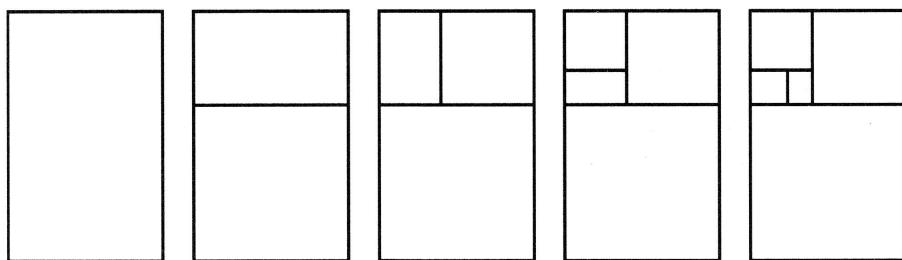
**Figura 15.12 —** Esquerra: *Editar i recompilar la definició del mètode.* Dreta: *El mètode patro ha estat recompliat.*

## Resum

- A Smalltalk, el valor d'una variable no inicialitzada és per defecte l'objecte nil, que s'utilitza per representar un valor indefinit.
- El depurador d'Squeak és una eina que us permet explorar els mètodes ja executats. Utilitzant el depurador, podeu escriure els valors dels arguments, modificar la definició dels mètodes, i continuar l'execució.
- Per obrir el depurador explícitament, inseriu l'expressió self halt en un mètode. Quan l'expressió self halt s'executa, podeu obrir el depurador.

## Capítol 16

# Descompondre per recompondre



En aquest capítol voldríem ensenyar-vos que una bona aproximació per resoldre un problema és *descompondre* el problema en petits problemes més senzills de resoldre, i després *compondre* les solucions dels problemes més petits per resoldre el problema original. Sovint, definir i utilitzar un cert nombre de mètodes simples us permet resoldre un problema complex d'una manera molt més senzilla i natural que intentar resoldre el problema tot de cop. Aquest és un altre exemple del poder de l'abstracció que vam introduir al capítol 12.

Desafortunadament, tot i que la decomposició de problemes en problemes més petits és una tècnica molt valiosa, no hi ha cap manera sistemàtica de descompondre un problema. Només amb experiència i molta prova i error és possible aprendre i desenvolupar alguna intuïció sobre la resolució de problemes. En aquest capítol, us proposo alguns problemes, petits però no triviais, que hauríeu de provar de resoldre. Intenteu descompondre'ls en problemes més petits i amb les solucions d'aquests compondre una solució al problema plantejat en un principi. Els problemes plantejats en aquest capítol són els exercicis més complexos de tot el llibre, així que no us desanimeu si no els resoleu a la primera. Tot i així, és molt important que ho intenteu.

## Laberints i espirals

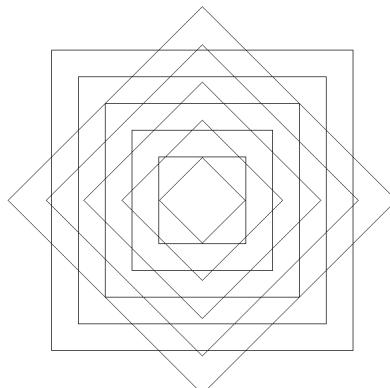
Al capítol 10, vau fer experiments amb repeticions i variables. En aquell moment, no sabíeu que podíeu utilitzar mètodes escrits per vosaltres mateixos. Ara us suggerim que torneu a fer aquells experiments utilitzant mètodes com quadrat: i que definiu mètodes amb múltiples arguments per ajudar-vos. Com ja vam mencionar, resoldre problemes complexos utilitzant mètodes més senzills és una tasca difícil que només es pot aprendre amb la pràctica. Per tant, us encoratgem a fer tots els experiments d'aquest capítol, i, de fet, hauríeu de provar de trobar més d'una solució per a cada un d'ells. Intenteu veure com la solució a un problema pot ser expressada a un cert nivell elevat d'abstracció. Aquesta aproximació fa que el problema global sigui més senzill d'expressar. Per exemple, per dibuixar una piràmide de quadrats, podríeu simplement dibuixar un gran nombre de quadrats. Però a un nivell més alt d'abstracció podríeu veure la piràmide com un cert nombre de fileres de diferent mida amb quadrats com a components, i provar de definir i utilitzar un mètode que dibuixi fileres de  $n$  quadrats.

### Quadrats centrats

Els experiments següents us proporcionen més oportunitats de practicar les vostres habilitats. Imagineu diferents estratègies per resoldre aquests problemes. Una pista: definiu un mètode quadratCentrat: mida que dibuixi un quadrat central al voltant del robot, i que després de dibuixar el quadrat faci que el robot es desplaci a la seva posició original.

#### Experiment 16-1 (ones quadrades en un estany quadrat)

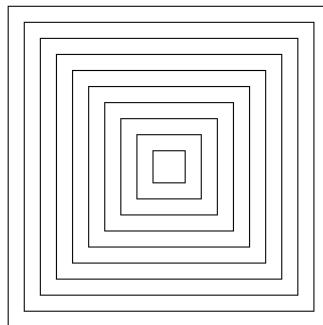
Mitjançant la composició de mètodes, dibuixe la figura que veieu més avall.



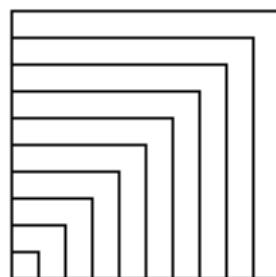
---

**Experiment 16-2 (un passadís)**

Mitjançant la composició de mètodes, dibuixe la figura que veieu més avall.

**Experiment 16-3 (quadrats russos)**

Utilitzant el mètode quadrat: per construir quadrats de diferents mides, com podeu veure a la figura.



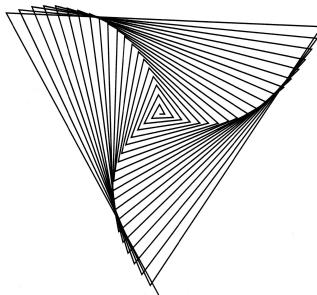
## Espirals

Les espirals d'aquest capítol s'han dibuixat totes utilitzant només segments rectes, angles i repetitions. No tenen cap corba; el robot dibuixa una línia recta a mida que va endavant, gira un cert angle i repeteix el procés.

Al primer grup d'espirals, la longitud de cada segment canvia una mica respecte de la del segment previ. El robot gira el mateix angle després de cada segment recte. Aquestes espirals s'anomenen *espirals d'angle constant* ja que l'angle no canvia.

### Experiment 16-4 (una espiral d'angle constant)

Definiu un mètode `espiralAngleConstant`: que dibuixi una espiral d'angle constant. L'angle que el robot ha de girar és l'únic paràmetre. Canviieu la distància que el robot viatja en la mateixa quantitat per a cada segment. Divertiu-vos i proveu diferents angles; podeu crear dibuixos molt bonics. L'*script* següent ha creat l'espiral que veieu aquí.



```
| pica |
pica := Bot nou.
pica espiralAngleConstant: 121
```

---

### Experiment 16-5 (una altra espiral)

En l'experiment anterior, a cada pas afegíeu el mateix nombre a la distància a què el robot es desplaçava. Ara experimenteu calculant la mida d'un costat, no afegint una constant a la longitud del costat previ, sinó multiplicant-la per una proporció constant. Pareu atenció! Multiplicar per 1.1 implica un increment d'un 10 per cent.

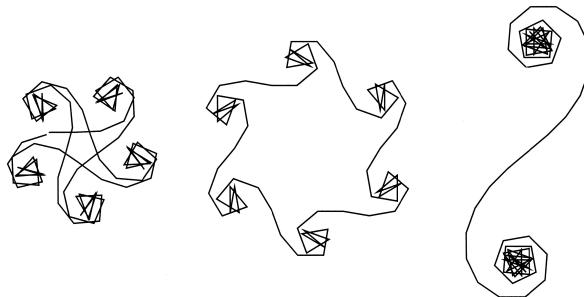
---

### Experiment 16-6 (una espiral amb quatre paràmetres)

El mètode `espiralAngleConstant`: depèn de quatre valors. Aquests són el *nombre de segments*, la *longitud inicial* del primer segment, la *quantitat afegida a cada segment*, i l'*angle* que gira el robot. Definiu un mètode `espiralSegments:longitudInicial:incrementLongitud:angleConstant:` que dibuixi qualsevol espiral canviant aquests quatre paràmetres. Proveu, per exemple, l'*script* següent, que dibuixa dues espirals diferents.

```
| pica |
pica := Bot nou.
pica espiralSegments: 50 longitudInicial: 10 incrementLongitud: 3 angleConstant: 144.
pica color: Color vermell.
pica espiralSegments: 120 longitudInicial: 1 incrementLongitud: 3 angleConstant: 12.
```

---



**Figura 16.1** — Esquerra: *Iteracions: 90, angle inicial: 2, increment: 20, longitud: 30.* Mig: *Iteracions: 72, angle inicial: 40, increment: 30, longitud: 30.* Dreta: *Iteracions: 100, angle inicial: 40, increment: 5, longitud: 23.*

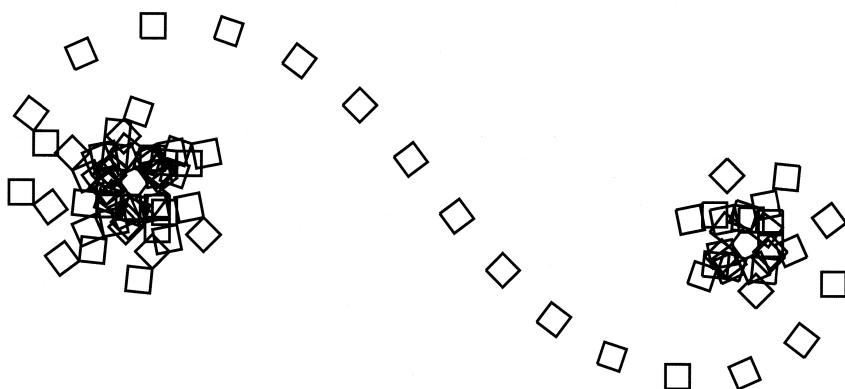
### Experiment 16-7 (espirals amb distància constant)

Fins ara heu creat espirals *canviant la distància* que el robot es movia endavant, i el robot sempre girava *el mateix angle*. Ara experimenteu fent el contrari: manteniu la distància constant, i incrementeu l'angle una quantitat fixada després de cada segment. Per a aquests experiments, definiu un mètode amb quatre arguments: el nombre de repeticions, el valor inicial de l'angle, l'increment de l'angle, i la longitud d'un segment. Com podeu veure a la figura 16.1, que mostra tres espirals, fer prediccions de les corbes que aquest mètode pot generar és més aviat difícil. Juguieu amb diferents valors. Divertiu-vos!

---

### Experiment 16-8 (espirals a partir de quadrats)

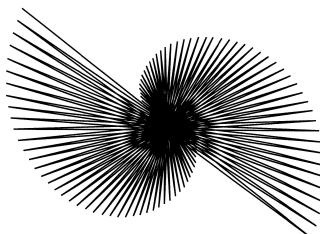
Si canvieu el mètode que heu definit a l'Experiment 16-7 per a que dibuixi un petit quadrat en lloc d'una línia, podeu aconseguir figures molt extravagants, com la que veieu a la figura 16.2.



**Figura 16.2** — Dibuix d'una “espiral” de longitud constant utilitzant quadrats.

### Experiment 16-9 (una espiral a partir de línies)

Intenteu reproduir la figura de més avall. Algunes indicacions: comenceu amb una longitud de 5 i incrementeu-la de 3 en 3, i feu que el robot giri un angle de 178 graus.



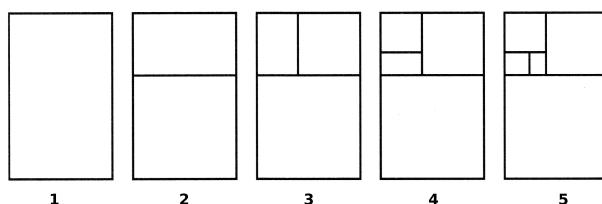
## Rectangles d'or

Ara és el moment d'escriure alguns mètodes per dibuixar el famós rectangle d'or que vam introduir al capítol 8. Us encoratgem a provar els vostres propis mètodes abans de llegir cap solució. Per a alguns dels experiments no donem cap solució. En lloc d'això, us oferim algunes pistes.

Un rectangle d'or és un rectangle amb la propietat que si traieu un quadrat d'un extrem del rectangle, amb costat de longitud igual a l'amplada del rectangle, allò que queda és també un rectangle d'or. Per exemple, el rectangle 2 de la figura 16.3 està compost d'un quadrat a la part de sota i un altre rectangle d'or a la part de dalt. Al rectangle 3, el rectangle d'or dins el rectangle 2 ha estat dividit en un quadrat i un rectangle d'or més petit encara. El procés es repeteix en els rectangles 4 i 5.

No és difícil demostrar que per tal que un rectangle tingui aquesta propietat, la proporció entre llargada i amplada ha de ser  $\frac{1+\sqrt{5}}{2}$  (que és aproximadament igual a 1.6)<sup>1</sup>. Aquest nombre especial s'anomena la *raó àuria* o la *secció àuria*. Podem calcular-lo a Smalltalk amb l'expressió  $1 + 5 \text{ sqrt} / 2$ .

El problema que us estem plantejant és dibuixar diversos rectangles d'or, cada un dins de l'anterior, com podeu veure a la figura 16.3.



**Figura 16.3 —** Els primers cinc passos en la construcció dels rectangles d'or.

Proveu d'identificar accions que us puguin ajudar a aconseguir el vostre objectiu. Preneu, si cal, un tros de paper i dibuixe un rectangle d'or de 10 cm d'amplada amb un o dos rectangles d'or a dins, com els de la figura 16.3. Per exemple, podríeu definir primer un mètode `rectangleDaurat`: que dibuixa un rectangle d'or. Després podríeu definir un mètode `moltsRectanglesDaurats`: que dibuixi diversos rectangles d'or cridant el mètode anterior.

Una altra aproximació és dibuixar un rectangle d'or i traçar una línia dins del rectangle per

<sup>1</sup>Si l'amplada d'un rectangle d'or és 1 i la seva llargada és  $x$ , quan traieu un quadrat del rectangle d'or, el rectangle d'or més petit que queda té llargada 1 i amplada  $x - 1$ . Com que aquests rectangles són similars, la proporció entre els seus costats ha de ser la mateixa. Traient els denominadors i resolent l'equació quadràtica resultant posem de manifest que la llargada  $x$  del rectangle original ha de ser igual a la raó àuria.

crear el proper rectangle d'or, i així successivament. No us amoïneu si no us en sortiu a la primera i heu de fer una mica de prova i error, aquest exercici és complicat.

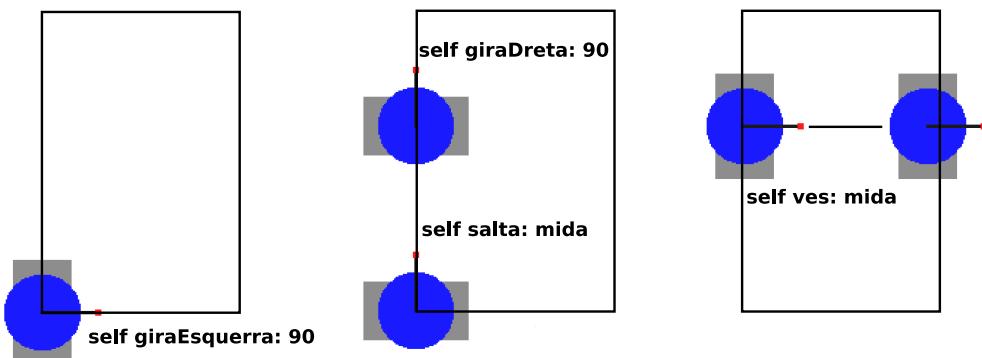
Per ajudar-vos a començar, fixeu-vos en l'*Script 16.1*, que dibuixa un rectangle d'or. Comenceu convertint aquest *script* en un mètode que té com a paràmetre l'amplada del rectangle.

**Script 16.1** *Pica dibuixa un rectangle d'or.*

```
| pica longitud amplada |
pica := Bot nou.
amplada := 100.
longitud := amplada * ((1 + 5 sqrt) / 2).
2 vegadesRepetir:
  [ pica ves: amplada ;
    giraEsquerra: 90 ;
    ves: longitud ;
    giraEsquerra: 90. ]
```

### Una solució d'un-rectangle-per-línia

Aquí teniu una solució basada en el fet que només necessiteu dibuixar completament el rectangle més gran. Després d'això podeu dibuixar una sola línia en el lloc adequat per crear el proper rectangle d'or més petit. Aquest procés es mostra a la figura 16.4.



**Figura 16.4** — Un cop un robot ha dibuixat un rectangle d'or, es col·loca en la posició adequada i dibuixa una línia que talla el rectangle per formar un quadrat i un rectangle d'or més petit.

Per tant, per dibuixar diversos rectangles d'or us cal saber com (1) dibuixar un rectangle d'or sencer, (2) dibuixar una línia que defineixi el quadrat dins del rectangle d'or, i (3) calcular l'amplada del proper rectangle d'or. Així, tenim tres tasques:

1. Definir un mètode que donada una amplada, sap com calcular la llargada corresponent i dibuixa un rectangle d'or, com hem suggerit que féssiu a la secció anterior. Anomenem aquest mètode rectangleDaurat: amplada, on el paràmetre amplada representa l'amplada del rectangle d'or.
2. Definir un mètode que dibuixa la línia per delimitar el quadrat inclòs dins del rectangle, tal com ho mostra el segon rectangle de la figura 16.3. Anomenem aquest mètode liniaDivisoria: llargada; primer mou el robot una distància llargada des de la seva posició de partida, i després dibuixa un segment de longitud llargada en la direcció inicial del robot. Fixeu-vos que aquestes dues distàncies són la mateixa ja que són dos costats d'un quadrat.
3. Un cop els mètodes rectangleDaurat: i liniaDivisoria: han estat definits, us caldrà un mètode que ho combini tot: Hauria de dibuixar un rectangle d'or, i aleshores, repetidament, dibuixar el costat que falta del quadrat per formar el proper rectangle d'or més petit i calcular la mida del proper quadrat, que també és l'amplada del proper rectangle d'or. També necessitareu assegurar-vos que el robot parteix del lloc adequat, i que apunta en la direcció correcta en el moment de dibuixar el proper quadrat. Anomenem aquest mètode rectanglesDaurats: ampleMaxim aNivell: n, on ampleMaxim representa l'amplada del primer rectangle d'or i n indica el nombre de quadrats que es dibuixaran (de manera que al final hi haurà  $n + 1$  retangles d'or).

Anem a desenvolupar els tres mètodes.

### El Mètode rectangleDaurat: amplada

El Mètode 16.1 no té res d'especial; donada l'amplada d'un rectangle d'or, calcula la llargada i dibuixa el rectangle.

**Mètode 16.1** *Dibuixa un rectangle d'or, donada la seva amplada.*

rectangleDaurat: amplada

"Dibuixa un rectangle la llargada del qual és la raó àuria multiplicada per la seva amplada"

```
| llargada |
llargada := amplada * (1 + 5 sqrt / 2).
2 vegadesRepetir:
[ self ves: amplada ;
  giraEsquerra: 90 ;
```

```

    ves: llargada ;
    giraEsquerra: 90. ]

```

El mètode rectangleDaurat: comença dibuixant en la direcció actual del robot i dibuixa primer l'amplada del rectangle. Quan acaba, el robot torna al punt de partida apuntant en la direcció original.

### El Mètode liniaDivisoria: llargada

La definició del mètode liniaDivisoria: parteix de la base que el robot és en una cantonada i apuntant cap a l'amplada del rectangle, amb la llargada del rectangle a la seva esquerra. Aquest escenari coincideix amb la situació al final del mètode rectangleDaurat:. La figura 16.4 mostra l'acció de liniaDivisoria:, on el robot gira noranta graus a l'esquerra, es desplaça la llargada del rectangle, gira noranta graus a la dreta, i dibuixa el costat que li falta al quadrat per crear el nou rectangle d'or.

A la situació de la figura 16.4, on el robot és a la base del rectangle d'or i apunta a la dreta (primer dibuix de la figura), acaba dibuixant un segment horitzontal de mida llargada a una distància llargada de la seva posició inicial (tercer dibuix de la figura). Aquesta explicació és de fet més complicada que la definició del mètode.

**Mètode 16.2** *Dibuixa una línia que divideix un rectangle d'or en un quadrat i un rectangle d'or més petit.*

liniaDivisoria: llargada

"Mou la distància llargada a l'esquerra de la posició inicial,  
i dibuixa un segment de mida llargada en la direcció inicial"

```

self giraEsquerra: 90 ;
salta: llargada ;
giraDreta: 90 ;
ves: llargada

```

### El Mètode rectanglesDaurats: ampleMaxim aNivell: n

Aquest mètode, Mètode 16.3, és una mica més complicat. Proveu d'entendre'l vosaltres mateixos abans de llegir-ne l'explicació.

### Mètode 16.3

```
rectanglesDaurats: ampleMaxim aNivell: n
    "Dibuixa n + 1 rectangles d'or; el més gran té amplada ampleMaxim"
(1)    | ampleActual |
(2)    ampleActual := ampleMaxim.
(3)    self rectangleDaurat: ampleActual.
(4)    n vegadesRepetir: [ self líniaDivisoria: ampleActual.
(5)        self giraEsquerra: 90.
(6)        ampleActual := ampleActual * ((1 + 5 sqrt) / 2 - 1) ]
```

Aquest mètode genera un grup de rectangles d'or un dins de l'altre el primer dels quals té ampleMaxim com a amplada. Per crear sis rectangles d'or, executeu l'expressió Bot nou rectanglesDaurats: 100 aNivell: 5. Això és el que passa:

- La variable ampleActual representa l'amplada del rectangle actual. També representa la mida del segment del quadrat que queda per dibuixar.
- La línia (2) inicialitza el valor de la variable ampleActual a l'amplada del primer rectangle d'or. Aquesta amplada es dóna com a argument del mètode. Aquesta és l'amplada del primer rectangle d'or.
- La línia (3) dibuixa el primer rectangle d'or, que serveix de base a la resta de rectangles.
- La línia (4) defineix un bucle que repeteix les mateixes accions n vegades.
  - La línia (4) primer dibuixa la part que falta del quadrat utilitzant el mètode líniaDivisoria:. La mida d'aquest quadrat és l'amplada del rectangle d'or actual. Per tant, el mètode utilitza la variable ampleActual com a argument. Fixeu-vos que utilitzar la variable ampleMaxim no funcionaria per als rectangles petits, ja que el seu valor no canvia dins del cos del bucle.
  - La línia (5) posiciona el robot de manera que pugui dibuixar el proper segment. La nostra hipòtesi de partida és que el rectangle sempre es dibuixa amb la seva amplada en la direcció apuntada pel robot. Així, l'expressió self giraEsquerra:90 apunta el robot cap a l'amplada del nou rectangle (per exemple, després del darrer dibuix de la figura 16.4, el robot girarà de l'est al nord).
  - La darrera línia sorgeix de l'observació que la llargada d'un rectangle d'or és l'amplada del rectangle anterior. Per tant, l'amplada d'un nou rectangle és la diferència entre la llargada i l'amplada del rectangle previ. Expressant-ho matemàticament, tenim

$$\text{novaAmplada} = \text{llargadaRectangleActual} - \text{ampladaRectangleActual}$$

A partir de la fórmula de la raó àuria,

$$\text{llargadaRectangleActual} = \frac{1+\sqrt{5}}{2} \times \text{ampladaRectangleActual}$$

Per tant,

$$\text{novaAmplada} = \text{ampladaRectangleActual} \times \left( \frac{1+\sqrt{5}}{2} - 1 \right)$$

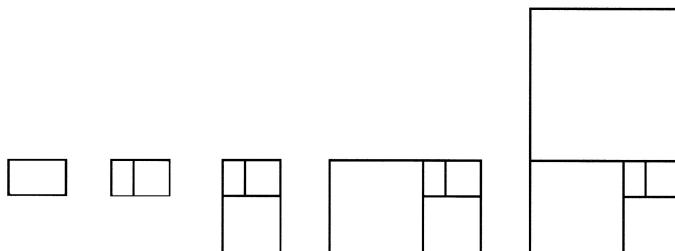
A Smalltalk escrivim això com: `ampleActual := ampleActual * ((1 + 5 sqrt) / 2 - 1).`

Els problemes de dibuixar un rectangle d'or i dibuixar la línia divisòria són força senzills. La composició de les solucions d'aquests dos problemes per obtenir la solució desitjada requereix parar atenció a les suposicions que fem per tal que cada mètode funcioni correctament, és a dir, el context pel qual ha estat definit.

Potser voleu provar de trobar una altra solució a aquest problema. Podeu utilitzar la mateixa tècnica amb l'Experiment 16-10.

### Experiment 16-10 (rectangles d'or incrementals)

Heu vist un mètode per dibuixar un conjunt de rectangles d'or *decreixents*. Ara feu un mètode per crear un conjunt de rectangles d'or creixents. Les figures d'aquí sota us ensenyen alguns possibles passos.



## Enrajolar

Ara ens agradaria que intentéssiu reproduir algunes figures utilitzant el mètode quadrat:. Aquests experiments són importants per desenvolupar i entendre la composició de mètodes i la reutilització de les abstraccions que representen. La idea que intentem promoure amb aquests exercicis és fonamental per entendre com es construeixen els programes complexos.

Us donarem algunes pautes: Primer de tot, hi ha diverses maneres d'aproximar-se als problemes. En general, proveu de veure com simplificar-los. Si mireu les tres figures dels tres experiments següents, podeu imaginar-vos que seria força convenient disposar d'un mètode anomenat, diguem, filaDeQuadrats: que pogués dibuixar una fila d'un nombre arbitrari de quadrats. Suposant que tinguéssiu aquesta solució, proveu de donar un esquema de solució per a cada un dels problemes. Després, implementeu el mètode i comproveu si us ha ajudat. Descompondre un problema és difícil, i l'única manera d'aprendre'n és inventant i provant diferents idees, avaluant el resultat, millorant la idea, i provant-ho altre cop.

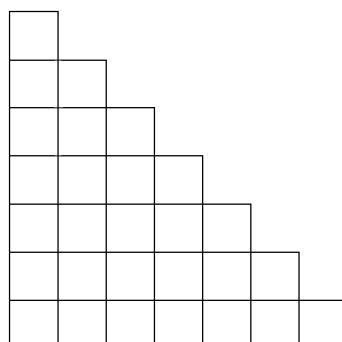
Com a alternativa, també podeu utilitzar bucles dins d'altres bucles:

```
n vegadesRepetir: [  
    m vegadesRepetir: [ . . . ] ]
```

També us suggerim que definiu els vostres mètodes de manera que acabin col·locant el robot al punt de partida. Això fa més senzill compondre els mètodes. Experimenteu amb totes aquestes aproximacions.

### Experiment 16-11 (Una Piràmide Rectangular)

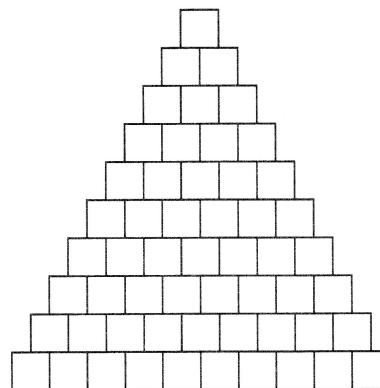
Utilitzant el mètode quadrat:, que dibuixa un quadrat d'una mida determinada, filaDeQuadrats:, que dibuixa una fila de quadrats, i altres mètodes que pugueu definir, dibuixe la piràmide que veieu aquí a sota.



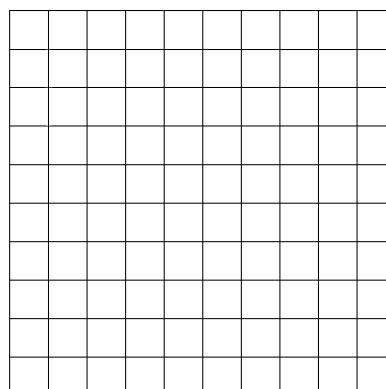
---

**Experiment 16-12 (una piràmide triangular)**

Utilitzant el mètode `quadrat()`, que dibuixa un quadrat d'una mida determinada, `filaDeQuadrats()`, que dibuixa una fila de quadrats, i altres mètodes que pugueu definir, dibuixeu la piràmide que veieu aquí a sota.

**Experiment 16-13 (tauler de dames)**

Utilitzant el mètode `quadrat()`, que dibuixa un quadrat d'una mida determinada, `filaDeQuadrats()`, que dibuixa una fila de quadrats, i altres mètodes que pugueu definir, dibuixeu el tauler que veieu aquí a sota.



## Resum

- Per resoldre un problema gran, descomponeu-lo en problemes més petits; aleshores torneu a compondre la solució als problemes més petits per arribar a la solució del problema gran.
- Pareu atenció al context en què els problemes petits es resolen, ja que caldrà assegurar-vos que aquest context no us passa per alt en el moment de compondre les solucions.



## Capítol 17

# Cadenes, i eines per entendre programes

En aquest capítol us presentem un concepte important: la noció de cadena de caràcters<sup>1</sup>. Una cadena és una seqüència de caràcters que podem utilitzar per representar paraules o frases. Una utilització destacada de les cadenes és la interacció amb l'usuari. En propers capítols, aprendreu a utilitzar les cadenes com a eina per entendre estructures més complexes, com ara condicions i bucles condicionals. En aquest capítol us presentarem només les característiques més importants de les cadenes, la informació bàsica que utilitzareu després en propers capítols. També us ensenyarem com utilitzar cadenes per ajudar-vos a entendre l'execució dels programes. Finalment us recomanem que proveu d'utilitzar el depurador com a ajut per a la comprensió dels experiments proposats en aquest capítol.

### Cadenes

Les cadenes s'utilitzen per representar informació i mostrar-la a l'usuari. Les cadenes estan delimitades per cometes simples ('Això és una cadena de caràcters'), i, com podeu veure, poden contenir espais. Per exemple, la cadena 'Squeak és interessant' representa una seqüència de vint-i-un caràcters: S q u e ... Fixeu-vos que l'espai és també un caràcter. Una cadena pot contenir qualsevol nombre de caràcters, fins i tot zero. Per tant, " és una cadena buida, 'a' és una cadena amb un sol caràcter a, i ' ' és una cadena amb un sol caràcter espai.

---

<sup>1</sup>Nota del Traductor: En anglès s'anomenen *Strings*, classe d'Smalltalk que, com ja hem dit, no s'ha traduït. A més, per no utilitzar l'expressió composta "cadenes de caràcters" tota l'estona, i seguint els consells de SoftCatalà, les anomenarem "cadenes" d'ara en endavant.

---

**Important!** Una cadena és una seqüència de caràcters delimitada per cometes simples. Una cadena representa informació textual, com paraules o frases. Les cadenes es poden utilitzar per mostrar informació a la pantalla.

---

Seleccionant una cadena i triant **escriu-ho** al menú, s'escriu la cadena. Les cadenes tenen associades diversos mètodes, però el més important dins del context d'aquest llibre és el mètode , amb el caràcter coma com a nom. Quan s'envia el missatge , a una cadena com a receptor amb una cadena com a argument, el mètode , retorna una cadena resultat de concatenar el receptor amb l'argument. És a dir, retorna una cadena els caràcters de la qual són els de la primera cadena seguits dels caràcters de la segona cadena.

'squeak' "el valor d'una cadena és la cadena mateixa"

– *Escriure el valor retornat: 'squeak'*

'a' "el valor d'una cadena és la cadena mateixa"

" "una cadena buida"

'squeak' , 'és interessant' "concatenar dues cadenes"

– *Escriure el valor retornat: 'squeak és interessant'*

'squeak' , ' ' , 'és interessant' "concatenar múltiples cadenes"

– *Escriure el valor retornat: 'squeak és interessant'*

El mètode copyReplaceAll:with: us permet modificar una cadena substituint totes les aparicions d'una subcadena particular per una cadena diferent. A l'exemple de sota, 'no' és substituït per 'realment':

'Squeak no és interessant' copyReplaceAll: 'no' with: 'realment'

– *Escriure el valor retornat: 'Squeak realment és interessant'*

## La comunicació amb l'usuari

Squeak ofereix eines per mostrar informació per pantalla i per demanar informació a l'usuari. La classe PopUpMenu permet fer aparèixer un menú i mostrar informació a l'usuari utilitzant cadenes. Per exemple, l'expressió PopUpMenu inform: 'squeak és interessant' fa que aparegui una petita

finestra mostrant el missatge 'squeak és interessant' que s'espera a que l'usuari premi el botó **ok**. La classe `FillInTheBlank` permet demanar dades a l'usuari. Per exemple, l'expressió `FillInTheBlank request: 'és interessant squeak?'` fa aparèixer una capsa de diàleg amb lloc per introduir text i espera que l'usuari escrigui algun text i premi el botó **Acceptar** o el botó **Cancelar**. El resultat d'aquesta expressió és una cadena que representa el que ha estat escrit per l'usuari. Podeu especificar també una resposta per defecte de l'usuari utilitzant el missatge `request:initialAnswer:` com podeu veure a l'*script* següent, i a la figura 17.1.

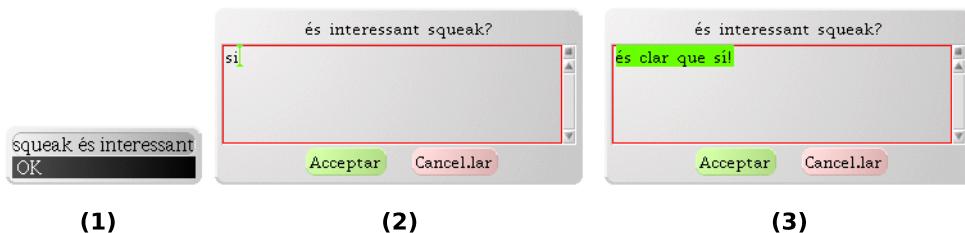
(1) `PopUpMenu inform: 'squeak és interessant'`

(2) `FillInTheBlank request: 'és interessant squeak?'`

– *Escriure el valor retornat: 'si'*

(2) `FillInTheBlank request: 'és interessant squeak?' initialAnswer: 'és clar que sí!'`

– *Escriure el valor retornat: 'és clar que sí!'*



**Figura 17.1 — Finestres de missatges i d'entrada de dades.**

## Cadenes i caràcters

Una cadena està composta de caràcters. Així com una cadena es col·loca entre cometes simples per deixar clar que és una cadena, els caràcters individuals són prefixats pel signe del dòlar \$ per indicar que són caràcters. Per exemple, \$a és el caràcter representant la lletra "a". Fixeu-vos que, encara que els caràcters individuals són prefixats per \$, quan s'edita una cadena els caràcters s'escriuen sense el signe del dòlar.

Hi ha diverses maneres d'accedir als caràcters individuals d'una cadena. Per exemple, els mètodes `first`, `second` i `third` retornen el primer, segon o tercer caràcter d'una cadena. El mètode

size retorna el nombre de caràcters de la cadena, mentre que el mètode at: unNombre retorna el caràcter present a una posició determinada de la cadena. Podeu substituir el caràcter a una posició específica de la cadena (unNombre) per un altre caràcter utilitzant at: unNombre put: unCaracter. El mètode copyUpTo: unCaracter retorna una cadena composta de tots els caràcters de la cadena receptora des del començament fins el primer caràcter que coincideix amb unCaracter. Aquí teniu alguns exemples:

'squeak és interessant' first  
 – *Escriure el valor retornat:* \$s

'squeak és interessant' size  
 – *Escriure el valor retornat:* 21

'squeak' at: 5  
 – *Escriure el valor retornat:* \$a

'squeak és interessant' at: 11 put: \$f  
 – *Escriure el valor retornat:* \$f

'squeakésinteressant' copyUpTo: \$i  
 – *Escriure el valor retornat:* 'squeakés'

'squeak és interessant' copyUpTo: Character space  
 – *Escriure el valor retornat:* 'squeak'

Per crear un caràcter sense representació gràfica, com l'espai, el tabulador o el *return*, podeu enviar un missatge a la classe Character. Els missatges Character space, Character tab i Character cr retornen respectivament els caràcters espai, tabulador i *return*.

L'Script 17.1 mostra com inserir un *return* dins d'una cadena. Fixeu-vos que el mètode at:put: no retorna la cadena modificada, sinó el caràcter inserit. Aquest és un exemple on l'*efecte* d'un missatge i el seu *resultat* són clarament diferents. Escriure el resultat del missatge 'squeak és interessant' at: 7 put: Character cr no il·lustra l'*efecte* del mètode. El que podem fer és escriure la cadena modificada. Per revisar com escriure per pantalla els resultats d'un enviament de missatge, veieu el capítol 5.

### **Script 17.1 Inserir un return dins una cadena.**

```
| laMevaCadena |
laMevaCadena := 'squeak és interessant'.
laMevaCadena at: 7 put: Character cr.
```

laMevaCadena

- *Escriure el valor retornat: 'squeak és interessant'*

Un caràcter pot ser transformat en una cadena enviant-li el missatge `asString`. A l'*Script 17.2*, es concatenen tres cadenes. La del mig és la cadena '`a`' creada per l'enviament de missatge `$a asString`.

**Script 17.2** *Un caràcter es transforma en una cadena, que és concatenada amb altres cadenes.*

`'sque', $a asString, 'k'`

- *Escriure el valor retornat: 'squeak'*

## Cadenes i nombres

Una cadena pot representar un nombre. Per exemple, la *cadena* '`10`' és una representació textual del *nombre* 10. Tot i així, una cadena no és un nombre. Una cadena no sap com realitzar cap operació matemàtica, i un nombre no sap com comportar-se com una cadena. Per exemple, no podem concatenar dos nombres o sumar dues cadenes. Tot i així, un nombre sap com generar una cadena que *el representa*, utilitzant el mètode `asString`. A més, una cadena sap com convertir la representació d'un nombre en un nombre amb el mètode `asNumber`.

Així doncs, hi ha una diferència entre el nombre 10 i la cadena '`10`'. El nombre 10 representa la quantitat matemàtica 10, mentre la cadena '`10`' és la representació textual del nombre 10 que consisteix en els dos caràcters 1 i 0. La cadena '`10`' es compon dels dos caràcters `$1` i `$0`, i la cadena '`12`' es compon dels dos caràcters `$1` i `$2`. Aquí teniu algunes il·lustracions d'operacions amb cadenes i nombres:

`10 , 12`

- *error! un nombre no coneix el missatge ,*

`'10' , '12'`

- *Escriure el valor retornat: '1012'*

`10 asString`

- *Escriure el valor retornat: '10'*

`10 asString , 12 asString`

– *Escriure el valor retornat: '1012'*

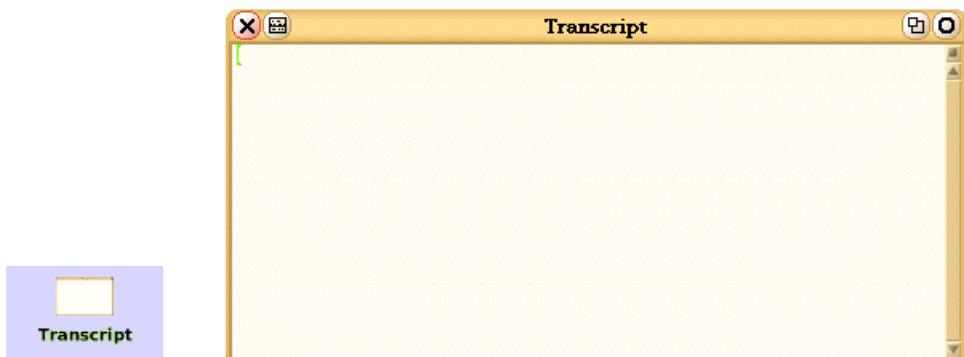
'10' asNumber

– *Escriure el valor retornat: 10*

**Nota** Una cadena (*string*) pot representar un nombre, però aquesta cadena no és un nombre. Per exemple, la cadena '79' es compon dels dos caràcters \$7 i \$9. Per aconseguir la cadena que representa un nombre, cal enviar el missatge *asString* al nombre.

## Utilitzar el *Transcript*

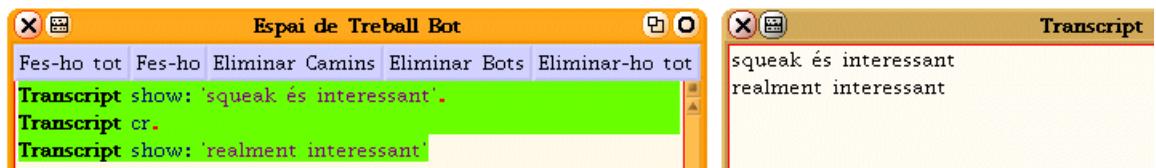
Squeak us ofereix diverses eines molt potents per entendre l'execució dels programes, com per exemple el depurador que ja vam veure al capítol 15. Una altra eina s'anomena el *Transcript*<sup>2</sup>, i pot mostrar informació textual amb cadenes de caràcters. Per obrir una finestra de *Transcript*, arrossegueu la icona disponible en una de les solapes a l'entorn, o trieu l'opció **transcript** del menú **obrir ...**. Això obre una finestra com la que veieu a la figura 17.2.



**Figura 17.2** — Per obrir una finestra amb un Transcript arrossegueu i deixeu dins l'entorn la icona que podeu trobar en una solapa.

<sup>2</sup>Nota del Traductor: Igual que ens ha passat amb *script*, deixarem *Transcript* en anglès, escrivint-lo en cursiva per deixar clar que és un anglicisme.

Hi ha dos missatges per mostrar informació en un *Transcript*: `show:` i `cr.` El missatge `show:` una Cadena escriu la cadena a la finestra, i el missatge `cr.` insereix una línia nova. Veieu la figura 17.3.



**Figura 17.3 —** Escriure dins d'un *Transcript*.

L'*Script* 17.3 dóna alguns exemples mostrant informació en un *Transcript*.

#### **Script 17.3 Mostrar informació en una finestra de Transcript**

`Transcript show: 'squeak és interessant'.`

`Transcript cr.`

`Transcript show: 'realment interessant'.`

Fixeu-vos que una finestra amb un *Transcript* només pot mostrar cadenes. Per tant, si voleu escriure un nombre, haureu d'obtenir la cadena que el representi amb el mètode `asString`. Això ho il·lustrem a l'*Script* 17.4.

#### **Script 17.4 Un nombre es transforma en una cadena abans de poder ser escrit.**

`Transcript show: '21 + 21 és: ' , 42 asString ; cr`

## Generar i comprendre una traça

Ara ens agradaria ensenyar-vos com podeu utilitzar la classe `Transcript` per generar la traça d'un programa. Una traça és una col·lecció d'indicadors del que està passant generats per un programa. Per exemple, podria ser que volguéssiu seguir els moviments d'un robot en un *script* fent que l'*script* escrigués 'Etic girant a la dreta' cada cop que el robot giri a la dreta. Per generar una traça, simplement cal que introduïu una o més expressions dins del vostre *script* que no canviïn l'execució original del programa però que, per exemple, mostrin informació en un *Transcript*. Comencem amb l'*Script* 17.5, que dibuixa una escala amb esglaons de longitud creixent.

**Script 17.5** Una escala amb esglaons de longitud creixent.

```
| pica longitudEsglao |
pica := Bot nou.
longitudEsglao := 10.
10 vegadesRepetir:
    [ pica ves: longitudEsglao.
    pica giraEsquerra: 90.
    pica ves: 5.
    pica giraDreta: 90.
    longitudEsglao := longitudEsglao + 10. ]
```

La primera traça senzilla que podem generar ens diu quan el programa està tot just a punt d'entrar al bucle `vegadesRepetir`, i quan ha sortit del bucle. Això es pot veure a l'*Script 17.6*.

**Script 17.6** L'escala amb una traça senzilla.

```
| pica longitudEsglao |
pica := Bot nou.
longitudEsglao := 10.
Transcript show: 'Abans del bucle' ; cr.
10 vegadesRepetir:
    [ pica ves: longitudEsglao.
    pica giraEsquerra: 90.
    pica ves: 5.
    pica giraDreta: 90.
    longitudEsglao := longitudEsglao + 10. ]
Transcript show: 'Després del bucle' ; cr.
```

### Experiment 17-1 (posar una traça dins del bucle)

Modifiqueu l'*Script 17.6* introduint l'expressió `Transcript show: 'dins del bucle' ; cr.` dins del bucle. El vostre *transcript* hauria d'escriure 'dins del bucle' deu vegades, una vegada cada iteració. També podeu inserir l'expressió `self halt` per permetre obrir el depurador. Aneu amb compte, però, sinó acabareu amb deu depuradors!

Ara utilitzarem la mateixa tècnica per generar una traça una mica més sofisticada. Per exemple, estaria bé veure com canvia el valor de la variable `longitudEsglao` mentre s'executa el pro-

grama. L'*Script* 17.7 conté una expressió nova que escriu el valor de la variable longitudEsglao al començament del bucle cada cop que és executat. Els resultats els podeu veure a la figura 17.4.



**Figura 17.4 — Afegir una traça a un script.**

#### Script 17.7 L'escola amb una traça més sofisticada.

```

| pica longitudEsglao |
pica := Bot nou.
longitudEsglao := 10.
10 vegadesRepetir:
[ Transcript show: '>> ', longitudEsglao asString ; cr.
pica ves: longitudEsglao.
pica giraEsquerra: 90.
pica ves: 5.
pica giraDreta: 90.
longitudEsglao := longitudEsglao + 10. ]

```

Afegir una traça a una assignació és sovint útil, ja que revela un comportament important del programa. Per exemple, a l'*Script* 17.8, l'expressió `Transcript show: 'Després := ' , longitudEsglao asString ; cr.` s'ha afegit després de la instrucció d'assignació que és la darrera expressió del bucle. La traça, que tenia tot seguit de l'*script*, escriu el valor de la variable `longitudEsglao` al començament i al final del bucle. Aquests dos valors haurien de ser el mateix, i, de fet, ho són, com veieu a la traça.

**Script 17.8** L'escala amb una traça després d'una assignació.

```
| pica longitudEsglao |
pica := Bot nou.
longitudEsglao := 10.
10 vegadesRepetir:
[ Transcript show: 'longitudEsglao: ', longitudEsglao asString ; cr.
  pica ves: longitudEsglao.
  pica giraEsquerra: 90.
  pica ves: 5.
  pica giraDreta: 90.
  longitudEsglao := longitudEsglao + 10.
Transcript show: 'longitudEsglao després de := ', longitudEsglao asString ; cr. ]
```

I aquí teniu la traça:

```
longitudEsglao: 10
longitudEsglao després de := 20
longitudEsglao: 20
longitudEsglao després de := 30
longitudEsglao: 30
longitudEsglao després de := 40
longitudEsglao: 40
longitudEsglao després de := 50
longitudEsglao: 50
longitudEsglao després de := 60
longitudEsglao: 60
longitudEsglao després de := 70
longitudEsglao: 70
longitudEsglao després de := 80
longitudEsglao: 80
longitudEsglao després de := 90
longitudEsglao: 90
longitudEsglao després de := 100
longitudEsglao: 100
longitudEsglao després de := 110
```

## Resum

- Una cadena és una seqüència de caràcters delimitada per cometes simples: 'Això és una cadena'. Una cadena representa informació textual com paraules o frases i es pot utilitzar per mostrar informació a la pantalla. Per exemple: 'squeak és interessant' és una cadena de 21 caràcters.
- Un caràcter és una lletra prefixada amb el signe del dòlar \$. Així, \$a representa el caràcter a.
- Una cadena pot representar un nombre, però aquesta cadena no és un nombre. Per exemple, la cadena '79' es compon dels dos caràcters \$7 i \$9. Per aconseguir la cadena que representa un nombre, cal enviar el missatge `asString` al nombre.
- Una finestra de Transcript és una petita finestra que s'utilitza per mostrar missatges. El missatge `show:` unaCadena escriu el valor de l'argument unaCadena, que ha de ser una cadena, a una finestra de *transcript*. El missatge `cr` insereix una línia nova a una finestra de *transcript*.



## Part IV

# Condicionals

Fins ara, els vostres programes han executat totes i cada una de les seves expressions. No heu disposat de cap manera d'expressar que algunes parts d'un programa haurien de ser executades només si es compleixen certes condicions. En aquesta part, us presentem les expressions condicionals, que resolen aquest problema. Aquesta part també introduceix la noció de referències en l'espai bidimensional d'un pla i altres comportaments dels robots. Finalment us ensenyarem com utilitzar un robot per simular el comportament d'animals simples.



# Capítol 18

## Condicions

Fins ara, els programes que heu definit executen *totes* les expressions que contenen, una darrera l'altra. No teniu cap manera de dir que algunes expressions s'haurien d'executar només si determinades *condicions* es satisfan. Aquest capítol i el següent introduceixen un concepte molt important en programació: la noció d'*execució condicional*, és a dir, l'execució d'un determinat fragment de codi només quan es compleix certa condició. Formalment, una *condició* és una expressió que pot ser o bé cert (true) o bé fals (false).

Aquest capítol comença definint un problema senzill que mostra la necessitat de l'execució condicional. Després veurem que una expressió condicional està composta d'una *condició* i d'un *missatge condicional* que pren un o més arguments, l'execució del qual depèn del valor de la condició. Els arguments d'un missatge condicional s'anomenen *blocs condicionals*, i són seqüències d'expressions tancades entre claudàtors [ ].

### Els autèntics colors d'un robot

Imaginem que volem canviar el color d'un robot dependent de la seva distància al centre de la pantalla. Si un robot és a menys de 200 pixels del centre, hauria de ser vermell. Sinó, hauria de ser verd. Aquest problema requereix una execució *condicional*. En funció d'una condició –la posició del robot– el color hauria de canviar.

El mètode *detectorDistancia*, que podeu veure com a Mètode 18.1, mostra una possible solució, i l'*Script 18.1* us ensenya com s'ha d'utilitzar.

**Script 18.1** *Pica canvia el color d'acord amb el mètode detectorDistancia.*

```
| pica |  
pica := Bot nou.
```

```
pica salta: 20.
pica detectorDistancia.
pica salta: 200.
pica detectorDistancia.
```

**Mètode 18.1 Instruccions per tal que qualsevol robot dibuixi un quadrat senzill.**

### **detectorDistancia**

```
| distanciaDelCentre |
distanciaDelCentre := self distanciaDesde: World center.
distanciaDelCentre < 200
    siCert: [ self color: Color vermell ]
    siFals: [ self color: Color verd ]
```

Analitzem què passa quan l'expressió pica detectorDistancia s'executa. Primer, l'expressió self distanciaDesde: World center calcula la distància del receptor al centre de la pantalla. Aquesta distància és guardada a la variable distanciaDelCentre.

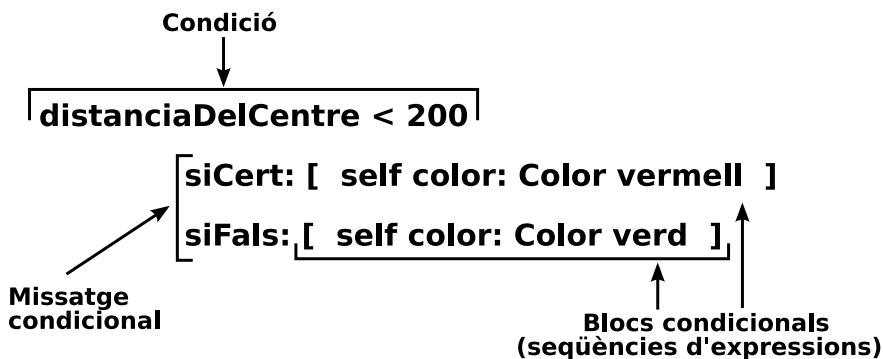
Després l'expressió distanciaDelCentre < 200 siCert: [ self color: Color vermell ] siFals: [ self color: Color verd ] s'executa de la següent manera: *si* la distància del centre és inferior a 200, el color del receptor és canviat a vermell; *sinó*, és canviat a verd. Aquesta expressió és una expressió *condicional*. Ocupa tres línies dins la definició del mètode, però és una sola expressió, i la podeu distingir ja que no hi ha cap punt entre cap de les seves parts.

Diguem-ho un altre cop, una expressió condicional es compon de dues parts: una *condició* i un *missatge condicional* els arguments del qual són *blocs condicionals*. L'expressió distanciaDelCentre < 200 és una condició, l'expressió siCert: [ self color: Color vermell ] siFals: [ self color: Color verd ] és un missatge condicional, i [ self color: Color vermell ] i [ self color: Color verd ] són blocs condicionals, com podeu veure a la figura 18.1.

El mètode siCert:siFals:<sup>1</sup> executa una condició, aquí distanciaDelCentre < 200, i en funció del seu valor, executa un dels blocs condicionals i no executa l'altre. La paraula clau siCert: indica que el bloc condicional [ self color: Color vermell ] s'executa només si la condició és certa. Igualment, la paraula clau siFals: indica que el bloc condicional [ self color: Color verd ] s'executa només si la condició és falsa. Els blocs condicionals també s'anomenen bifurcations de la condició. Suposeu que esteu resseguint amb el dit, en una fotografia, el tronc d'un arbre, i cada cop que trobeu una branca heu de decidir quin camí seguir, i només podeu seguir un camí a la vegada. El terme *bifurcació* es refereix a aquesta situació. Representa el fet que l'execució del programa ha de triar entre camins i executar només un camí, prescindint de l'altre camí.

---

<sup>1</sup>Nota del Traductor: Tal com hem explicat al principi del llibre, aquest mètode està traduït. L'original Smalltalk és ifTrue:ifFalse:.



**Figura 18.1** — Una expressió condicional es compon d'una condició i d'un missatge condicional, els arguments del qual són blocs condicionals.

Ja heu vist dos tipus diferents d'expressió: un tipus, per exemple `self distanciaDesde: World center`, s'executa sempre que la trobem en un programa, mentre que l'altre tipus, aquelles que estan dins de blocs condicionals, s'executen només quan la condició associada és certa o falsa, depèn de quin sigui el cas.

Un bloc condicional no està limitat a una sola expressió, sinó que pot contenir una seqüència d'expressions, com veurem al proper capítol. Així, el mètode `siCert:siFals:` defineix dos blocs condicionals, cada un contenint una seqüència d'expressions (com veureu a la propera secció, fins i tot podeu tenir una seqüència de cap expressió).

Finalment, fixeu-vos que el mètode `siCert:siFals:` és *un* mètode amb *dos* arguments, un pel cas cert i l'altre pel cas fals. Per tant no heu de posar cap punt després del claudàtor `dret ]` que tanca el bloc `siCert:`, ja que trencarieu la instrucció condicional acabant-la massa aviat, i provocarieu un error.

## Afegir una traça per veure què va passant

Per entendre com s'executen les expressions condicionals, hauríeu d'experimentar enviant missatges al Transcript per generar una traça, com vam veure al capítol 17. També podeu utilitzar el depurador inserint l'expressió `self halt`, com us vam ensenyuar al capítol 15. Si voleu saber si s'està executant algun dels blocs condicionals en particular, introduïu `expressions` en aquell bloc. Per exemple, podríeu inserir l'expressió `Transcript show: 'ara executant el bloc corresponent a cert' ; cr` a la branca `siCert:..`. El Mètode 18.2 presenta una manera de generar una traça com la mencionada al context del mètode `detectorDistancia`. Depenent de la posició del robot, es generaran traces diferents. Proveu d'enveigar les traces abans d'executar l'*Script* 18.1. No dubteu a afegir

i modificar aquest tipus de traces a tots els *scripts* que trobeu difícils d'entendre.

**Mètode 18.2** *El mètode per detectar la distància amb una traça.*

### detectorDistancia

```
| distanciaDelCentre |
distanciaDelCentre := self distanciaDesde: World center.
Transcript show: 'sempre' ; cr .
distanciaDelCentre < 200
  siCert: [ self color: Color vermell.
    Transcript show: 'vermell' ; cr ]
  siFals: [ self color: Color verd.
    Transcript show: 'verd' ; cr ]
```

## El valor retornat per un mètode

Quan us vam ensenyar com definir mètodes al capítol 12, vam explicar que executant un mètode no només s'avaluen els missatges que conté, sinó que a més *retorna* un valor. Fins ara, no hem posat especial èmfasi en els valors retornats pels mètodes. Ara, però, començarem a fixar-nos en els valors retornats per la condició d'una expressió condicional. Per exemple, la condició `distanciaDelCentre < 200` retorna un valor (cert o fals) que ens diu si la distància al centre de la pantalla és o no és inferior a 200 (aprendreu més sobre true –cert– i false –fals– al capítol 20).

Altres expressions que apareixen al Mètode 18.1 retornen valors d'interès. Per exemple, no només l'expressió `self distanciaDesde: World center` calcula la distància del receptor al centre de la pantalla, sinó que també *retorna* aquesta distància. Després, la condició `distanciaDelCentre < 200` utilitza aquesta distància per decidir quina branca de l'expressió condicional hauria de ser executada.

**Important!** Una expressió condicional es compon d'una *condició* i d'un *missatge condicional* que pren un o més arguments i l'execució del qual depèn del valor d'una condició. Els arguments d'un missatge condicional s'anomenen *blocs condicionals*, i són seqüències d'expressions tancades entre claudàtors [ ]:

### unaCondicio

```
siCert: [ expressionsSiCondicioEsCerta ]
siFals: [ expressionsSiCondicioEsFalsa ]
```

## Expressions condicionals amb una sola bifurcació

Algunes vegades, heu de realitzar una acció quan determinada condició és certa, però si la condició és falsa no voleu fer res (i viceversa). Prenent un exemple de la vida real, si porteu un paraigües tancat i comença a ploure, voleu obrir-lo, però si no plou, no voleu fer res de res. Prenent un exemple d'Squeak, el mètode vermellSiPropDelCentre (Mètode 18.3) canvia el color del receptor a vermell quan és a una distància més petita que 200 píxels del centre de la pantalla, però si la distància no és més petita que 200 píxels, el color del robot no canvia.

**Mètode 18.3** *Si voleu no fer res quan la condició és falsa, podeu utilitzar un bloc condicional buit amb siCert:siFals:.*

### vermellSiPropDelCentre

```
| distanciaDelCentre |
distanciaDelCentre := self distanciaDesde: World center.
distanciaDelCentre < 200
    siCert: [ self color: Color vermell ]
    siFals: [ ]
```

Amb el mètode siCert:siFals:, deixeu la segona branca buida, amb un bloc condicional sense cap expressió: [ ]. Smalltalk, però, proporciona dos mètodes alternatius, siCert: i siFals:, per escriure aquesta classe d'expressions condicionals. El mètode siCert: executa el seu bloc condicional quan la condició és certa. Utilitzant el mètode siCert: podem reescriure el Mètode 18.3 més convenientment com el Mètode 18.4.

**Mètode 18.4** *Amb el mètode siCert:, ja no us cal un bloc condicional buit.*

### vermellSiPropDelCentre

```
| distanciaDelCentre |
distanciaDelCentre := self distanciaDesde: World center.
distanciaDelCentre < 200
    siCert: [ self color: Color vermell ]
```

El mètode siFals: és completament anàleg al mètode siCert:, executa el seu bloc condicional quan la condició és *falsa*. Cada un dels mètodes siCert: i siFals: executa una condició i després, en funció del valor retornat per la condició, el mètode executa, o no, el bloc condicional.

---

**Important!** El mètode `siCert:` executa el seu bloc condicional quan la condició és certa. El mètode `siFals:` executa el seu bloc condicional quan la condició és falsa.

*unaCondicio*

**siCert:** [ *expressionsSiCondicioEsCerta* ]

*unaCondicio*

**siFals:** [ *expressionsSiCondicioEsFalsa* ]

---

## Triar el mètode condicional adequat

Utilitzar `siCert:siFals:` no és el mateix que utilitzar `siCert:` seguit de `siFals:`. Amb `siCert:siFals:` la condició que el precedeix només s'executa una vegada, però si voleu utilitzar `siCert:` seguit de `siFals:`, us cal una condició davant de cada missatge, i totes dues seran executades, fins i tot si són idèntiques. Això pot donar lloc a conseqüències no desitjades si el bloc condicional de la primera expressió condicional (`siCert:`) modifica allò que comprovem a la condició de la segona expressió condicional (`siFals:`). En aquest cas, utilitzar `siCert:` seguit de `siFals:` no és equivalent a utilitzar `siCert:siFals:`.

## Imbricar expressions condicionals

Una expressió condicional pot contenir qualsevol altra expressió dins dels seus blocs condicionals, en particular, aquests blocs poden contenir altres expressions condicionals. Això és el que ara tractarem. No hi ha res conceptualment nou, però és una pràctica útil i habitual, per tant pensem que és interessant que us l'ensenyem.

## Acolorir robots amb tres colors

Modifiquem el problema anterior. Si un robot és a menys de 200 píxels del centre de la pantalla, hauria de ser de color vermell; si està situat entre 200 i 300 píxels lluny del centre, hauria de ser de color groc, i si la distància és superior a 300 píxels, hauria de ser de color verd.

En aquest problema, diferents parts del mètode haurien de ser executades sota circumstàncies diferents. És a dir, hi ha tres rangs diferents de distàncies, i el color del robot hauria de canviar a vermell, groc o verd dependent d'on es trobi. Una solució possible al nostre problema la podem veure al Mètode 18.5. El que hem fet és dividir el problema en dos trossos: què fer si la distància del robot al centre és superior a 300 píxels, i què fer si no ho és. Després dividim el tros “què fer si no” en dos subproblems: què fer si la distància al centre és inferior a 200, i què fer si no ho és.

**Mètode 18.5** Acolorim el robot amb un color, entre tres colors possibles, en funció de la seva distància del centre de la pantalla.

### modificaTresColors

```
| distancia |
distancia := self distanciaDesde: World center.
distancia > 300
    siCert: [ self color: Color verd ]
    siFals: [ distancia < 200
        siCert: [ self color: Color vermell ]
        siFals: [ self color: Color groc ] ]
```

El Mètode 18.6 conté exactament el mateix codi, amb codi emfatitzat per mostrar les expressions condicionals. Hi ha dues expressions condicionals diferents. La primera (Expressió condicional 1) es mostra en itàlica, i la segona (Expressió condicional 2) en negreta. La segona expressió condicional s'executa només si la condició de la primera expressió condicional és falsa. És a dir, si la distància és menor o igual a 200, aleshores l'expressió condicional 2 s'executa. Si l'expressió condicional 2 s'executa, la seva condició  $< 200$  també s'executa, i en funció del seu valor, la branca *siCert:* o la branca *siFals* s'executarán. Aquí teniu les dues expressions condicionals:

### Expressió condicional 1:

```
distancia > 300
    siCert: [ self color: Color verd ]
    siFals: [ "executa l'expressió condicional 2" ]
```

### Expressió condicional 2:

```
distancia < 200
    siCert: [ self color: Color vermell ]
    siFals: [ self color: Color groc ]
```

I aquí teniu el Mètode 18.6.

**Mètode 18.6** Aquest és el Mètode 18.5 emfatitzat.

### modificaTresColors

```
| distancia |
distancia := self distanciaDesde: World center.
distancia > 300
  siCert: [ self color: Color verd ]
  siFals: [ distancia < 200
    siCert: [ self color: Color vermell ]
    siFals: [ self color: Color groc ] ]
```

Si teniu dificultats identificant quin bloc condicional serà executat, trieu alguns valors particulars per a la distància (com per exemple, 150, 250 i 350). Proveu de fer una execució manual del mètode i subratlleu la part del mètode que serà executada. Seguint cada pas veureu que només s'executen algunes branques. Podeu afegir algunes traces per mostrar com s'executen les diferents condicions, o podeu utilitzar el depurador per anar executant el mètode pas a pas.

## Aprendre dels errors

Tothom comet errors. Precisament per això estudiar errors de programació és una manera excellent d'aprendre i entendre un concepte des d'una altra perspectiva. Hem definit un mètode girDeColors: unAngle que canvia el color d'un robot en funció de la direcció a què apunta. Quan el robot apunta al nord, ens agradaria que fos de color blau per representar el fred. Voldríem que fos vermell quan apunta al sud, i en els altres casos, hauria de ser verd. El primer intent de definir aquest mètode, que no va tenir gaire èxit, el teniu com a Mètode 18.7.

**Mètode 18.7** El color del robot depèn de la seva direcció: primer intent (amb un error).

### girDeColors: unAngle

"canvia el color del robot de manera que és blau quan apunta al nord, vermell si apunta al sud, i verd en els altres casos"

```
self gira: unAngle.
self direccio = 90
  siCert: [ self color: Color blau ].
self direccio = -90
  siCert: [ self color: Color vermell ]
  siFals: [ self color: Color verd ]
```

La definició del Mètode 18.7 no és correcte. Proveu d'entendre per què no ho és abans de continuar llegint. Seguiu la definició del mètode per veure que quan un robot està apuntant al nord, primer tindrà el color blau, tal com hauríem esperat, però després es tornarà verd, i això no és el que hauria de passar. Hauria de mantenir el color blau després d'haver-li assignat el color blau. L'*Script* 18.2 il·lustra l'error en el mètode.

**Script 18.2** *Hi ha un error en el mètode!*

```
| pica |
pica := Bot nou.
pica girDeColors: -90.
pica color.
– Escriure el valor retornat: Color red “ok”
pica girDeColors: 90.
pica color.
– Escriure el valor retornat: Color green “ok”
pica girDeColors: 90.
pica color.
– Escriure el valor retornat: Color green “error”
```

*Què ha anat malament?* Executeu el Mètode 18.7 mentalment per identificar l'error. El problema és que quan el robot està apuntant al nord, la condició self direccio = 90 és certa, i el bloc associat s'executa, acolorint el robot de blau. El mètode hauria d'acabar aquí. Però quan el mètode continua i executa el bloc condicional de la segona expressió condicional, el color del robot canvia a verd ja que la direcció del robot no és sud. El següent codi comentat ho il·lustra.

```
pica girDeColors: 90.

self direccio = 90
    siCert: [ self color: Color blau ]           “és cert”
                                                “així el missatge condicional cert
                                                s'executa; el robot es torna blau
                                                i evalua la propera condició”
self direccio = -90
    siFals: [ self color: Color verd ]          “és fals”
                                                “de manera que el missatge condicional
                                                fals s'executa. Error!”
```

*Com corregir-ho?* En fer la traça del mètode, vau veure que quan el robot apunta al nord, la segona expressió condicional no s'hauria d'executar. S'hauria d'executar només si la primera

condició és falsa. Per tant, podeu utilitzar expressions condicionals imbricades. El codi correcte el teniu al Mètode 18.8.

**Mètode 18.8** *El color del robot depèn de la seva direcció: versió correcta.*

#### **girDeColors: unAngle**

“canvia el color del robot de manera que és blau quan apunta al nord, vermell si apunta al sud, i verd en els altres casos”

```
self gira: unAngle.
self direccio = 90
  siCert: [ self color: Color blau ].
  siFals: [ self direccio = -90
    siCert: [ self color: Color vermell ]
    siFals: [ self color: Color verd ] ]
```

## Interpretar un petit llenguatge

Investigadors en biologia teòrica han desenvolupat sistemes per estudiar el creixement de les plantes, anomenats *Lindenmeyer Systems*. Els sistemes de Lindenmeyer es basen en gràfics de robots semblants als robots amb què ja heu estat experimentant. Els robots de Lindenmeyer entenen un petit llenguatge format per caràcters com \$v i \$g. L'acció d'un robot s'associa a cada un d'aquests caràcters. Per exemple, el caràcter \$v (de “ves”) és associat al moviment cap a endavant, mentre \$g (de “gira”) és associat a girar un angle de 45 graus. Un sistema de Lindenmeyer genera una seqüència de caràcters. Aquests caràcters són interpretats per un robot, i la seqüència d'accions que fa el robot genera el dibuix.

Anem a definir un mètode *interpreta: unCaracter* que fa que un robot es mogui endavant si el caràcter és \$v i gira 45 graus en sentit contrari a les agulles del rellotge si el caràcter és \$g. L'*Script* 18.3 il·lustra com utilitzar el mètode, i el mètode es defineix al Mètode 18.9.

**Script 18.3** *Utilitzar el mètode *interpreta: unCaracter**

```
| pica |
pica := Bot nou.
4 vegadesRepetir:
  [ pica
    interpreta: $v ;
    interpreta: $g ;
    interpreta: $v ;
```

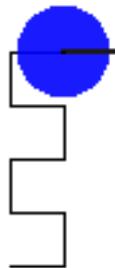
interpreta: \$v ;  
interpreta: \$g ;  
interpreta: \$v ]

### **Mètode 18.9 Interpretar un caràcter.**

interpreta: unCaracter

```
unCaracter = $v
    siCert: [ self ves: 20 ]
    siFals: [unCaracter = $g
        siCert: [ self gira: 45 ] ]
```

Proveu d'escriure un *script* que reproduueixi la figura 18.2. Com que una cadena és una seqüència de caràcters, pot codificar un dibuix generat per un robot de Lindenmeyer. Encara que utilitz a uns mètodes que encara no hem explicat (seran explicats a la seqüèl d'aquest llibre), intenteu l'*Script* 18.4, que repetidament envia el missatge interpreta: al robot amb cada caràcter de la cadena. Després, experimenteu una mica més canviant la cadena utilitzada a l'*script* per crear les vostres figures. Construireu un sistema de Lindenmeyer complet a la seqüèl d'aquest llibre.



**Figura 18.2** — Un dibuix generat amb el mètode interpreta: amb la seqüència de caràcters 'vgggggvooooooooooooovggggggvggvvgggggggvggggv'.

#### **Script 18.4 Utilitzar interpreta: en un bucle**

## Més experiments

Milloreu el mètode interpreta: unCaracter de manera que tant \$v com \$V facin que el robot vagi endavant, i tant \$g com \$G facin que el robot giri 45 graus en sentit contrari a les agulles del rellotge. També podeu utilitzar el caràcter \$+ abans de \$g o \$G per fer girar el robot 45 graus en sentit contrari a les agulles del rellotge, i de manera similar \$- per fer-lo girar 45 graus en el sentit de les agulles del rellotge.

## Resum

Una expressió condicional es compon de dues parts: una *condició* i un *missatge condicional*, que conté un o més *blocs condicionals*. Quin bloc condicional serà executat depèn del valor de la condició. La taula següent mostra alguns dels mètodes utilitzats amb condicions.

Mètode	Descripció	Exemple
<code>unaCondicio siCert: [ expressionsSiCondicioCerta ] siFals: [ expressionsSiCondicioFalsa ]</code>	Executa <code>expressionsSiCondicioCerta</code> només si <code>unaCondicio</code> és certa.  Executa <code>expressionsSiCondicioFalsa</code> només si <code>unaCondicio</code> és falsa.	<pre>self direccio = 90 siCert: [ self color: Color verd ] siFals: [ Beeper beep ]</pre> <p>Si un robot apunta al nord, es torna verd.</p> <p>El sistema es queixa només quan el robot no està apuntant al nord</p>
<code>unaCondicio siCert: [ expressionsSiCondicioCerta ] siFals: [ expressionsSiCondicioFalsa ]</code>	Executa <code>expressionsSiCondicioCerta</code> si <code>unaCondicio</code> és certa; en altre cas, executa <code>expressionsSiCondicioFalsa</code> .	<pre>self direccio = 90 siCert: [ self color: Color verd ] siFals: [ Beeper beep ]</pre>



# Capítol 19

## Repeticions condicionals

Les expressions condicionals són una eina molt potent per crear programes complexos ja que us permeten controlar el flux d'execució, bifurcant-lo d'una manera si una determinada condició és certa i d'una altra manera si és falsa. Tot i així, no n'hi ha prou només amb les condicions. Alguns programes necessiten combinar bucles i condicions en *repeticions condicionals*, és a dir, bucles que executen un bloc d'expressions *mentre* es satisfà una determinada condició, aturant l'execució quan aquesta condició ja no es compleix. Aquest capítol presenta els bucles condicionals oferits per Smalltalk i introduceix alguns exemples senzills. Després, al capítol 23, utilitzarem els bucles condicionals per simular alguns comportaments animals.

### Repeticions condicionals

La idea darrera dels bucles condicionals és que un bloc (una seqüència d'expressions) és repetit mentre es compleix una certa condició (o, de manera alternativa, mentre una determinada condició no es compleix). Smalltalk defineix dos missatges, `whileTrue:` i `whileFalse:`<sup>1</sup>, que us permeten definir bucles condicionals.

El que fan aquests bucles és clar a partir del nom del mètode: `mentreCert:` executa la seva condició i executa el bloc condicional mentre la condició sigui certa. El mètode `mentreFals:` fa el mateix, però executa el bloc condicional només mentre la condició sigui falsa.

---

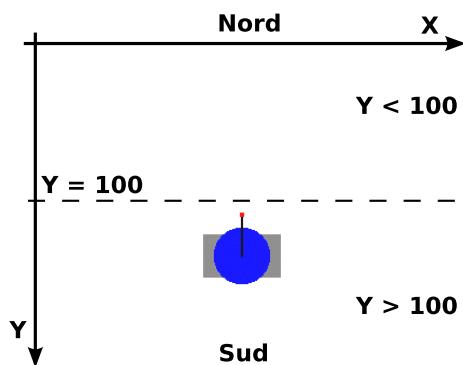
**Important!** `mentreCert:` i `mentreFals:` us permeten definir bucles condicionals pels que es repeteix el bloc condicional mentre la condició es satisfà (`mentreCert:`) o no es satisfà (`mentreFals:`)

---

<sup>1</sup>Nota del Traductor: Que, com ja hem dit, aquí hem traduït per `mentreCert:` i `mentreFals:`.

## Un exemple

Anem a posar un exemple senzill. Suposem que tenim un robot en algun lloc a la part sud de la pantalla, on la coordenada  $y$  és més gran que 100 (per Squeak, l'eix  $y$  va de nord a sud, és a dir, creix cap avall), i volem que el robot es mogui cap al nord fins que la seva coordenada  $y$  sigui menor que 100 pixels (veure figura 19.1). Una solució utilitzant un bucle condicional es pot veure al Mètode 19.1, i la podeu fer servir com veieu a l'Script 19.1.



**Figura 19.1** — El mètode finsA100 mou el robot cap al nord mentre la seva coordenada  $y$  sigui més gran que 100.

### Script 19.1 Utilitzar el mètode finsA100

```
| pica |
pica := Bot nou.
pica finsA100
```

### Mètode 19.1 Moure el robot cap al nord mentre la seva coordenada $y$ sigui més gran que 100

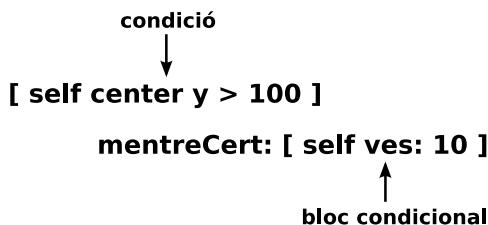
finsA100

“Gira el receptor cap al nord i mou-lo deu pixels cada vegada fins que la seva coordenada  $y$  sigui inferior a 100. Després torna verd el receptor”

```
self nord.
[ self center y > 100 ]
    mentreCert: [ self ves: 10 ].
self color: Color verd.
```

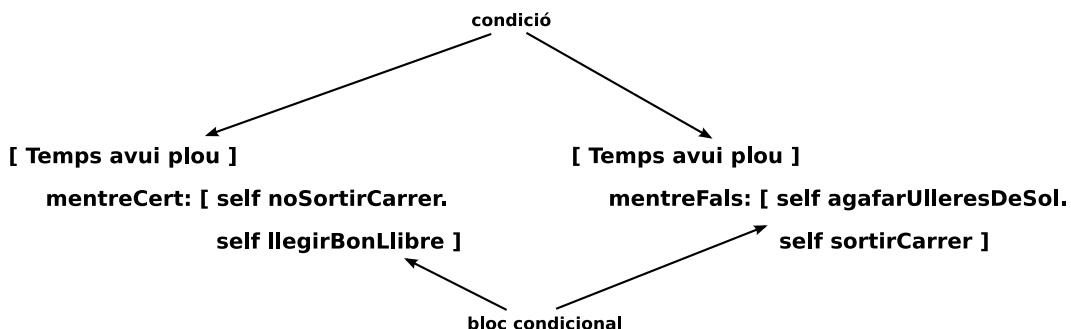
Anem a veure amb detall què passa quan executem aquest mètode.

1. L'expressió `self` nord no és part del bucle condicional, així només s'executa un cop.
2. El bucle condicional es compon d'una condició i d'un missatge condicional, com veieu a les figures 19.2 i 19.3. La condició, que aquí expremem amb el bloc `[ self center y > 100 ]`, s'executa (i retorna un valor cert o fals).



**Figura 19.2** — Un bucle condicional `mentreCert:` es compon d'una condició i d'un missatge condicional, que conté un bloc condicional (una seqüència d'expressions).

3. El mètode `mentreCert:` especifica el bucle: Si el resultat de la condició del pas 2 és certa, llavors el bloc condicional `[ self ves: 10 ]` s'executa, i després de l'acabament de l'execució del bloc condicional, l'execució del mètode torna al pas 2 (on la condició s'executa altre cop). Per altra part, si la condició és falsa, l'execució continua al pas 4.



**Figura 19.3** — Els bucles condicionals `mentreCert:` i `mentreFals:` es componen d'una condició i d'un missatge condicional. El bloc condicional del `mentreFals:` s'executa mentre la condició del `mentreFals:` és falsa; el bloc condicional del `mentreCert:` s'executa mentre la condició del `mentreCert:` és certa.

4. En aquest pas, el resultat de la condició [ self center y > 100 ] del pas 2 és false, de manera que el bloc condicional no s'executa i el bucle s'atura. El programa va al pas 5.
  5. L'execució del mètode es reprèn a la primera expressió després del bloc condicional, en aquest exemple, l'expressió self color: Color verd s'executa, i el mètode s'acaba.
- 

**Important!** Un bucle condicional es compon d'una condició i d'un missatge condicional, que conté un bloc condicional (una seqüència d'expressions).

```
[ condicio ] mentreFals:
  [ missatges condicionals ]
[ condicio ] mentreCert:
  [ missatges condicionals ]
```

---

## Experiències amb traces

Us suggerim que afegiu traces al mètode que heu escrit i analitzeu la traça resultant. També, utilitzeu el depurador. No dubteu a modificar el lloc on poseu la traça dins del mètode, ja que diferents col·locacions de Transcript show: poden mostrar traces diferents. Per exemple, al Mètode 19.2 hem introduït una traça al començament del bloc condicional. Obtenim la traça que veieu a la figura 19.4.

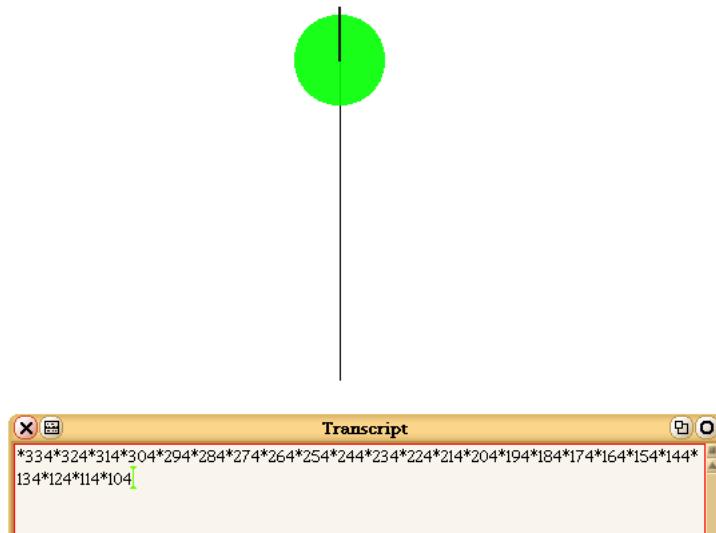
**Mètode 19.2** *El mètode finsA100 amb una traça dins del bucle*

### finsA100

"Gira el receptor cap al nord i mou-lo deu píxels cada vegada fins que la seva coordenada y sigui inferior a 100. Després torna verd el receptor"

```
self nord.
[ self center y > 100 ]
  mentreCert:
    [ Transcript show: '*' , self center yasString.
      self ves: 10 ].
```

self color: Color verd.



**Figura 19.4 — Una traça de l'execució del mètode finsA100**

Com a alternativa podeu introduir la línia Transcript show: '#', self center yasString ; cr. després de l'expressió self ves: 10, com veieu al Mètode 19.3, o fins i tot dins del bloc condicional, abans de la primera línia, com al Mètode 19.4.

### Mètode 19.3 Posar transcripts dins del mètode finsA100

#### finsA100

"Gira el receptor cap al nord i mou-lo deu píxels cada vegada fins que la seva coordenada y sigui inferior a 100. Després torna verd el receptor"

```
self nord.
[ self center y > 100 ]
mentreCert:
[ self ves: 10.
  Transcript show: '#', self center yasString ; cr ]
self color: Color verd.
```

**Mètode 19.4 Posar transcripts dins el mètode finsA100****finsA100**

"Gira el receptor cap al nord i mou-lo deu píxels cada vegada fins que la seva coordenada y sigui inferior a 100. Després torna verd el receptor"

```
self nord.
[ Transcript show: 'c' , self center y asString ; cr.
self center y > 100 ]
mentreCert:
[ Transcript show: '*' , self center y asString ; cr.
self ves: 10 ].
self color: Color verd.
```

Comparem les traces produïdes pels diferents mètodes. Fixeu-vos en particular en els valors finals. Observeu que mentreCert: pot ser convertit en mentreFals: negant la condició (quedar-se a casa mentre plou és lògicament el mateix que quedar-se a casa mentre és fals que no plou). Utilitzeu el mètode amb el qual entengueu millor el vostre programa. Proveu de redefinir el mètode finsA100 utilitzant mentreFals: en lloc de mentreCert:. Després definiu un mètode que faci moure el robot cap al nord un píxel cada cop. Compareu les posicions exactes on el robot s'atura.

Definir blocs condicionals pot ser difícil. Tingueu al cap els següents aspectes (per a un bucle mentreCert:; els mateixos aspectes, convenientment adaptats, podrien ser emfatitzats dels bucles mentreFals:):

- La condició s'executa *abans* que el missatge a mentreCert: s'executi.
- Si la condició és certa i per tant s'executa el bloc condicional, la condició s'executa altre cop. *Alguna cosa ha de passar dins del bucle* (com per exemple moure el robot cap al nord) que eventualment faci falsa la condició. Sinó, el bucle mai no acabaria.
- Si la condició és falsa la primera vegada que s'executa, el bucle mai no s'executarà.

És fàcil oblidar-se de comprovar la condició amb cura pel que fa al segon aspecte tot just mencionat, és a dir, que la condició eventualment es torni falsa (o certa en el cas del bucle mentreFals:). Sinó, el bucle es repetirà infinitament. En escriure un bucle condicional, sempre cal tenir en compte que, d'alguna manera, el bucle hauria d'anar treballant per arribar a la seva fi. El bucle hauria de tendir cap a una situació que fa la condició falsa per a un bucle mentreCert: o certa per a un bucle mentreFals:.

Per investigar el tercer aspecte mencionat, proveu de fer el següent experiment: Moveu el robot utilitzant l'halo negre prop de la part de dalt de la finestra d'halos de manera que la seva coordenada  $y$  sigui inferior a 100. Invoqueu aleshores el mètode `finsA100`. Com veieu, no passa res, que és exactament el que hauria de passar: el mètode s'invoca i la condició `self center y > 100` és falsa, ja que la posició del robot és inferior a 100. Per tant, el bloc condicional no s'executa.

## Aturar un bucle infinit

No és difícil escriure un bucle sense fi. Us en trobareu un si la condició no retorna mai `true` per `mentreFals`: o mai retorna `false` per `mentreCert`:

Si trobeu que heu escrit i executat un bucle sense fi, el podeu aturar prement *Command* i punt en un Mac o bé *Alt* i punt o *Control* i C en altres plataformes. Un cop heu aturat el bucle, heu d'esbrinar per quina raó el bucle no ha acabat entrant en el depurador, prement el botó **debug** a la finestra que apareix. També podeu escriure i analitzar informació utilitzant el Transcript.

---

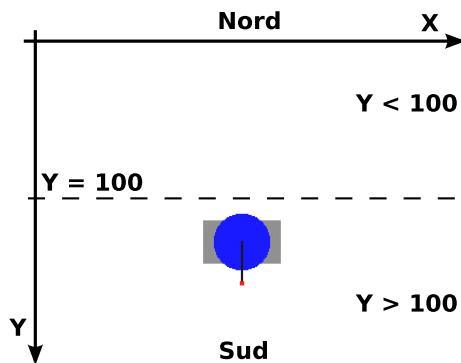
**Important!** Si un bucle condicional s'executa una vegada, no s'aturarà mai si el bloc condicional no realitza alguna acció que eventualment “trenqui” la condició. Això vol dir que la faci falsa si el bucle és `mentreCert`: o la faci certa si el bucle és `mentreFals`:

---

Anem a considerar aquesta difícil qüestió amb l'exemple que ja coneixem. Si el bucle ha d'acabar-se, la distància entre el robot i la línia horitzontal amb coordenada  $y$  de valor 100 ha de fer-se cada cop més petita, fins que el robot creui la línia amb coordenada  $y$  100, i per tant, per poder acabar el bucle, les expressions dins del bloc condicional han de reduir aquesta distància d'alguna manera. Més concretament, mentre la coordenada  $y$  del robot és massa gran ( $>= 100$ ), el bucle continuarà. Per estar segurs que el bucle s'acabará, el bucle condicional ha de disminuir la coordenada  $y$  del robot. L'expressió `self ves: 10` dins el bloc condicional fa això, ja que la coordenada  $y$  del robot es fa cada cop més petita a mida que el robot es mou cap al nord (i el mètode `s'assegura al principi` que el robot apunti al nord).

Analitzem el cas del Mètode 19.5, `finsA100Infinit`. Aquí la condició, com en els mètodes previs, aturarà el bucle quan la coordenada  $y$  és inferior a 100, és a dir, quan el robot passi per sobre de la línia horitzontal a 100 píxels del capdamunt de la pantalla. Aquest mètode, però, fa que inicialment el robot apunti al sud, com veieu a la figura 19.5, i no hi ha cap manera de fer que la seva coordenada  $y$  sigui inferior a 100 si comença al sud de la línia amb coordenada  $y$  100. En aquest cas, el Mètode 19.5 mai s'acabará, ja que el bloc condicional no pot fer que la condició sigui falsa. El problema en aquest cas és que el bloc condicional *incrementa* el valor de la coordenada  $y$ , i així la possibilitat que la condició s'avaluï a fals disminueix a cada moviment del robot. Aquest

exemple és més aviat exagerat, però il·lustra clarament el problema de fallar a l'hora de construir un bucle que s'acabi.



**Figura 19.5 — La situació per a l'execució del mètode finsA100Infinit.**

**Mètode 19.5** Una situació que ens duu a un bucle infinit

#### finsA100Infinit

"Gira el receptor cap al sud i mou-lo deu píxels cada vegada fins que la seva coordenada y sigui inferior a 100. Després torna verd el receptor"

```
self sud.  
[ self center y > 100 ]  
    mentreCert: [ self ves: 10 ].  
    self color: Color verd.
```

---

**Important!** Quan definiu un bucle, sempre pregunteu-vos si hi ha cap possibilitat que la condició no es pugui "trencar" mai. Si la condició no té cap possibilitat de fallar, el bucle continuarà executant-se per sempre.

---

## Aprofundir en els bucles condicionals

El lector astut s'haurà adonat que la condició en un bloc condicional és un bloc, ja que és una expressió envoltada per claudàtors. Una conseqüència d'això és que la condició d'un bucle

condicional no té per què limitar-se a contenir només una sola expressió. Aquest bloc pot contenir una seqüència d'expressions, sempre que el darrer missatge de la condició retorna True o False. Això us permet expressar bucles condicionals més complicats. El següent "script" us mostra el bloc en forma esquemàtica. Fixeu-vos que el receptor del missatge condicional és un bloc, en què el receptor fa alguna cosa, després algun objecte fa una altra cosa, i finalment s'avalua una condició: encara treballa el receptor?

```
[ self fesAixo.
  unAltroObjecte fesAllo.
  self encaraTreballa ]
  mentreCert:
    [ self renegar!ContinuarTreballant ]
```

Amb aquesta opció podeu canviar el mètode finsA100 per que s'assemblí al Mètode 19.6. Tot i que aquest mètode és molt similar al mètode finsA100, pot tenir efectes diferents en circumstàncies diferents. Proveu d'entendre quina és la diferència. Per exemple, afegiu una traça o invoqueu el depurador dins el Mètode 19.6 i analitzeu-ho.

**Mètode 19.6** *Un mètode modificat per moure el robot cap al nord.*

#### noElMateixFinsA100

"Gira el receptor cap al nord i mou-lo deu píxels cada vegada fins que la seva coordenada *y* sigui inferior a 100. Després torna verd el receptor"

```
self nord.
[ self ves: 10.
  self center y > 100 ]
  mentreCert: [].
self color: Color verd.
```

Com que la condició *sempre s'executa com a mínim un cop*, el robot es mourà com a mínim deu píxels, fins i tot si la seva coordenada *y* és inferior a 100 al començament. Això no passava a les versions anteriors d'aquest mètode.

## Una aplicació interactiva senzilla

Suposem que voleu que l'usuari decideixi quants passos de deu píxels cap a l'est i després deu píxels cap al nord hauria de fer un robot. A l'Script 19.2 veureu com podeu fer això. L'usuari és preguntat pel nombre de passos; després s'utilitza una expressió condicional (com les del capítol

anterior) en la qual una condició comprova si l'entrada de l'usuari és un nombre vàlid (amb el mètode `isAllDigits`). Si ho és, es converteix en un nombre (mètode `asNumber`), i el robot fa el nombre de passos especificat. Si l'entrada de l'usuari no és vàlida, l'*script* s'acaba i el robot ja no es mou més.

**Script 19.2** Una escala interactiva amb dades d'entrada proporcionades per l'usuari.

```
| pica resposta |
resposta := (FillInTheBlank
            request: 'Nombre de passos'
            initialAnswer: '15').
resposta isAllDigits
siCert: [ pica := Bot nou.
           resposta asNumber vegadesRepetir:
             [ pica ves: 10 ; nord ; ves: 10 ; est ] ]
```

Però ho podem fer millor! L'*Script 19.3* mostra un meravellós ús dels bucles condicionals per demanar a l'usuari un valor, i continuar demanant-lo (bucle) fins que s'introdueixi un valor vàlid. Altre cop, demanarem a l'usuari que introdueixi una cadena que representi el nombre de passos. Altre cop, comprovarem si la cadena introduïda representa un nombre amb `isAllDigits`. Aquesta vegada, però, la petició de dades a l'usuari està dins del bloc condició, i si les dades introduïdes no representen un nombre vàlid (... `resposta isAllDigits`] mentreFals:), es repetirà la petició a l'usuari. Aquest bucle es repetirà fins que l'usuari introduceixi un nombre vàlid.

**Script 19.3** Una escala interactiva amb un bucle controlat per les dades d'entrada proporcionades per l'usuari.

```
| pica resposta |
[ resposta := (FillInTheBlank
              request: 'Nombre de passos'
              initialAnswer: '10').
resposta isAllDigits ] mentreFals: [ ].
pica := Bot nou.
resposta asNumber vegadesRepetir:
  [ pica ves: 10 ; nord ; ves: 10 ; est ]
```

## Quan hem d'utilitzar claudàtors

Sens dubte us heu fixat que hem introduït condicions que requereixen l'ús de claudàtors sense res a dins [ ]. També pot ser que us hagueu fixat que hem posat claudàtors envoltant la condició (o bloc condició) a les expressions condicionals dels `mentreCert:` i `mentreFals:` que han aparegut en aquest capítol. Així, quan cal utilitzar claudàtors? Hi ha bàsicament dues regles a Smalltalk: heu d'envoltar una expressió o seqüència d'expressions amb [ i ] en els casos següents:

- Quan heu d'executar la mateixa expressió un cert nombre de vegades. Per exemple,
  - `4 vegadesRepetir: [ pica ves: 10 ; giraEsquerra: 90 ]` repeteix els enviaments de missatge `pica ves: 10 ; giraEsquerra: 90` quatre vegades.

Fixeu-vos que el nombre de vegades que l'expressió es repeteix pot ser 1 o fins i tot zero, però tot i així heu de fer servir claudàtors: `1 vegadesRepetir: [ self ves: 120 ]`.

- Quan una expressió és executada un nombre variable de vegades. Per exemple,
  - `distancia < 200 siCert: [ self color: Color vermell ]` executa `self color: Color vermell` només en determinades circumstàncies.
  - `[ self center y > 100 ] mentreCert: [ self ves: 10 ]` repeteix condicionalment les dues expressions `self center y > 100` i `self ves: 10` múltiples vegades. Per tant el receptor i l'argument són blocs.

Per concretar, aquí teniu la definició real de les situacions en què els claudàtors són necessaris: l'argument d'un missatge condicional (`siCert:`, `siFals:`, `siCert:siFals:,` `vegadesRepetir:`) o un bucle condicional (`mentreCert:,` `mentreFals:`) són envoltats de claudàtors. El receptor d'un missatge en un bucle condicional (`mentreCert:,` `mentreFals:`) també s'ha de representar entre claudàtors.

## Resum

- Un bucle condicional es compon d'una condició i d'un missatge condicional, l'argument del qual és un bloc condicional que conté una seqüència d'expressions.
- Els mètodes `mentreCert:` i `mentreFals:` us permeten definir bucles condicionals en els quals el bloc condicional és repetit mentre la condició es satisfà (`mentreCert:`) o no (`mentreFals:`).
- Quan esteu definint una repetició, sempre us heu de preguntar si existeix la possibilitat que la condició no es satisfaci mai, cas en què el bucle no s'executarà mai. Per altra part, si la condició no té cap opció de fallar, el bucle es repetirà indefinidament.

- Envolteu una expressió o seqüència d'expressions amb claudàtors quan (1) necessiteu executar la mateixa expressió diverses vegades (4 vegadesRepetir: [ self ves: 10 ]) o (2) l'expressió s'executa condicionalment (dist < 200 siCert: [ self color: Color blau ]). També, el receptor d'un bucle condicional (mentreCert:, mentreFals:) s'ha de representar entre claudàtors.

Aquí teniu una descripció dels mètodes introduïts en aquest capítol:

Mètode	Descripció	Exemple
[ unaCondició ] mentreFals: [ SeqüènciaDeMissatges ]	Executa <i>unaCondició</i> . Si és falsa, executa <i>SeqüènciaDeMissatges</i> i repeteix aquest pas.  Si <i>unaCondició</i> és certa, passar a executar la propera expressió, sense executar <i>SeqüènciaDeMissatges</i> .	[ resposta := (FillInTheBlank request: 'Nombre de passos' initialAnswer: '10'). resposta isAllDigits ] mentreFals: [ ].
[ unaCondició ] mentreCert: [ SeqüènciaDeMissatges ]	Executa <i>unaCondició</i> . Si és certa, executa <i>SeqüènciaDeMissatges</i> i repeteix aquest pas.  Si <i>unaCondició</i> és falsa, passar a executar la propera expressió, sense executar <i>SeqüènciaDeMissatges</i> .	[ self center y > 100 ] mentreCert: [ self ves: 10 ]

## Capítol 20

# Booleans i expressions booleans

Tal com hem mencionat en els dos capítols anteriors, les expressions condicionals i els bucles condicionals requereixen expressions que retornin un valor *cert* (true) o *fals* (false). Aquestes expressions s'anomenen expressions *booleans*, i ara les estudiarem detalladament, ja que les expressions booleans i els valors booleans són conceptes clau en programació i, de fet, en tota la informàtica. Aprendreu a escriure expressions booleans bàsiques i com combinar-les per expressar condicions més complexes. Finalment, us presentarem alguns dels errors més comuns que sorgeixen de posar malament els parèntesis.

### Valors booleans i expressions booleans

Una *Expressió booleana* és una expressió que retorna o bé *cert* o bé *fals*. Aquests valors s'anomenen booleans<sup>1</sup>. Un valor booleà *només* pot ser cert o fals. En els llenguatges de programació, els valors booleans són importants ja que són la base de l'execució condicional.

#### Valors booleans

Els valors booleans representen la veritat o la falsedat d'un enunciat. Aquí teniu alguns enunciats certs: *2 + 2 igual a 4*; *la terra completa una rotació al voltant del seu eix aproximadament cada 24 hores*. Aquí teniu alguns enunciats falsos: *La revolució francesa va acabar el 1648*; *56 < 34*. A Smalltalk, hi ha dos objectes disponibles per representar els booleans: *true* i *false*. L'objecte *true* representa

---

<sup>1</sup>La paraula *boolè* és en honor de George Boole (1815 - 1864), un lògic i matemàtic britànic que va descobrir que els enunciats matemàtics podien ser representats com a expressions simbòliques (avui dia anomenades expressions booleans) i manipulats com a objectes matemàtics.

l'enunciat “és cert”, i l’objecte `false` representa “és fals”. Els objectes `true` i `false` entenen tots els missatges que us permeten utilitzar expressions booleanes, com veureu de seguida.

## Expressions booleanes

Un expressió booleana és una expressió que retorna un objecte booleà (`true` o `false`). L’expressió `(2 > 1)` és una expressió booleana. Retorna un objecte booleà. Podeu pensar en una expressió booleana com una qüestió amb resposta cert o fals. (és 2 més gran que 1? cert!). La taula 20.1 mostra alguns exemples d’expressions booleanes i del tipus de qüestions que poden expressar. Proveu d’escriure l’expressió `Time now > (Time new hours: 8)`. Obtindreu `true` o `false`, depenent de l’hora que sigui quan ho executeu.

**Taula 20.1 — Exemples d’expressions booleanes senzilles.**

Expressió	Explicació
<code>Bot nou color = Color vermell</code>	És vermell el color del nou robot?
<code>Bot nou center = (100@153)</code>	Està el nou robot posicionat 153 píxels avall i 100 píxels a la dreta del costat esquerre de la pantalla?
<code>Time now &gt; (Time new hours: 8)</code>	És més tard de les vuit del matí?
<code>  pica  </code> <code>pica := Bot nou.</code> <code>pica ves: 100</code> <code>(Rectangle origin: 100@200</code> <code>    corner: 300@400)</code> <code>    containsPoint: pica center</code>	Està el centre del robot dins un rectangle amb cantonades esquerra superior i dreta inferior determinades per les coordenades 100@200 i 300@400?

Observeu que les dues primeres qüestions podrien ser respostes només a partir de la informació a l’expressió booleana (ja que els robots nous tenen per defecte una posició i un color). Les altres requereixen saber què ha passat, en algun *script*, abans de l’avaluació de l’expressió booleana. Avalueu i escriviu els resultats de les expressions booleanes de la taula. Després de provar cada exemple, experimenteu canviant l’expressió i comprovant les vostres prediccions del resultat.

Les expressions booleanes senzilles es basen en diversos missatges: `=`, que retorna `true` o `false` depenent de si dos objectes són iguals; `~=`, que retorna `true` o `false` depenent de si dos objectes no són iguals; i altres missatges com `>`, `<`, `>=`, `<=`, que retornen `true` o `false` depenent de si dos objectes tenen una determinada relació d’ordre.

## Combinar expressions booleanes bàsiques

Les expressions presentades dins les seccions prèvies són senzilles, i sovint no són suficients per elles mateixes. Malgrat això, poden ser combinades per expressar condicions complexes. Podem construir expressions booleanes *compostes* a partir d'expressions booleanes més senzilles utilitzant tres connectives lògiques: la *negació* (*no*, en anglès *not*), la *conjunció* (*i*, en anglès *and*) i la *disjunció* (*o*, en anglès *or*). Fixeu-vos que la negació no combina expressions booleanes, però s'acostuma a presentar juntament amb la conjunció i la disjunció.

A Smalltalk hi ha tres missatges, corresponents a les tres connectives lògiques, que permeten compondre expressions booleanes a partir d'expressions booleanes més simples: *not* per a la negació (*no*), & per a la conjunció (*i*) i | per a la disjunció. Per compondre expressions compostes, només cal enviar qualsevol d'aquests missatges a una expressió booleana, amb una altra expressió booleana com a argument en el cas de la conjunció i la disjunció. Els missatges s'utilitzen de la següent manera:

```
unaExpressióBooleana not
unaExpressióBooleana & unaAltraExpressióBooleana
unaExpressióBooleana | unaAltraExpressióBooleana
```

La taula 20.2 presenta alguns exemples d'expressions booleanes compostes. Observem detalladament com compondre expressions booleanes combinant expressions booleanes senzilles utilitzant els missatges *not*, & i |.

**Taula 20.2 — Exemples d'expressions booleanes compostes.**

Expressió	Explicació
(Bot nou color = Color vermell) not	És d'algun altre color que no sigui vermell el nou robot?
pica   pica := Bot nou.	Està el centre del robot nou a la posició 100@100 i també apuntant al nord?
(pica center = (100@100)) & (pica direccio = 90)	
Time now > (Time new hours: 8)   (Date today weekDayasString = 'Sunday')	És més tard de les vuit del matí o és diumenge (o les dues coses)?
Time now > (Time new hours: 8)   (Date today weekDayasString = 'Sunday') not	És més tard de les vuit del matí o no és diumenge (o les dues coses)?
((pica color = Color vermell) & (pica direccio = 90))   ((pica color = Color blau) & (pica direccio = 90)) not	És cert que <i>o bé</i> el color de pica és vermell i apunta cap al nord <i>o</i> el color de pica és blau i no està apuntant cap al nord (o les dues coses)?

## Negació (no)

La negació s'utilitza per expressar el contrari d'alguna cosa. A Smalltalk, la negació s'expressa utilitzant el missatge `not`, que senzillament nega l'expressió booleana a la qual s'envia. A la darrera línia de l'*Script 20.1*, el missatge `not` s'envia a l'expressió (`unBot color = Color vermell`). Si aquesta expressió és certa, la seva negació serà falsa, i viceversa. Podeu interpretar (`unBot color = Color vermell`) `not` de la següent manera: “És el robot d'algun altre color que no sigui vermell?” o “És fals que el robot sigui de color vermell?”.

**Script 20.1** Exemple de negació.

```
| unBot |
unBot := Bot nou.
unBot color: Color verd.
(unBot color = Color vermell) not
“És el robot d'algun altre color que no sigui vermell?”
```

Fixeu-vos que sempre podeu negar (invertir lògicament) una condició per passar d'una forma a l'altra. Per exemple, al Mètode 20.1 `distancia >= 200` és la negació de l'expressió `distancia < 200`, com podeu veure en el Mètode 20.2. Aquests dos mètodes tenen exactament el mateix efecte. Altre cop, us suggerim que afegiu una traça per entendre què està passant.

**Mètode 20.1** Si la distància del robot al centre no és més gran que 200 (és a dir, menor que 200), acoloreix-lo de vermell.

`vermellSiPropDelCentre`

```
| distancia |
distancia := self distanciaDesde: World center.
distancia >= 200
siFals: [ self color: Color vermell ]
```

**Mètode 20.2** Si la distància del robot al centre és menor que 200, acoloreix-lo de vermell.

`vermellSiPropDelCentre`

```
| distancia |
distancia := self distanciaDesde: World center.
distancia < 200
siCert: [ self color: Color vermell ]
```

## Conjunció (i)

El terme *conjunció* significa literalment *ajuntar-se*. La conjunció s'utilitza per indicar que una expressió booleana composta *expressio1 & expressio2* és certa només quan *totes dues* expressions booleanes *expressio1* i *expressio2* són certes. A Squeak, una conjunció es defineix enviant el missatge binari **&** a una expressió booleana amb una altra expressió booleana com a argument. Altre cop, una conjunció es certa només quan les dues subexpressions que la componen són certes. Si alguna, o les dues, és falsa, aleshores l'expressió composta és falsa. A la taula 20.3, l'expressió composta serà certa només si *totes dues* (*unBot center = 100@100*) i (*unBot direccio = 90*) són certes.

**Taula 20.3 — Exemple de conjunció.**

Expressió	Explicació
( <i>unBot center = 100@100</i> ) & ( <i>unBot direccio = 90</i> )	Éstà el robot a la posició 100@100 <i>i</i> apuntant cap al nord?

## Disjunció (o)

La disjunció s'utilitza per expressar la idea d'alternativa. Penseu-ne alguna: voleu cafè o te? pastís o gelat? Una disjunció es defineix enviant el missatge **|** a una expressió booleana amb una altra expressió booleana com a argument. Una disjunció s'utilitza per expressar que voleu que *almenys una* de les expressions booleanes sigui certa. Per tant una disjunció és certa quan una o dues de les expressions booleanes que la componen són certes.

Aquesta definició de disjunció és sovint motiu de confusió. Això és degut al fet que la paraula “o” (utilitzada per expressar disjuncions) s'utilitza de *dues maneres diferents*. Quan sou sopant a casa d'en Fred, ell pot preguntar “Què prefereixes, cafè o te?”, esperant la resposta “cafè” o o la resposta “te”, o també podeu dir “no res, gràcies”. Però si pregunta “Què t'agradaria més, gelat o pastís?”, probablement espera que respongueu “pastís”, “gelat”, o “tots dos, si us plau!”. Quan “o” significa una opció o l'altra, però no les dues, l'anomenem *o exclusiu*. Quan “o” significa una opció o l'altra, o les dues, l'anomenem *o inclusiu*. A la taula 20.4, l'expressió composta serà certa si l'expressió (*unBot center = 100@100*) és certa *o* l'expressió (*unBot direccio = 90*) és certa, o si *totes dues* són certes.

**Taula 20.4 — Exemple de disjunció.**

Expressió	Explicació
( <i>unBot center = 100@100</i> )   ( <i>unBot direccio = 90</i> )	Éstà el robot a la posició 100@100 <i>o</i> està apuntant cap al nord ( <i>o</i> les dues coses)?

## Tot junt

Els dos darrers exemples de la taula 20.2 mostren que podeu combinar expressions booleanes moltes vegades, negar-les, i agrupar-les amb conjuncions (*i*) i disjuncions (*o*) per representar condicions complexes.

## Alguns aspectes d'Smalltalk

Recordeu que les classes són fàbriques per produir objectes. Quan un robot en particular és creat per la classe Bot, diem que el robot és una instància de la classe Bot. A Smalltalk, els valors booleanos també són objectes. L'objecte true és una instància de la classe True, que defineix el comportament del valor booleà true. Igualment, false és una instància de la classe False, que defineix el comportament del valor booleà false. Fixeu-vos que fins i tot si true i false són objectes en el mateix sentit que el robot creat per la classe Bot és un objecte, són tan importants per Smalltalk que true i false són objectes especials. Per tant, no els heu de crear utilitzant new. true i false existeixen contínuament, i no us heu d'amoïnar per la seva creació. Fixeu-vos que els objectes booleanos true i false comencen amb lletra minúscula.

Com ja hem explicat a la primera secció d'aquest capítol, quan avaluueu una expressió com Time now > (Time new hours: 8) obteniu l'objecte booleà true o l'objecte booleà false, depèn de l'hora en què executeu l'expressió. També vam dir que per a compondre expressions booleanes, els missatges not, & i | s'envien a expressions booleanes, i el resultat d'una expressió booleana és o bé true o bé false. Per tant, els missatges not, & i | són mètodes definits a les classes True i False, que fabriquen els objectes true i false.

La taula 20.5 mostra com s'utilitzen els tres operadors booleanos.

## Oblidar parèntesis (un error freqüent)

De tant en tant podeu tenir problemes amb la sintaxi d'Smalltalk. Tots els programadors, fins i tot els més experimentats, tenen aquests problemes en algun moment. La diferència entre un novell i un programador experimentat no és que un comet errors i l'altre no. La diferència principal és que un programador experimentat és molt més hàbil trobant i corregint els errors.

Oblidar-se de posar parèntesis és una font freqüent d'errors, tant lògics com sintàctics, per tant ara us ensenyarem com analitzar els errors que possiblement cometreu en algun moment. Bàsicament, quan esteu component, heu d'identificar clarament l'expressió a què els missatges not, & i | són enviats. Analitzem un cas particular.

**Taula 20.5 — Exemple de disjunció.**

<b>Operador</b>	<b>Missatge</b>	<b>Exemples</b>	<b>Resultat</b>
Negació (no)	not	<i>expressioFalsa</i>	true
		<i>expressioCerta</i>	false
		(Bot nou color = Color vermell) not	true
Conjunció (i)	&	<i>expressioCerta &amp; expressioCerta</i>	true
		<i>expressioFalsa &amp; expressioCerta</i>	false
		<i>expressioCerta &amp; expressioFalsa</i>	false
		<i>expressioFalsa &amp; expressioFalsa</i>	false
		(unBot center = 100@100) & (unBot direcció = 90)	true o false
Disjunció (o)		<i>expressioCerta   expressioCerta</i>	true
		<i>expressioFalsa   expressioCerta</i>	true
		<i>expressioCerta   expressioFalsa</i>	true
		<i>expressioFalsa   expressioFalsa</i>	false
		Time now > (Time new hours: 8)   (Date today weekDay asString = 'Sunday')	true o false

## Cas d'estudi

L'*Script* 20.2 mostra una expressió booleana que falla en representar la qüestió següent: “És el color d'un robot nou diferent del color vermell?” Per veure que executar aquest *script* ens duu a un error, executeu l'expressió descrita a l'*Script* 20.2, obriu el depurador quan aparegui l'error, i seleccioneu la primera línia de la subfinestra superior per obtenir la figura 20.1.

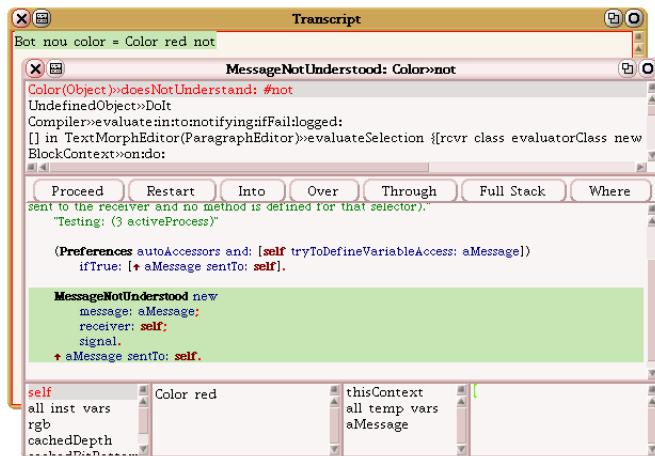
**Script 20.2** *Oblidar parèntesis provoca una identificació incorrecte del receptor d'un missatge not.*

Bot nou color = Color red not

## Utilitzar el depurador

El títol a la finestra del depurador ja us dóna informació: MessageNotUnderstood: Color »not. Això ens està dient que l'objecte color (un objecte creat per la classe Color) no entén el missatge not.

Ara, quan seleccioneu la línia de dalt de tot a la subfinestra superior, veieu el cos del mètode doesNotUnderstand: a la segona subfinestra. Quan feu clic a self a la subfinestra de baix de tot a l'esquerra, veieu (a la segona subfinestra de baix de tot) que el receptor no és un booleà, com hauria de ser, sinó un color, Color red. Si feu clic a aMessage a la subfinestra inferior dreta, veureu quin missatge no ha estat entès. En el nostre cas, veureu not, el que significa que el missatge not



**Figura 20.1** — El missatge `not` no és enviat a l'expressió booleana completa `Bot nou color = Color red`, sinó que és enviat a l'expressió `Color red`, que retorna un objecte color. Però els objectes color no entenen el missatge `not`, i per tant es genera un error.

no ha estat enviat al receptor correcte, ja que ha estat enviat al resultat de l'expressió `Color red`, que és un objecte color, i aquest no entén el missatge `not`.

## Entendre el problema

La raó per la qual el missatge `not` ha estat enviat a l'expressió `Color red` i no a l'expressió booleana completa està relacionada amb la manera que té Smalltalk d'executar expressions, com vam explicar al capítol 11. Recordeu que primer s'executen les expressions envoltades per parèntesis, després els missatges unaris, deprés els binaris i finalment els missatges de paraula-clau. En el nostre cas, el missatge `not` és un missatge unari. Per tant, és avaluat abans que el missatge binari `=`, i així és enviat al resultat de l'expressió `Color red`. Per aconseguir l'ordre d'execució correcte, hem d'envoltar l'expressió adequada amb parèntesis, com veieu a l'*Script* 20.3. Ara el missatge `not` serà enviat al resultat del missatge `=`.

L'ordre d'execució dels missatges en l'*Script* 20.2 incorrecte és el següent: L'expressió `Bot nou color = Color red not` s'executa com si hagués estat escrita tota ella amb parèntesis de la manera següent: `((Bot nou) color) = ((Color red) not)`. Per tant, primer s'avaluen les dues parts del mètode binari `=`, és a dir, l'expressió `((Bot nou) color)`, que retorna el color d'un robot nou, i l'expressió `((Color red) not)`. L'execució de l'expressió `((Color red) not)` avalia primer `Color red`,

que retorna un objecte color, i després el missatge `not` és enviat a aquest objecte color, que genera l'error.

Per obtenir el comportament desitjat, l'expressió s'hauria d'escriure entre parèntesis, de manera que el missatge `not` sigui enviat al resultat del missatge `=`. L'expressió correcte es mostra a l'*Script 20.3*.

**Script 20.3** *Utilitzar parèntesis ajuda a estar segur que el missatge s'ha escrit correctament.*

```
(Bot nou color = Color red) not
```

Aquesta expressió s'executa de la següent manera: primer, s'avaluen les dues parts del mètode binari `=`. La primera retorna un objecte color que representa el color del robot nou, mentre la segona retorna un objecte color vermell (`red`). Aquests dos objectes color o bé són iguals o bé són diferents. Després el missatge `=` s'executa, i envia el resultat de l'expressió a la dreta del `=` a l'objecte color associat al color del robot. L'execució del missatge `=` retorna un booleà, a què s'envia el missatge `not`.

Si mai no esteu segurs de l'ordre en què s'executen els missatges, hauríeu d'utilitzar parèntesis, ja que les expressions entre parèntesis s'executen abans que qualsevol altre tipus d'expressió. Per tant, podeu posar expressions entre parèntesis per assegurar-vos que els missatges s'executaran en l'ordre lògic correcte.

## Problemes semblants i solucions

Seria tediós provar d'examinar tots els problemes similars que podeu trobar amb les expressions booleanes. Proveu d'executar els *Scripts 20.4* i *20.6* i esbrinar si podeu entendre què hi ha d'incorrecte en aquests *scripts*. Les expressions correctes les trobareu als *Scripts 20.5* i *20.7*.

**Script 20.4** *Els parèntesis que falten porten a identificar incorrectament el receptor de &.*

```
| unBot |
unBot := Bot nou.
unBot center = 100@100 & unBot midaLlapis = 5
```

**Script 20.5** *Utilitzem parèntesis per aconseguir enviar el missatge & al receptor adequat.*

```
| unBot |
unBot := Bot nou.
(unBot center = 100@100) & (unBot midaLlapis = 5)
```

**Script 20.6** Els parèntesis que falten porten a identificar incorrectament el receptor de |.

```
| unBot |
unBot := Bot nou.
unBot center = 100@100 | unBot direccio = 90
```

**Script 20.7** Utilitzem parèntesis per aconseguir enviar el missatge | al receptor adequat.

```
| unBot |
unBot := Bot nou.
(unBot center = 100@100) | (unBot direccio = 90)
```

## Resum

- Els valors booleans d'Smalltalk són true i false. El valor true representa un enunciat cert, i false un enunciat fals.
- Les expressions booleanes són expressions que retornen valors booleans.
- Les expressions booleanes compostes es componen a partir d'expressions booleanes simples i conjuncions (i), disjuncions (o) i negacions (no).

La taula següent repassa les tres operacions booleanes.

Operador	Missatge	Exemples	Resultat
Negació (no)	not	<i>expressioFalsa</i>	true
		<i>expressioCerta</i>	false
		(Bot nou color = Color vermell) not	true
Conjunció (i)	&	<i>expressioCerta &amp; expressioCerta</i>	true
		<i>expressioFalsa &amp; expressioCerta</i>	false
		<i>expressioCerta &amp; expressioFalsa</i>	false
		<i>expressioFalsa &amp; expressioFalsa</i>	false
		(unBot center = 100@100) & (unBot direccio = 90)	true o false
Disjunció (o)		<i>expressioCerta   expressioCerta</i>	true
		<i>expressioFalsa   expressioCerta</i>	true
		<i>expressioCerta   expressioFalsa</i>	true
		<i>expressioFalsa   expressioFalsa</i>	false
		Time now > (Time new hours: 8)   (Date today weekDay asString = 'Sunday')	true o false

## Capítol 21

# Coordenades, punts i moviments absoluts

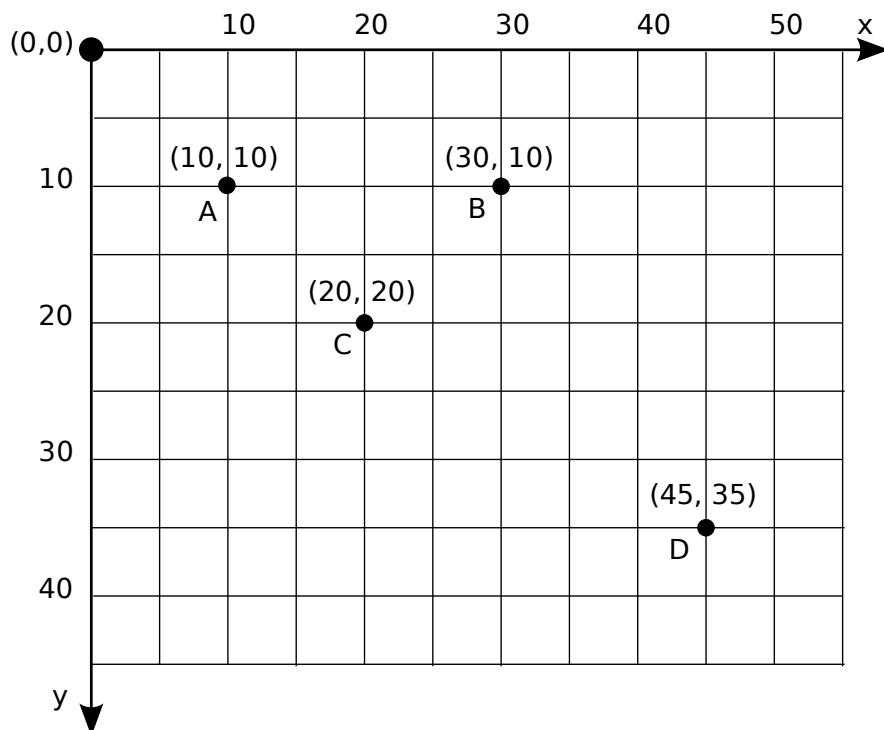
Fins ara, els missatges enviats a un robot demanant-li que es mogués eren instruccions relatives a la posició actual del robot. Per exemple, l'expressió pica ves: 100 li diu a pica que vagi endavant 100 píxels a partir de la seva posició *actual* i la seva direcció *actual*. Aquest moviment s'anomena *relatiu* ja que la posició assolida pel robot quan ha acabat de moure's depèn de la seva posició inicial. Aquest tipus de moviment és molt útil, com ja hem vist, però de vegades ens agradaria ser capaços de dir-li al robot que anés a un lloc específic de la pantalla, com ara el centre de la pantalla. Aquest moviment, en què el robot acaba en una posició determinada independentment d'on ha sortit, s'anomena *absolut*. Per fer moviments absoluts ens cal un sistema de coordenades, que és una manera de representar un lloc concret a la pantalla.

Ja us heu trobat amb aquest sistema de coordenades en capítols anteriors, però ara és moment d'estudiar-lo detalladament. Podeu haver trobat sistemes de coordenades a les assignatures de matemàtiques. De fet, esteu utilitzant un sistema de coordenades quan busqueu un carrer en un mapa. El llistat de carrers en un mapa pot indicar que el carrer que busqueu és en un quadrat identificat per una lletra i un nombre, o per dues lletres i dos nombres. El mateix es pot aplicar a la pantalla de l'ordinador. Podeu referir-vos a un punt a la pantalla donant un parell de nombres, un per a la direcció horitzontal i un altre per a la direcció vertical.

En aquest capítol us presentarem les definicions precises de punt i coordenada. Després introduirem nous comportaments dels robots i us oferirem alguns experiments per tal que els proveu de fer. Entendre els punts i les coordenades us ajudarà a explorar problemes nous en el futur, tal com utilitzar robots per simular alguns tipus de comportament animal.

## Punts

Recordem que tot és un objecte a Smalltalk, en particular, els *punts* a la pantalla també són objectes. Els punts són creats per la classe Point, i el seu comportament és similar al dels punts matemàtics amb què segurament ja esteu familiaritzats. En dues dimensions, un punt es compon de dues coordenades: la coordenada *x*, per a la direcció horitzontal, i la coordenada *y* per a la vertical. Un punt es crea enviant el missatge @ a un nombre amb un altre nombre com a argument. Per exemple, el punt D a la figura 21.1 és creat per l'expressió 45@35. La seva coordenada *x* és 45 i la seva coordenada *y* és 35.



**Figura 21.1** — A Smalltalk, un punt representa un lloc de l'espai en dues dimensions. Un punt té una coordenada (horitzontal) *x* i una coordenada (vertical) *y*.

---

**Important!** 200@400 és un punt amb 200 com a coordenada *x* i 400 com a coordenada *y*.

---

L'*Script* 21.1 mostra com accedir als components individuals d'un punt.

**Script 21.1** *Accedir als components d'un punt.*

```
| punt1 |
punt1 := 45@35.
punt1 x
->45
punt1 y
->35
```

Val la pena que us fixeu que el sistema de coordenades d'Smalltalk no és ben bé el sistema de coordenades matemàtic. La figura 21.1 mostra que, en contrast amb el model matemàtic habitual, la direcció positiva de l'eix *y* comença a la part de dalt de la pantalla i creix cap a baix. Tot i així, l'eix *x* s'incrementa d'esquerra a dreta, com en la notació matemàtica normal. Direm que en Squeak l'*origen* de coordenades és a la cantonada superior esquerra. Així, el punt 45@35 està a 45 píxels a la dreta del costat esquerre de la pantalla i 35 píxels cap avall. Els punts amb alguna coordenada negativa (o les dues) estan localitzats en algun lloc fora de la pantalla.

**Important!** El sistema de coordenades d'Smalltalk té el seu origen (0,0) a la cantonada superior esquerra de la pantalla, i la direcció positiva de l'eix *y* és cap avall, des de dalt de la pantalla fins a baix de tot.

Totes les operacions matemàtiques usuals estan disponibles pels punts. En aquest capítol us presentarem només les operacions que farem servir més endavant. Per exemple, podeu multiplicar un punt per un nombre per obtenir un punt les coordenades del qual són les del punt inicial multiplicades pel nombre. Així,  $(100@200) * 3$  dóna com a resultat el punt  $(300@600)$ . També podeu fer servir la suma i la resta de punts, on se sumen o es resten les coordenades, per tant,  $(100@200) + (40@360)$  dóna com a resultat  $(140@560)$ . Fixeu-vos que les operacions binàries com \* i + poden operar amb punts per crear punts nous. L'*Script* 21.2 mostra aquestes operacions.

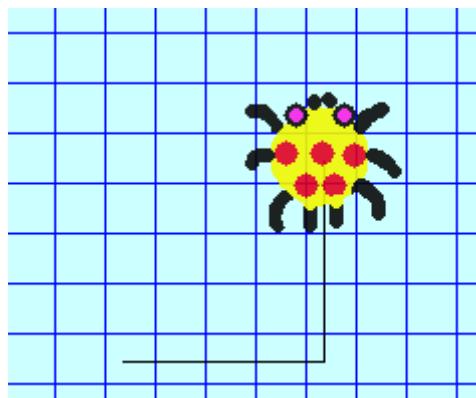
**Script 21.2** *Manipular punts per crear més punts.*

```
| punt1 punt2 punt3 |
punt1 := 200@400.
punt2 := punt1 * 2.
punt2
- Escriure el valor retornat: 400@800
punt2 x
- Escriure el valor retornat: 400
```

```
punt2 y
– Escriure el valor retornat: 800
punt3 := (50@60) + punt1.
punt3 x
– Escriure el valor retornat: 250
punt3 y
– Escriure el valor retornat: 460
punt1 + 85      “85 és una abreviació pel punt 85@85”
– Escriure el valor retornat: 285@485
```

## Utilitzar quadrícules

Per ajudar-vos a entendre els punts, podeu fer una ullada a la informació mostrada dins de les bafarades de text que apareixen quan el ratolí s'atura un instant a sobre d'un robot. També podeu fer servir una quadrícula. Squeak pot dibuixar una quadrícula al fons de la pantalla. Per aconseguir-ho, obriu el menú del Món (**World**), trieu l'opció **aspecte...**, i després l'opció **tria textura de fons o fes la teva pròpia textura ....**. Triar aquesta darrera opció us permet triar la mida i el color de la quadrícula.



**Figura 21.2 — Una quadrícula de mida 25 pixels.**

Com veieu a l'*Script 21.3* i s'il·lustra a la figura 21.2, també podeu programar la quadrícula amb els mètodes següents: `drawGrids` i `unDrawGrids` per dibuixar i eliminar quadrícules, `gridColor:` `unColor` per canviar el color d'una quadrícula, `gridSize:` `unEnter` per especificar la mida d'una

quadrícula (el nombre de píxels del costat de cada quadrat), gridWorldColor: unColor per canviar el color del fons d'una quadrícula, i worldColor: unColor per canviar el color del Món. També podeu recuperar la mida d'una quadrícula amb gridSize.

### **Script 21.3 Crear una quadrícula**

```
| entorn |
entorn := BotEnvironment default.
entorn gridSize: 25.
entorn gridWorldColor: Color blauPàlid.
entorn gridColor: Color blau.
entorn drawGrids
```

Utilitzant el menú **world**, podeu canviar la mida de la pantalla. Per anar a mode pantalla completa, obriu el menú del Món, trieu l'opció **aspecte...**, i canvieu de pantalla completa a pantalla normal o viceversa. També podeu utilitzar els mètodes fullScreenOff i fullScreenOn com veieu a l'*Script 21.4*.

### **Script 21.4 Canviar la mida de la pantalla**

```
BotEnvironment default fullScreenOn.
```

## **Una font d'errors amb els punts**

La manera de crear punts i l'ordre en què els missatges s'executen pot causar errors, com podeu veure a la primera línia de l'*Script 21.5*. L'enviament de missatge 50@60 + 200@400 retorna aB3dVector en lloc d'un punt representant la suma dels dos punts. El problema és que, degut a l'ordre d'execució dels missatges, un *punt*, i no un *enter*, és el receptor del missatge @, i en aquest cas retorna un altre tipus de punt (un vector 3D) que no ens interessa ara. Explicarem exactament què ha anat malament en el proper paràgraf. En tot cas, segur que no hem obtingut el resultat que volíem! Una altra vegada, us recordem que s'ha de parar molta atenció a l'ordre d'enviament de missatges i que feu servir els parèntesis si és necessari.

### **Script 21.5 Un possible error amb els punts**

```
50@60 + 200@400      "retorna un 3D vector, no un punt"
-> a B3DVector3(250.0 260.0 400.0)
(50@60) + (200@400)
-> 250@460
```

---

**Nota** Per evitar problemes amb els punts, envolteu-los amb parèntesis quan participin en operacions complexes.

---

Recordeu del capítol 11 l'ordre en què els missatges s'executen, i en particular el següent:

- Les expressions dins de parèntesis () s'avaluen primer.
- Els missatges unaris s'executen abans que els binaris, i els binaris s'executen abans que els missatges de paraula clau.
- Els missatges del mateix tipus s'avaluen d'esquerra a dreta.

El mètode @ és un mètode binari com qualsevol altre, i té la mateixa prioritat que altres mètodes binaris com +, \* i =. Per tant, l'expressió 50@60 + 200@400 s'executa com si s'hagués escrit de la següent manera: (((50@60) + 200)@400). El segon missatge @ acabarà sent enviat a un punt i no a un enter. Anem a veure què passa a la primera línia de l'*Script* 21.5, que no retorna el punt 250@260, com caldria esperar.

### Descompondre 50@60 + 200@400

Tal com hem mencionat, 50@60 + 200@400 és equivalent a (((50@60) + 200)@400).

**Pas 1.** @ és enviat a 50 amb argument 60. Es retorna el punt 50@60.

**Pas 2.** + és enviat al punt 50@60 amb argument 200. El punt 250@260 és retornat, ja que quan un nombre es passa a un punt com a argument, es considera com a un punt tenint el mateix nombre com a coordenada *x* i com a coordenada *y*, aquí 200@200.

**Pas 3.** @ és enviat a 250@260 amb 400 com a argument, i l'objecte B3DVector3(250.0 260.0 400.0) és retornat.

Si ara senzillament posem parèntesis al voltant dels dos punts, obtenim un punt que n'és la suma.

### Descompondre (50@60) + (200@400)

**Pas 1.** Expressions entre parèntesis s'avaluen primer.

**Pas 1.1.** @ s'envia a 50 amb 60 com a argument i es retorna el punt 50@60.

**Pas 1.2.** @ s'envia a 200 amb 400 com a argument i es retorna el punt 200@400.

**Pas 2.** `+` és enviat a `50@60` amb argument `200@400` i es retorna el nou punt `250@460`.

El missatge final és: poseu parèntesis al voltant dels punts quan els manipuleu.

## Moviments absoluts

Ara que ja podem especificar un lloc a la pantalla, podem dir-li a un robot que vagi directament a aquell lloc. Per fer això, tenim els mètodes `vesA: unPunt` i `saltaA: unPunt`.

- Enviar el missatge `vesA: unPunt` a un robot li diu d'anar al lloc representat pel punt.
- Enviar el missatge `saltaA: unPunt` a un robot li diu de saltar al lloc representat pel punt.

Fixeu-vos que els missatges `salta:` i `saltaA:` no deixen cap traça, mentre `ves:` i `vesA:` sí que ho fan. Anem a practicar. Proveu d'endevinar què fa l'*Script 21.6*. Després mireu d'estimar la mida de la vostra pantalla en píxels posant un robot tan a prop com pugueu de la cantonada inferior dreta.

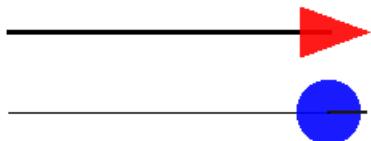
**Script 21.6** *Anar i saltar directament a un lloc de la pantalla.*

```
| pica |
pica := Bot nou.
pica vesA: 200@400.
pica saltaA: 300@400.
pica ves: 1.
pica saltaA: 400@400.
pica vesA: 450@400.
```

La secció següent posarà èmfasi en les diferències entre els mètodes `ves:` `unaDistancia` i `vesA:` `unPunt`, i `salta:` `unaDistancia` i `saltaA:` `unPunt`.

## Moviment relatiu versus moviment absolut

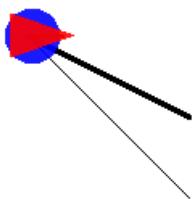
Ara estudiarem detalladament la diferència entre els mètodes `ves:` `unaDistancia` i `vesA:` `unPunt`. El mètode `ves:` li diu al robot que es mogui una certa distància *seguint la seva direcció actual*. Així, on anirà a parar el robot depèn de la seva posició actual i la seva direcció actual. L'*Script 21.7* il·lustra això.

**Script 21.7** *Moviment paral·lel*

```
| pica marge |
pica := Bot nou.
marge := Bot nou.
marge aparentaTriangle.
pica aparentaCercle.
marge color: Color vermell.
marge midaLlapis: 3.
marge nord.
marge salta: 50.
marge est.
pica ves: 200.
marge ves: 200.
```

Com podeu veure, els dos robots es mouen en línies paral·leles i no acaben al mateix lloc, encara que tots dos han rebut el mateix missatge ves: 200 al final de l'script.

En canvi, el mètode vesA: *unPunt* li diu al robot de moure's a un lloc fix *no importa* la seva posició i direcció abans del moviment. Això està il·lustrat a l'Script 21.8.

**Script 21.8** *Moviment convergent*

```
| pica marge |
pica := Bot nou.
marge := Bot nou.
marge aparentaTriangle.
```

```
pica aparentaCercle.
```

```
marge color: Color vermell.
```

```
marge midaLlapis: 3.
```

```
marge nord.
```

```
marge salta: 50.
```

```
marge est.
```

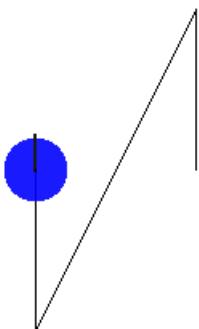
**pica vesA: World center - 100.**

**marge vesA: World center - 100.**

En aquest cas, els dos robots acaben al mateix lloc. Diem que el mètode `ves:` produeix un moviment *relatiu*, mentre que el mètode `vesA:` produeix un moviment *absolut*. A l'script hem utilitzat l'expressió `World center - 100` per obtenir exactament el mateix dibuix que hem reproduït aquí, fins i tot si el vostre monitor té una resolució diferent del monitor utilitzat per escriure aquest llibre.

Finalment, fixeu-vos que els mètodes `ves:` i `vesA:` no canvien la direcció del robot. Això ho podeu veure a l'Script 21.9. En aquest script diem a `pica` que es mogui endavant 100 píxels a partir de la seva posició actual, després que vagi directament a una posició localitzada a distància (100, 100) del centre de la pantalla, i després que es mogui endavant 100 píxels en la seva direcció actual.

#### Script 21.9 Combinar moviments absoluts i relatius



```
| pica |
```

```
pica := Bot nou.
```

```
pica aparentaCercle.
```

```
pica nord.
```

```
pica ves: 100.
```

pica vesA: (World center - (100@-100)).

pica ves: 100.

## Alguns experiments

Aquí teniu alguns experiments per provar, amb la idea que us familiaritzeu més amb els conceptes presentats en aquest capítol. Com podeu veure, el robot no canvia de direcció quan és transportat a una posició donada utilitzant vesA:

### Experiment 21-1 (rectangle 1)

Utilitzant els mètodes vesA: i saltaA: definiu un mètode rectangleSuperiorEsquerra: punt1 inferiorDreta: punt2 que dibuixa un rectangle. Un enviament de missatge hauria de ser similar a pica rectangleSuperiorEsquerra: 200@500 inferiorDreta: 350@700.

---

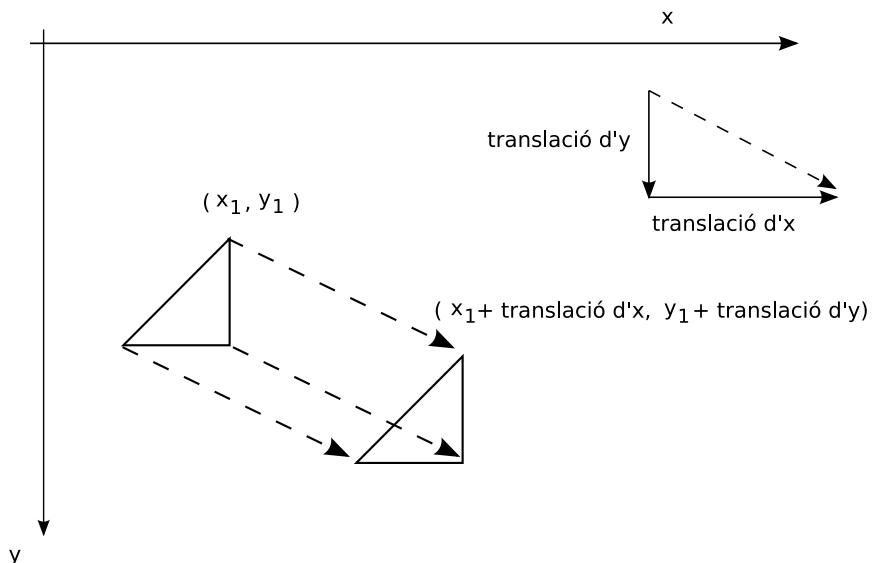
### Experiment 21-2 (rectangle 2)

Definiu un altre mètode rectangleOrigen: punt1 area: punt2 en què el segon punt ja no representa la cantonada oposada sinó la mida (base i altura) del rectangle. Per exemple pica rectangleOrigen: 200@600 area: 350@500 dibuixa un rectangle amb la cantonada superior esquerra 200@600, amb base de mida 350 pixels i altura 500.

---

## Translació

Si movem cada punt d'una figura geomètrica la mateixa distància en la mateixa direcció, obtenim la mateixa figura, però en una altra posició. Aquesta operació s'anomena *translació* en matemàtica. Com veieu a la figura 21.3, una translació pot ser pensada com un moviment en la direcció  $x$  d'una determinada quantitat i un moviment en la direcció  $y$  d'una altra quantitat. Naturalment, aquests nombres no han de ser els mateixos. Per tant, podem representar la translació d'una figura com la suma d'un vèrtex i un "punt de translació", representant la mida de la translació en les direccions  $x$  i  $y$ . Així, el vèrtex de la nova figura és igual a la suma del vèrtex de la figura original i el punt de translació. A la figura 21.3, el vertex  $(x_1, y_1)$  és traslladat afegint un punt de translació, i el resultat és un punt la coordenada  $x$  del qual és  $x_{nou} = x_1 + \text{translació } x$  i la coordenada  $y$  del qual és  $y_{nou} = y_1 + \text{translació } y$ . Per exemple, el punt 200@300 traslladat pel punt de translació 50@75 és 250@375.

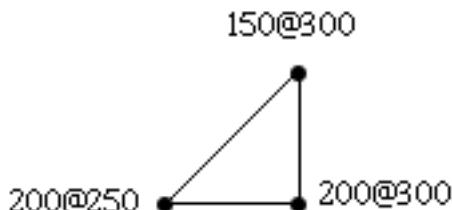


**Figura 21.3** — Traslladar una figura requereix afegir el mateix valor  $x$  i valor  $y$  a cada punt de la figura.

### Experiment 21-3 (triangle 1)

Definiu un mètode triangleA: primerPunt punt2: segonPunt punt3: tercerPunt que dibuixa un triangle amb els tres arguments com a vertexs del triangle. L'Script 21.10 il·lustra com utilitzar aquest mètode.

**Script 21.10** Utilitzar triangleA: primerPunt punt2: segonPunt punt3: tercerPunt



| pica |  
pica := Bot nou.

pica

triangleA: 200@300

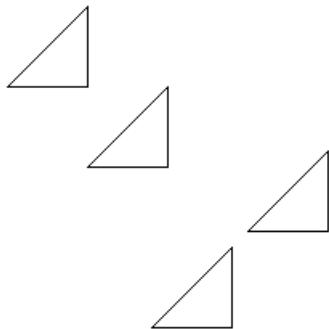
punt2: 200@250

punt3: 150@300

## Traslladar triangles

Ara que teniu un mètode per dibuixar triangles, podeu dibuixar diversos triangles idèntics senzillament traslladant-ne un de dibuixat primer. A l'*Script 21.11*, definim tres translacions i dibuixem els triangles corresponents.

**Script 21.11 Utilitzar triangleA: primerPunt punt2: segonPunt punt3: tercerPunt**



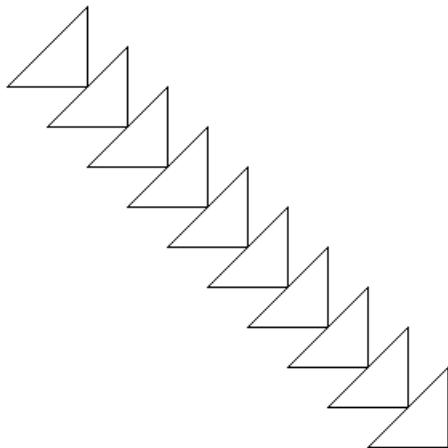
```
| pica primerPunt segonPunt tercerPunt t1 t2 t3 |
primerPunt := 200@300.    "Tres punts d'un triangle"
segonPunt := 200@250.
tercerPunt := 150@300.
t1 := 50@50.                "Tres punts per traslladar un triangle"
t2 := 90@150.
t3 := 150@90.
pica := Bot nou.
pica tornarInvisible.
pica triangleA: primerPunt punt2: segonPunt punt3: tercerPunt.
pica triangleA: primerPunt + t1 punt2: segonPunt + t1 punt3: tercerPunt + t1.
pica triangleA: primerPunt + t2 punt2: segonPunt + t2 punt3: tercerPunt + t2.
pica triangleA: primerPunt + t3 punt2: segonPunt + t3 punt3: tercerPunt + t3.
```

També podríem haver definit un mètode triangleA:punt2:punt3:translacio: que dugués a terme la translació, així no caldría que nosaltres féssim la suma de punts. Aquesta solució és més segura ja que d'aquesta manera no podríem aplicar diferents translacions a diferents punts del mateix triangle, i per tant us suggerim que la implementeu.

## Oques volant

Podem repetir l'operació de translació per obtenir patrons repetits. L'*Script 21.12* genera un patró que ens recorda a *oques volant*. Fixeu-vos que triem la translació de maneira que cada triangle toqui el següent i així estiguin col·locats en diagonal.

### Script 21.12 Oques volant



```
| pica translacio primerPunt segonPunt tercerPunt |
primerPunt := 200@300.
segonPunt := 200@250.
tercerPunt := 150@300.
translacio := 25@25.
pica := Bot nou.
10 vegadesRepetir:
    [ pica triangleA: primerPunt punt2: segonPunt punt3: tercerPunt.
      primerPunt := primerPunt + translacio.
      segonPunt := segonPunt + translacio.
      tercerPunt := tercerPunt + translacio ].
```

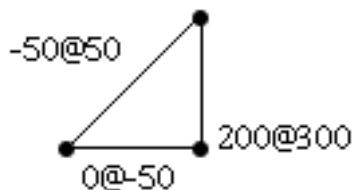
L'*Script* 21.13 us mostra com podeu escriure la translació d'una manera més concisa utilitzant el fet que els punts poden ser multiplicats, a més de sumats.

### **Script 21.13 Oques volant**

```
| pica vegades |
pica := Bot nou.
10 vegadesRepetir:
  [ pica triangleA: 200@300
    punt2: 200@250
    punt3: 150@300
    translacio: (25@25) * vegades.
    vegades := vegades + 1 ].
```

### Experiment 21-4 (triangle 2)

Com a variació de l'Experiment 21-3, definiu un mètode anomenat triangleA: unPunt delta1: unPunt1 delta2: unPunt2 que dibuixa un triangle començant al punt unPunt i utilitza els dos altres arguments com la diferència entre un altre punt del triangle i el primer punt. Així, triangleA: 200@300 delta1: 0@50 delta2: -50@50 dibuixa el mateix triangle que triangleA: 200@300 punt2: 200@250 punt3: 150@300.



## Utilitzar moviments absoluts

Us podeu preguntar per quina raó parem tanta atenció als punts. Fins ara, cap dels dibuixos que hem fet requereixen punts. De fet, executar la majoria d'aquests dibuixos utilitzant punts hagués estat difícil. Imagineu-vos provar de dibuixar un pentàgon utilitzant només el missatge vesA:. Malgrat tot, els moviments absoluts són útils. Els exemples que segueixen ho il·lustren.

Utilitzarem un punt per desar la posició del robot en diferents moments de l'execució d'un dibuix complex. Després utilitzarem aquesta posició per continuar el dibuix.

El primer exemple està basat en l'*Script 3.4* del capítol 3, en què dibuixàvem una lletra A. En aquell capítol vam veure que aquest *script* dibuixa dues vegades la meitat inferior de la barra vertical dreta de la lletra A. En aquell moment no ho vam considerar un problema massa gran. Imagineu, però, una situació en la qual dibuixar una línia fos molt costós, ja sigui en temps de càlcul o en tinta d'impressora. En aquest cas, valdria la pena estalviar-nos dibuixar una línia dues vegades. La nostra solució és utilitzar un punt per desar la posició del robot al costat esquerre de la barra horitzontal i tornar a aquest punt per dibuixar la barra després d'haver dibuixat la resta de la lletra.

L'*Script 21.14* dibuixa una lletra A com a l'*Script 3.4*. Després l'hem modificat per obtenir l'*Script 21.15*, que dibuixa l'A sense dibuixar a sobre d'una línia ja dibuixada, utilitzant un salt absolut.

**Script 21.14** *Dibuixar la lletra A passant dos cops per la mateixa línia.*

```
| pica |
pica := Bot nou.
pica giraEsquerra: 90.
pica ves: 100.
pica giraDreta: 90.
pica ves: 100.
pica giraDreta: 90.
pica ves: 100.
pica giraDreta: 180.
pica ves: 40.
pica giraEsquerra: 90.
pica ves: 100.
```

**Script 21.15** *Dibuixar la lletra A utilitzant un salt a un punt absolut.*

```
| pica punt |
pica := Bot nou.
pica giraEsquerra: 90.
pica ves: 40.
punt := pica center.
pica ves: 60.
pica giraDreta: 90.
pica ves: 100.
```

pica giraDreta: 90.

pica ves: 100.

**pica saltaA: punt.**

pica giraEsquerra: 90.

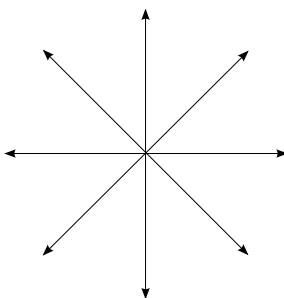
pica ves: 100.

A l'*Script* 21.15, la barra vertical de l'esquerra de l'A es dibuixa en dos passos, primer 40 i després 60 píxels. Entre aquests passos, la posició absoluta del robot s'obté amb el mètode center, i aquesta posició és desada a la variable punt. Aquesta és la posició on la barra de l'A s'hauria de dibuixar. Després de dibuixar la darrera barra vertical, el robot salta a la posició de la barra amb el mètode saltaA: punt i dibuixa la barra horitzontal.

Podeu verificar que la lletra A és dibuixada correctament en qualsevol angle afegint un enviament de missatge giraEsquerra: o giraDreta: qualsevol nombre de graus, després de la creació del robot.

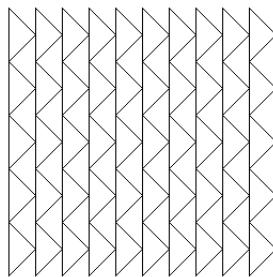
### Experiment 21-5 (fletxes)

Utilitzant la mateixa tècnica que amb l'*Script* 21.15, feu un *script* que generi vuit fletxes sortint de l'origen del robot, en vuit direccions diferents, com veieu a la figura de més avall. Ajuda: Primer definiu un mètode anomenat fletxa: unPunt que dibuixa una fletxa apuntant en la direcció actual començant en el punt donat. Després utilitzeu una variable addicional per recordar l'origen de la fletxa. Un cop hagueu escrit aquest mètode i us funcioni correctament, us suggerim que resolgueu el mateix problema utilitzant els mètodes ves: i salta:. D'aquesta manera, realment entendreu la diferència entre les dues maneres d'expressar el mateix problema.



## Bucles i translacions

Abans de continuar llegint, intenteu definir un *script* que dibuixi la figura 21.4.



**Figura 21.4 — Un patró d'oques volant.**

La nostra solució apareix a l'*Script 21.16*, que dibuixa la figura 21.4. Els punts defineixen un cert nombre de mètodes útils, com per exemple negated i setX:setY::

- El mètode negated enviat a un punt retorna un altre punt les coordenades  $x$  i  $y$  del qual són els valors negats del punt receptor. Així, el punt  $(200@400)$  negated és el punt  $-200@-400$ . Fixeu-vos que els parèntesis són necessaris. De fet, negated és també un mètode entès pels nombres. Així, l'expressió  $200@400$  negated dóna com a resultat el punt  $200@-400$  (un punt fora de la pantalla), ja que el mètode negated, sent un mètode unari, és executat pel nombre 400 abans d'executar-se el mètode  $@$ . A l'*Script 21.16*, aquest mètode és utilitzat per produir una translació en la direcció oposada.
- El mètode setX:setY: canvia les coordenades d'un punt. Així, després d'executar l'expressió `unPunt setX: 200 setY: 400`, el punt `unPunt` té coordenada  $x$  200 i coordenada  $y$  400.

### **Script 21.16** *Un patró fet d'oques volant*

```
| pica punt1 mou desplaçament |
punt1 := 200@300.
mou := 25@0.
desplaçament := -25@50.
pica := Bot nou.
5 vegadesRepetir:
  [ 10 vegadesRepetir:
    [ pica
```

```

triangleA: punt1
delta1: -25@-25
delta2: -25@25.
punt1 := punt1 + mou ].
punt1 := punt1 + desplacament.
mou := mou negated.
desplacament setX: desplacament x negated setY: desplacament y ].

```

L'*Script* 21.16 utilitza els mètodes negated i setX:setY: dins d'un doble bucle per generar el patró d'oques volant sobre una determinada regió de la pantalla. El bucle interior és fet de manera similar a l'*Script* 21.12, excepte que l'orientació del triangle i la de la translació han estat girades, de manera que la línia de triangles és ara horitzontal. El bucle exterior fa una translació del darrer triangle per posar-lo sobre de la línia de triangles tot just dibuixada, utilitzant la variable desplacament; després, la translació és invertida de manera que la línia següent és dibuixada en l'ordre capgirat. Els triangles, però, encara es dibuixen amb la mateixa orientació. El fet que una segona línia de triangles sembli apuntar en la direcció oposada és una il·lusió òptica provocada pel contacte que hi ha entre línies de triangles. Fixeu-vos que la variable desplacament ha de ser transformada d'una manera especial: el signe de la seva coordenada *x* és invertit al final de cada línia per compensar la darrera translació, que no es dibuixa.

## Més experiments

### Experiment 21-6 (traslladar un robot a un punt)

Definir mètodes amb un comportament senzill i ben definit és una manera de simplificar el vostre codi, tal com vam explicar al capítol 16. Com definiríeu un mètode trasllada: unPunt que trasllada el receptor, un robot, fins a unPunt des de la seva posició actual? Abans de consultar la nostra solució al Mètode 21.1, proveu de fer-ho vosaltres mateixos.

#### Mètode 21.1 Traslladar un Robot a un Punt

##### **trasllada: unPunt**

"trasllada el receptor a unPunt"

self vesA: (self center + unPunt)

Proposeu un mètode diferent, `traslladaX: x y: y` que pren els valor d'`x` i d'`y` separadament com a arguments.

### Experiment 21-7 (utilitzar el mètode `trasllada:` 1)

Canvieu la definició del mètode `triangleA:punt2:punt3:` per tal que utilitzi el mètode `trasllada:unPunt`.

---

### Experiment 21-8 (utilitzar el mètode `trasllada:` 2)

Utilitzant el mètode `trasllada:unPunt`, torneau a implementar alguns dels mètodes que heu fet en aquest capítol, i compareu-ne la mida i la complexitat.

---

## Resum

- Un punt és un parell de nombres: tenen una coordenada  $x$ , o coordenada horitzontal, i una coordenada  $y$ , o coordenada vertical.
- `200@400` és un punt la coordenada  $x$  del qual és 200 i la coordenada  $y$  del qual és 400.
- El sistema de coordenades d'Smalltalk té el seu origen (0,0) a la cantonada superior esquerra de la pantalla, i l'eix  $y$  té la seva direcció positiva cap avall.
- Per evitar problemes amb els punts, envolteu-los de parèntesis quan participen en operacions complexes.
- Els mètode `vesA: unPunt` i `saltaA: unPunt` fan que el receptor es mogui a la posició de l'argument, un punt.

Aquí teniu alguns dels mètodes associats als punts:

Missatge	Descripció	Exemple
<code>x @ y</code>	Crea un punt a les coordenades donades.	<code>300 @ 600</code>
<code>vesA: unPunt</code>	Ordena al robot moure's a un punt donat.	<code>pica vesA: 300 @ 600</code>
<code>saltaA: unPunt</code>	Posiciona un robot en el punt donat.	<code>pica saltaA: 300 @ 600</code>
<code>punt1 + punt2</code>	Crea un punt les coordenades del qual són la suma de les coordenades dels dos punts donats. Això és útil per representar translacions.	<code>(50@200) + (300@600)</code>
<code>punt1 * nombre</code>	Crea un punt les coordenades del qual són els productes de les coordenades del punt i el nombre.	<code>(50@200)*3</code>
<code>punt1 negated</code>	Construeix un punt les coordenades del qual són les oposades de les coordenades del punt original.	<code>(50@200) negated</code>
<code>center</code>	Retorna la posició actual del robot com a punt.	<code>punt := pica center</code>

## Capítol 22

# Comportament avançat dels robots

Fins ara us havíem presentat només un subconjunt dels missatges que es poden enviar a un robot. En aquest capítol presentarem alguns missatges més avançats que utilitzarem en propers experiments.

### Obtenir la direcció d'un robot

El primer mètode que ens cal és el mètode `direccio`, que retorna la direcció actual del robot en graus. Utilitzant aquest missatge podeu verificar que els missatges `nord`, `sud`, etc. que modifiquen la direcció del robot d'una manera absoluta estan d'acord amb les seves definicions matemàtiques, com veieu a l'*Script 22.1* i s'il·lustra a la figura 22.1. Fixeu-vos que la direcció és sempre un nombre entre -179 i 180 graus.

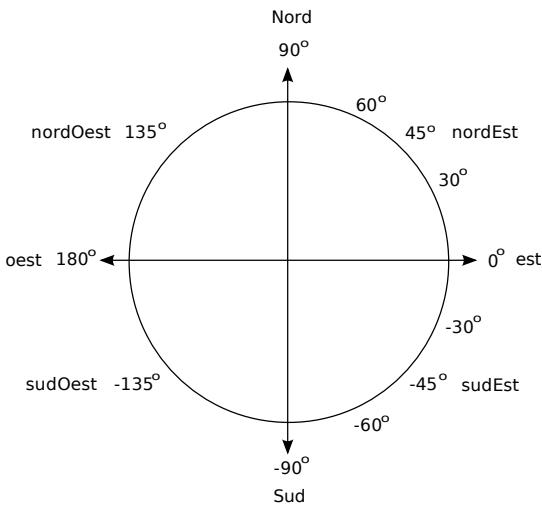
#### **Script 22.1** Il·lustrar el mètode `direccio`

```
| robot |
robot := Bot nou.
robot nord.
robot direccio.
– Escriure el valor retornat: 90
robot oest.
robot direccio.
– Escriure el valor retornat: 180
robot est.
robot direccio.
```

- *Escriure el valor retornat: 0*  
robot sudEst.
- robot direccio.
- *Escriure el valor retornat: -45*

## Apuntar en una direcció

El missatge `giraA: unaDireccio` li diu al receptor que giri cap a una determinada direcció donada com un angle en graus. Hem anomenat l'argument d'aquest missatge `unaDireccio` més que `unAngle` per emfatitzar el fet que el paràmetre representa un angle absolut, basat en la definició matemàtica il·lustrada a la figura 22.1. Així, per exemple, el missatge `giraA: 45` gira el robot receptor al nordest, sense importar on apuntava abans. Compareu amb el missatge `gira: 45`, que gira el robot 45 graus a l'esquerra *relatiu a la posició actual*. Això significa que l'expressió `robot giraA: 90` és equivalent a totes les següents expressions: `robot nord`, `robot giraA: -270` i `robot giraA: 90 + 360` (veieu l'*Script 22.2*). Fixeu-vos que els nombres a la figura 22.1 són els més petits en valor absolut per representar un angle, i de fet, la implementació d'Smalltalk sempre fa que el robot giri aquesta quantitat mínima.



**Figura 22.1 — Els angles associats amb els missatges de direccions absolutes.**

**Script 22.2 Il·lustrar el mètode giraA: unAngleAbsolutEnGraus**

```
| robot |
robot := Bot nou.
robot giraA: 90.
"dóna el mateix resultat que: robot nord"
```

**Distància des d'un punt**

La propera informació que voldríem obtenir d'un robot és la seva distància a un punt donat. Això és precisament el que obteniu quan envieu el missatge `distanciaDesde: unPunt` a un robot. Aquesta informació és útil, per exemple, quan volem saber si un robot s'està apropiant a un lloc determinat.

**Script 22.3 Il·lustrar distanciaDesde: unPunt**

```
| robot |
robot := Bot nou.
robot saltaA: 100@100.
robot distanciaDesde: (140@130).
– Escriure el valor retornat: 50
```

**Tornar al centre de la pantalla**

Un altre missatge útil és `origen`, que posiciona el robot receptor al lloc on va aparèixer quan es va crear, és a dir, al centre de la pantalla.

**Posició si es mogués**

Algunes vegades ens agradaria conèixer la posició que un robot ocuparia si es mogués una determinada distància en la seva direcció actual. Utilitzarem molt aquesta funcionalitat quan simulem comportament animal. Per a això s'ha definit el mètode `posicioSiVes: unaDistancia` i es pot utilitzar com veieu a l'*Script 22.4*.

**Script 22.4 Il·lustrar posicioSiVes: unaDistancia**

```
| robot |
robot := Bot nou.
robot saltaA: 100@100.
robot est.
robot posicioSiVes: 100.
```

– *Escriure el valor retornat: 200@100*

**En una capsa**

Abans de llegir-ne la solució, proveu de definir un mètode ves: unEnter siDinsCapsa: unRectangle que mou el receptor només si aquest moviment no el posiciona fora d'un determinat rectangle, com veieu a l'*Script 22.5*.

**Script 22.5 Il·lustrar ves: unEnter siDinsCapsa: unRectangle**

```
Bot nou
ves: 100
siDinsCapsa: (Rectangle center: World center extent: 400@300)
```

Per crear el rectangle podeu utilitzar, per exemple, el mètode center:extent:, que crea un rectangle centrat en un punt. L'expressió (Rectangle center: World center extent: 400@300) retorna un rectangle el centre del qual és el centre de la pantalla i la base i l'altura del qual són 400 i 300 píxels. Podeu demanar a un rectangle si conté un punt utilitzant el mètode containsPoint: unPunt. Per determinar quina seria la posició del robot si es mogués endavant una distància determinada podeu utilitzar el mètode posicioSiVes: unaDistancia.

El Mètode 22.1 us mostra la definició de ves: unEnter siDinsCapsa: unRectangle. Utilitza el mètode posicioSiVes:, que calcula la posició del robot si s'hagués de moure endavant una determinada distància en la seva direcció actual. Farem servir la idea de restringir el moviment del robot per representar comportament animal al capítol 23.

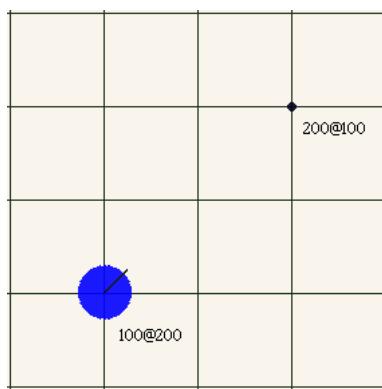
**Mètode 22.1 Mou el receptor només si anirà a parar dins d'un rectangle donat**

ves: unEnter siDinsCapsa: unRectangle  
 “mou endavant el receptor només si roman dins un rectangle”

```
(unRectangle containsPoint: (self posicioSiVes: unEnter))
siCert: [ self ves: unEnter. ]
```

## Apuntar cap a un punt

De vegades és necessari demanar a un robot que es dirigeixi cap a un punt determinat. El mètode apuntaA: unPunt canvia la direcció del receptor de manera que apunti en la direcció del punt unPunt. Un cop el mètode s'ha executat, el robot receptor apunta en la direcció del punt especificat, com veieu a la figura 22.2. Podeu verificar-ho utilitzant el mètode direccio, tal com mostrem als Scripts 22.6 i 22.7.



**Figura 22.2** — Un robot apunta al punt 200@100 després d'haver-li enviat el missatge apuntaA::.

### Script 22.6 Il·lustrar apuntaA: unPunt

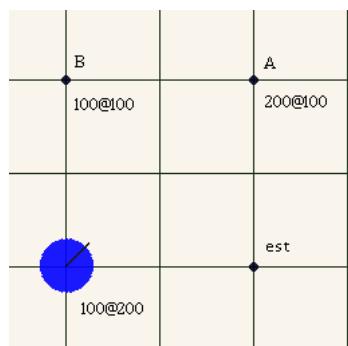
```
| robot |
robot := Bot nou.
robot saltaA: 100@200.
robot apuntaA: 200@100.
robot direccio
– Escriure el valor retornat: 45
```

### Script 22.7 Il·lustrar apuntaA: unPunt

```
| robot |
robot := Bot nou.
robot saltaA: 100@200.
robot apuntaA: 200@300.
robot direccio
– Escriure el valor retornat: -45
```

Tot i així, ser capaç d'apuntar a un lloc donat pot no ser suficient. De vegades, ens agradaria conèixer l'angle que hauria de girar un robot per apuntar a una posició determinada. El mètode anglePerApuntarA: unPunt retorna la diferència entre la direcció actual del receptor i la direcció amb què apuntaria cap a unPunt.

La figura 22.3 i l'*Script* 22.8 il·lustren aquest mètode. A la figura, un robot està apuntant en la direcció del punt A, i la seva direcció és 45. Ara, l'expressió anglePerApuntarA: 200@100 retorna 0, perquè el robot ja està apuntant cap a aquest punt. Després el robot és girat cap al punt B. Ara l'expressió anglePerApuntarA: 200@100 retorna -45, ja que el robot s'hauria de girar 45 graus en la direcció de les agulles del rellotge per apuntar cap al punt A.



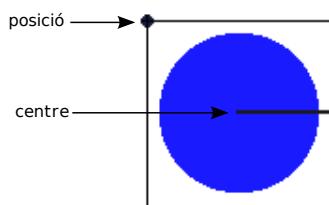
**Figura 22.3** — El mètode anglePerApuntarA: unPunt retorna l'angle que el robot hauria de girar per apuntar a unPunt.

**Script 22.8** Il·lustrar anglePerApuntarA:. Veieu la figura 22.3.

```
| robot |
robot := Bot nou.
robot saltaA: 100@200.
robot apuntaA: 200@100.
robot direccio
- Escriure el valor retornat: 45
robot anglePerApuntarA: 200@100.
- Escriure el valor retornat: 0
robot apuntaA: 100@100.
robot anglePerApuntarA: 200@100.
- Escriure el valor retornat: -45
```

## Centre versus posició

Finalment, pot ser que vulgueu conèixer la posició actual del robot a la pantalla. Per obtenir aquesta informació podeu fer servir el mètode `center`, que retorna la posició del llapis del robot. Un robot també entén el mètode `position`, proporcionat per Squeak, que retorna la cantonada superior esquerra del rectangle representant el robot, com mostrem a la figura 22.4. Usualment, no heu d'utilitzar el mètode `position`, que ve donat per Squeak i no és específic de la classe `Bot`.



**Figura 22.4 — La diferència entre la posició i el centre d'un robot.**

## Resum

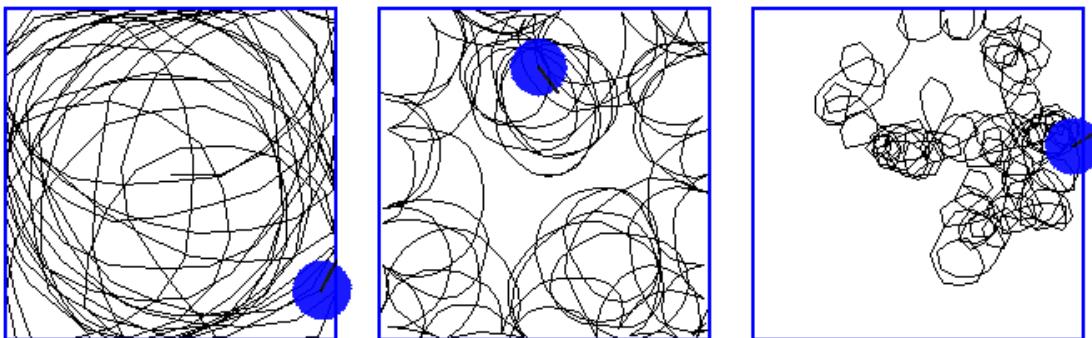
La taula següent resumeix els mètodes introduïts en aquest capítol:

Missatge	Descripció	Exemple
<code>anglePerApuntarA: unPunt</code>	Ordena al receptor calcular l'angle que hauria de girar per apuntar a <code>unPunt</code>	<code>berthe anglePerApuntarA: 100@100</code>
<code>giraA: unaDireccio</code>	Ordena al receptor girar en la direcció <code>unaDireccio</code>	<code>berthe giraA: 90</code>
<code>direccio</code>	Ordena al receptor informar de la seva direcció actual	<code>berthe nord; direccio</code>
<code>position</code>	Ordena al receptor comunicar quina és la seva cantonada superior esquerra	<code>berthe position</code>
<code>center</code>	Ordena al receptor informar de la posició del seu llapis	<code>berthe center</code>
<code>distanciaDesde: unPunt</code>	Ordena al receptor informar de la seva distància a <code>unPunt</code>	<code>berthe distanciaDesde: 140@130</code>



## Capítol 23

# Simular comportament animal

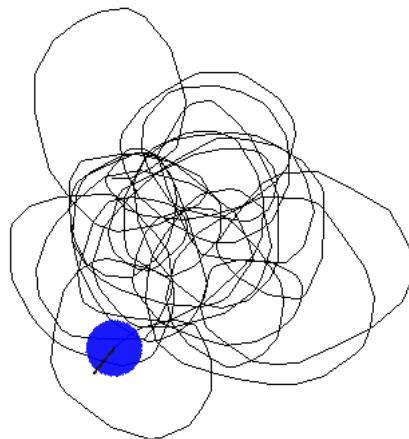


Els ordinadors són bons per modelitzar el món en què vivim, des del creixement de les plantes fins a models del comportament dels mercats en ciències econòmiques. En aquest capítol us ensenyarem com modelitzar determinats comportaments animals i a utilitzar la simulació per entendre els factors que influencien aquests comportaments. Modelitzarem algunes estratègies que els animals utilitzen per caminar, escapar-se, trobar menjar i romandre en un entorn favorable.

### Vagar

Comencem modelitzant com podria vagar un animal. L'aproximació bàsica per simular el comportament vagant d'un animal és escriure un bucle en el qual l'animal camina i gira una mica a l'atzar. Utilitzarem un nombre aleatori en definir el mètode vagant: `unNombre`, que fa que el re-

ceptor camini un nombre aleatori de passos, giri un angle aleatori i repeteixi aquests moviments `unNombre` vegades (per obtenir un nombre aleatori entre 1 i 30, cal enviar el missatge `atRandom` al nombre 30). Un resultat possible d'executar el mètode el podeu veure a la figura 23.1.



**Figura 23.1** — *Un animal vaga caminant un nombre aleatori de passos i després girant a l'atzar.*

El Mètode 23.1 presenta una manera de definir el comportament senzill descrit més amunt i il·lustrat a la figura. Aquí simplement ordenem al robot que es mogui aleatòriament una distància entre 1 i 30 píxels i que giri un angle entre 1 i 30 graus. L'Script 23.1 mostra com fer servir aquest mètode.

### Script 23.1

Bot nou vagant: 500

### Mètode 23.1

#### vagant: n

"Fer moure el robot una distància aleatòria i girar un angle a l'atzar n vegades"

n vegadesRepetir:

```
[ self ves: 30 atRandom.  
  self giraEsquerra: 30 atRandom ]
```

Naturalment, els animals no vaguen a l'atzar. Eons de desenvolupament evolutiu han creat animals que es mouen i giren en resposta a estímuls del seu entorn. Potser un animal gira i vaga fins que succeeix un cert esdeveniment. Per començar a modelitzar aquest comportament, que l'animal repeteix fins que passa un esdeveniment determinat, modelitzat aquí com un clic d'un botó del ratolí per part de l'usuari, podríeu utilitzar un bucle condicional. El mètode **vagantFinsBotoPitjat** (Mètode 23.2) il·lustra aquest aspecte del model. Permet vagar a un animal fins que l'usuari pitgi un dels botons del ratolí. Com que el bucle és executat molt ràpidament, pot passar que l'ordinador sembli bloquejat, per tant continueu pitjant el botó fins que s'aturi.

### Mètode 23.2 *Un animal vaga a l'atzar fins que succeeix un esdeveniment.*

#### **vagantFinsBotoPitjat**

"Fer moure el robot una distància aleatòria i girar un angle a l'atzar fins que un botó del ratolí sigui pitjat"

```
[ self anyButtonPressed ] mentreFals:  
    [ self ves: 30 atRandom.  
      self giraEsquerra: 30 atRandom ]
```

## Separar influències

El Mètode 23.1 és interessant, però barreja dos aspectes del vagar de l'animal: el moviment a l'atzar i els canvis de direcció aleatoris. És més, cada cop que voleu provar un valor nou per a l'angle o el nombre de passos, heu de recompilar el mètode **vagant**: n. Així, definiu un mètode **vagant**: n maxAngle: unAngle que pren com a segon argument el valor màxim de l'angle aleatori que el receptor hauria de girar. En aquest mètode, feu que el robot sempre es mogui endavant una distància constant. Aquest mètode es pot utilitzar com veieu a l'*Script 23.2*.

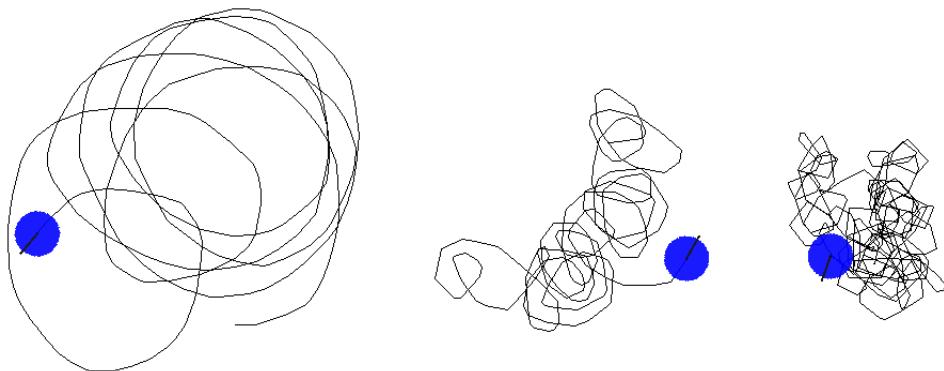
### Script 23.2

Bot nou vagant: 500 maxAngle: 60

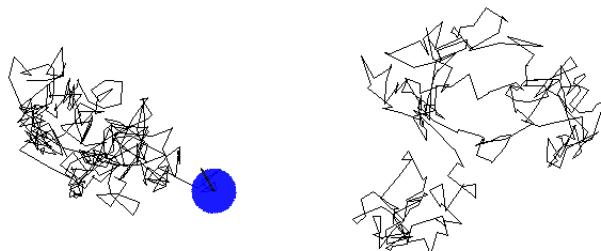
Proveu d'endevinar quin aspecte tindrà la traça deixada per l'animal abans d'executar els vostres *scripts*. Experimenteu amb diferents valors pels angles. Les figures 23.2 i 23.3 mostren resultats diferents amb 15, 60, 90, 180 i 360 graus.

## Estudiar la influència de la longitud

Hem estat jugant amb la influència de l'angle en la forma del recorregut. Quines hipòtesis teniu sobre la influència de la longitud? Què hauria de passar si un animal caminés una distància aleatòria i després girés un angle constant?



**Figura 23.2** — Caminant amb angles aleatoris de com a molt 15 (esquerra), 60 (mig) i 90 (dreta) graus.



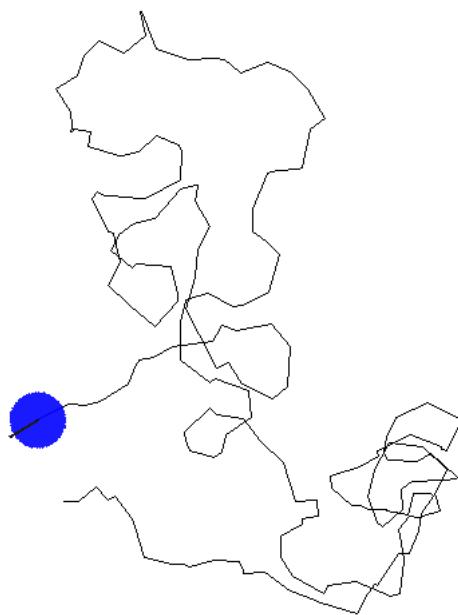
**Figura 23.3** — Caminant amb angles aleatoris de com a molt 180 (esquerra) i 360 (dreta) graus.

### Estudiar la influència del costat cap a on gira l'animal

Fins ara, el nostre animal sempre ha girat cap a l'esquerra. Podem estudiar la influència de l'habilitat de girar només cap a un costat o cap als dos costats. Proveu de pensar una solució a aquest problema de programació abans de llegir-ne la nostra. Ajuda: Fixeu-vos que `atRandom` retorna un nombre entre el receptor i 1. Per tant, 2 `atRandom` retorna o bé 1 o bé 2.

Una possible manera de generar una elecció aleatòria és introduint un nombre aleatori (1 o 2) per representar el costat cap a on es girarà, com s'esbossa a l'*Script 23.3*. Una altra idea és generar un nombre aleatori com a màxim el doble de l'angle màxim desitjat i restar-ne aquest

angle. Per exemple, per obtenir un nombre entre -45 i 45, podeu generar un nombre aleatori entre 1 i 91 i restar-ne 46, com veieu a l'*Script* 23.4. La figura 23.4 mostra què podria passar si utilitzéssim aquestes estratègies, i podeu veure que el recorregut sembla molt més el d'un animal real, diguem un cargol, que els recorreguts anteriors.



**Figura 23.4** — Un animal vaga caminant i girant a l'atzar, on és igualment probable girar a l'esquerra i girar a la dreta.

**Script 23.3** Un nombre aleatori (1 o 2) determina si l'animal gira a la dreta o a l'esquerra.

```
...
esquerra := 2 atRandom.
esquerra = 1
  siCert: [ self giraEsquerra: ... ]
  siFals: [ self giraDreta: ... ]
...

```

**Script 23.4** *L'angle màxim de gir es resta d'un nombre aleatori per donar com a resultat un angle que pot ser tant negatiu com positiu.*

```
...
rdAngle := ((1 + (angle * 2)) atRandom) - (1 + angle).
self gira: rdAngle.
...
```

## Atrapat dins una capsà

Ara ens agradaria restringir el moviment de l'animal de manera que romangui dins d'una capsà. Això ens permetrà estudiar estratègies diferents que aparentment alguns insectes fan servir quan es troben en una situació similar. Fer-ho és senzil. Abans d'ordenar l'animal que es mogui una certa distància, cal comprovar si la posició on anirà a parar és dins de la capsà. Aquest moviment restringit ja es va presentar al capítol 22, amb el Mètode 22.1, però repetirem el codi en el Mètode 23.3

### Mètode 23.3

**ves: unaDistancia siDinsCapsa: unRectangle**

"mou endavant el receptor només si roman dins un rectangle"  
`(unRectangle containsPoint: (self posiciosSiVes: unaDistancia))`  
`siCert: [ self ves: unaDistancia. ]`

Per crear una capsà, podem crear un rectangle, com veieu a l'*Script 23.5*. Aquest *script* crea una capsà de costat 200 píxels al voltant de la posició actual de l'animal.

**Script 23.5** *Crear un rectangle centrat en un animal.*

```
| pica rectangle |
pica := Bot nou.
rectangle := Rectangle center: pica center extent: 200@200.
```

Per millorar visualment la simulació, i si voleu veure la capsà a la pantalla, heu de crear un "rectangle morph", és a dir, un objecte gràfic la forma del qual és un rectangle, com fem a l'*Script 23.6*. Aquest *script* primer crea un rectangle, i després crea un *rectangle morph* afitat pel rectangle, amb l'interior transparent i un costat blau. Finalment, dibuixem el *rectangle morph* amb el mètode `openInWorld`.

**Script 23.6** Visualitzar un rectangle centrat en un animal.

```
| pica rectangle rm | pica := Bot nou.
rectangle := Rectangle center: pica center extent: 200@200.
rm := RectangleMorph new.
rm bounds: rectangle.
rm color: Color transparent.
rm setBorderWidth: 2 borderColor: Color blau.
rm openInWorld.
```

Ara definirem el mètode `capsa: unRectangle` que dibuixa una capsà representant el rectangle, ja que l'utilitzareu molt.

**Mètode 23.4****`capsa: unRectangle`**

"Dibuixa un *morph* per representar el rectangle"

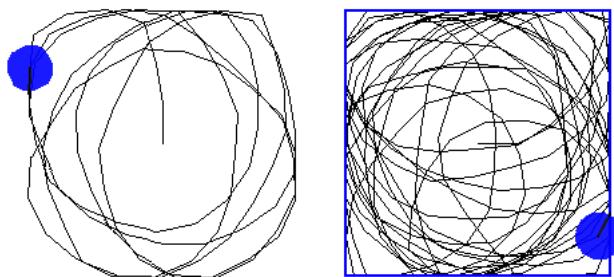
```
| rm |
rm := RectangleMorph new.
rm bounds: unRectangle.
rm color: Color transparent.
rm setBorderWidth: 2 borderColor: Color blau.
rm openInWorld.
```

Ara ja podem combinar totes les peces per crear un animal dins de la capsà i comprovar el mètode `ves: unaDistancia siDinsCapsa: unRectangle`. La propera qüestió és: Com podem modelitzar més fidelment el comportament d'un animal tancat? Imagineu diferents alternatives. Hauríeu d'experimentar amb unes quantes variacions.

**Resseguir els costats**

Una aproximació possible és fer que l'animal giri una miqueta quan no es pot moure, i ho torni a provar. Això és el que fem amb el Mètode 23.5. Quan el robot es pot moure, només vaga, però si el seu moviment el faria sortir de la capsà, gira un cert angle. Aquest moviment és dins d'un bucle que fa que el robot giri i torni a provar de moure's. Si el gir no és prou gran, continua girant fins que es pot moure altre cop.

Els *Scripts* 23.6 i 23.7 junts produeixen resultats similars als resultats que podeu veure a la figura 23.5. L'*Script* 23.6 mostra un *morph* rectangular per fer aparèixer la capsà i l'*Script* 23.7 controla el moviment de l'animal.



**Figura 23.5** — Un animal tancat dins d'una capsula gira fins que es pot moure sense xocar amb un costat.

**Script 23.7** *Un animal tancat en una capsula prova de moure's*

```
| t rec |
t := Bot nou.
rec := (Rectangle center: t center extent: 200@200).
t ressegueix: 500 costatDeCapsa: rec
```

**Mètode 23.5** *El robot gira si no es pot moure i torna a provar-ho.*

**ressegueix: n costatDeCapsa: unRectangle**

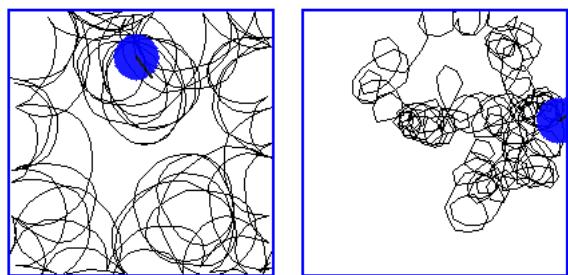
```
self capsula: unRectangle.
n vegadesRepetir:
[ (unRectangle containsPoint: (self posicioSiVes: 30))
  siCert: [ self ves: 30.
            self giraEsquerra: 30 atRandom ]
  siFals: [ self giraEsquerra: 1 ] ]
```

## Volar al costat oposat

Una altra estratègia per simular és la de l'insecte que prova de volar fins al costat oposat. Us deixarem programar-ho a vosaltres mateixos, en teniu una possible traça a l'esquerra de la figura 23.6.

## Direcció aleatòria

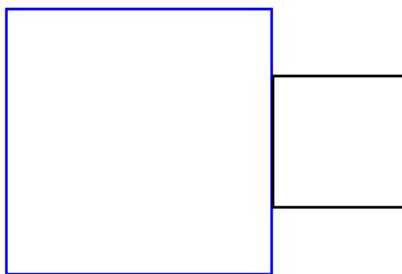
Una altra possibilitat per canviar la direcció de l'animal és fer-ho a l'atzar. Us deixarem definir aquest comportament, pel qual teniu una possible traça a la dreta de la figura 23.6.



**Figura 23.6** — Esquerra: L'insecte es mou en la direcció oposada quan es troba amb una paret. Dreta: L'insecte tria una direcció a l'atzar quan es troba amb una paret. Aquesta figura es va obtenir amb un valor màxim de moviment de 10 píxels i girant un màxim de 90 graus. Per escapar, farà un gir aleatori d'un màxim de 360 graus.

### Afegir una sortida a la capsa

Ara podeu afegir un altre rectangle per representar la sortida de la capsula, tal com està il·lustrat a l'*Script 23.8* i podeu veure a la figura 23.7.



**Figura 23.7** — Una capsula amb una sortida.

### Script 23.8 Visualitzar una capsula amb una sortida.

```
| box rm rm2 exit |
box := Rectangle center: World center extent: 200@200.
rm := RectangleMorph new.
```

```

rm bounds: box.
rm color: Color transparent.
rm setBorderWidth: 2 borderColor: Color blau.
rm openInWorld.
exit := Rectangle origin: (box topRight + (0@50)) extent: 100@100.
rm2 := RectangleMorph new.
rm2 bounds: exit.
rm2 color: Color transparent.
rm2 setBorderWidth: 2 borderColor: Color negre.
rm2 openInWorld.

```

Definiu un mètode per estalviar-vos de repetir aquest codi més d'una vegada. Podríeu crear un mètode anomenat escapant: unaCapsa ambSortida: unaSortida que primer comprovaria si el proper moviment estaria contingut dins el rectangle que representa la sortida, i si aquest fos el cas, el mètode deixaria escapar l'animal.

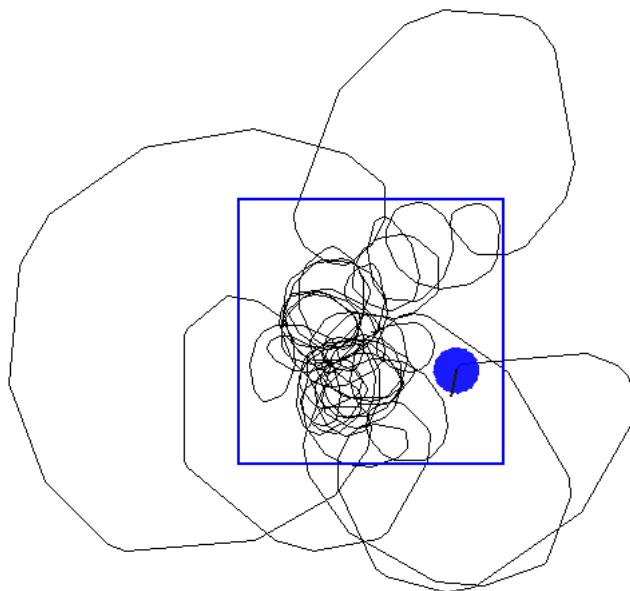
## Romandre en un entorn segur

Us heu preguntat mai com és possible que els bacteris del líquim romanguin sempre a prop de llocs humits, evitant els llocs secs? D'acord, potser no. Sigui com sigui, us proposem simular les estratègies que formes de vida bàsiques com els bacteris utilitzen per romandre en àrees on tenen més possibilitats de sobreviure. Us proposem que modelitzeu dues estratègies molt senzilles descobertes a finals dels anys 50 utilitzant només canvis en direcció i velocitat. Fixeu-vos que quan parlem de velocitat, volem dir la distància que l'animal es pot moure a cada pas, que en el nostre cas serà la distància en píxels que rep el mètode ves: com a argument.

La primera estratègia que els bacteris poden seguir és canviar de direcció a l'atzar en qualsevol circumstància però canviar de velocitat depenen del grau de seguretat de l'entorn. Aquesta estratègia té sentit ja que, si un bacteri considera una regió segura, trigarà més temps a fer un determinat recorregut, i per tant romandrà més temps en un entorn segur que en un entorn insegur. Un possible recorregut per un bacteri es mostra a la figura 23.8.

La segona estratègia és l'oposada. El bacteri es mou a velocitat constant però canvia de velocitat en funció de la seguretat que li ofereix l'entorn. Canvia la seva direcció un angle en mitjana més gran en un entorn segur que en un entorn insegur. Per tant, té més possibilitats de tornar sobre els seus passos i romandre dins una regió més petita. Alguns bacteris molt senzills fan servir aquesta estratègia per romandre en entorns on poden trobar menjar. Un possible recorregut per a un bacteri d'aquestes característiques el podeu veure a la figura 23.9.

Per implementar aquestes idees, només cal que us imagineu que el rectangle que hem estat utilitzant representa una regió segura. Per a la primera estratègia, podríeu definir un mètode

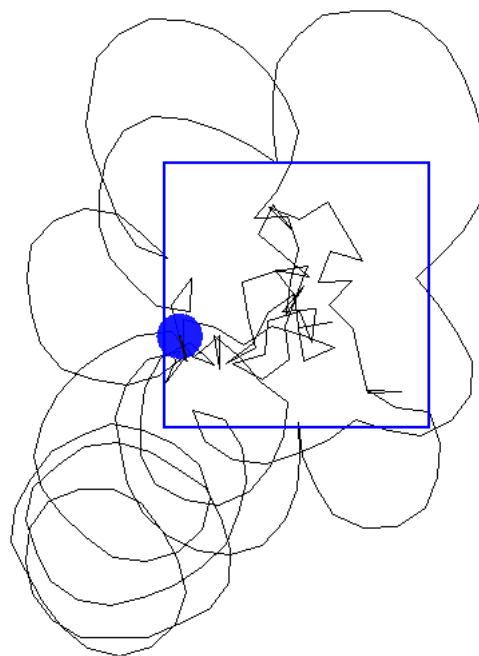


**Figura 23.8 —** Primera estratègia. El bacteri incrementa la seva velocitat quan es troba en un entorn insegur. Canvia de direcció amb una taxa constant. La velocitat en un entorn segur és un nombre aleatori com a molt gran de 25 píxels, mentre que en un entorn insegur és de 100 píxels. L'interior del rectangle representa l'entorn segur.

anomenat romanEnAngleConstantNVegades: unNombre dinsUn: unaRegio, com veieu a l'Script 23.9. Experimenteu amb diferents valors de l'angle que pot girar el bacteri.

### Script 23.9

```
| bacteri |  
Bot clearWorld.  
bacteri := Bot nou.  
bacteri romanEnAngleConstantNVegades: 500  
    dinsUn: (Rectangle center: bacteri center extent: 200@200).
```



**Figura 23.9 — Segona estratègia.** El bacteri incrementa el seu rang de canvi de direcció en un entorn segur. Aquí la velocitat és 25, i el canvi de direcció és com a màxim 36 en un entorn insegur i 360 en un de segur.

**Mètode 23.6** Un bacteri canvia la seva velocitat per romandre en un entorn segur.

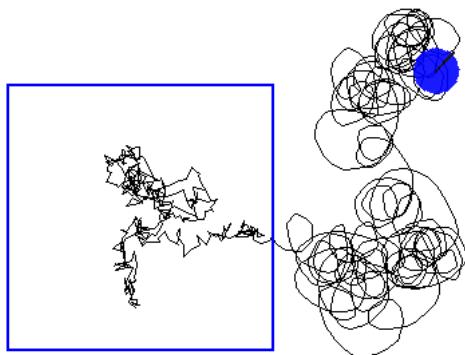
**romanEnAngleConstantNVegades: n dinsUn: unRectangle**

“El receptor prova de quedar-se dins un entorn segur canviant la seva velocitat, repetit n vegades”

```
self capsula: unRectangle.
n vegadesRepetir:
  [ (unRectangle containsPoint: self center)
    siCert: [ self ves: 25 atRandom ]
    siFals: [ self ves: 100 atRandom ].  
  self gira: 25 ]
```

## Més experiments

Podríem imaginar-nos que l'animal fa decretar la seva velocitat en funció de la distància a la zona segura. Podríeu introduir una mica de comportament a l'atzar dins el criteri (velocitat o angle de gir) que sinó no canvia. Podríeu també introduir la possibilitat que el bacteri pugui girar a favor o en contra del sentit de les agulles del rellotge, tal com hem fet abans en aquest capítol. Com mostra la figura 23.10, la segona estratègia no és especialment eficient; el bacteri no té manera de "saber" si s'està movent en direcció a la regió segura, de manera que pot acabar romanent fora de l'àrea segura massa temps i morir. Proposeu algunes solucions a aquest problema.



**Figura 23.10 — Segona estratègia.** Aquí la velocitat és com a molt 10; i el canvi de direcció del bacteri és 36 graus en un entorn inseguir i 360 en un de segur.

## Trobar menjar

Ara ens agradaria modelitzar les diferents maneres que un animal podria utilitzar per buscar menjar. Una primera aproximació pot basar-se en el fet que un animal pot localitzar el seu menjar visualment. Un altre cop, representarem l'àrea del menjar amb un rectangle.

## Comparar la distància

Imagineu que un animal pot avaluar la distància a una font d'aliment, que modelitzarem com la distància de l'animal al centre del rectangle utilitzat per representar l'àrea on el menjar està situat. Fixeu-vos que per obtenir la distància entre dos punts, podeu utilitzar el mètode `dist`. Per

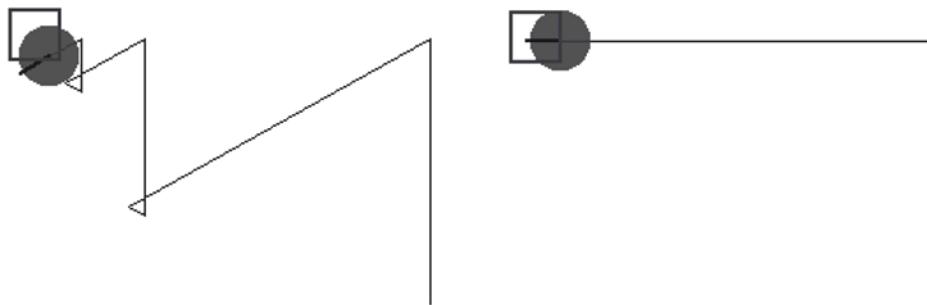
exemple, `100@100 dist: 200@200` retorna la distància entre els punts `100@100` i `200@200`. Podeu obtenir la distància d'un robot i un punt amb el mètode `distanciaDesde: unPunt`.

Implementeu un mètode anomenat, per exemple, `trobaAreaAlimentPerDistancia: unRectangleAlimentic` que determina si caminant un pas endavant l'animal s'acosta a la zona amb aliment. Quan s'acosta, continua movent-se, però si s'allunya, canvia la seva direcció una certa quantitat determinada. L'*Script 23.10* mostra com podríem fer servir aquest mètode.

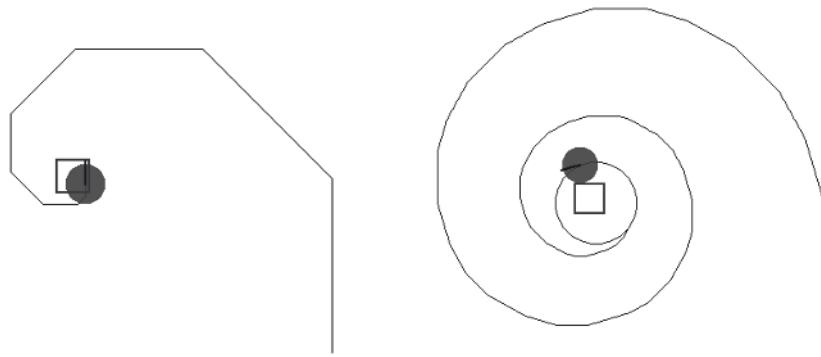
**Script 23.10** Executar `trobaAreaAlimentPerDistancia`:

```
| animal menjar |
animal := Bot nou.
menjar := Rectangle center: 400@250 extent: 30@30.
animal trobaAreaAlimentPerDistancia: menjар.
```

El comportament aquí descrit és una mica ingenu, ja que no tenim cap garantia que girant un cert angle millori la situació. Les figures 23.11 i 23.12 presenten algunes traces. A la dreta de la figura 23.12 es mostra una situació en què un animal s'aproxima al menjar i després hi dóna voltes sense arribar-hi mai. Proveu de pensar algunes solucions a aquest problema. El Mètode 23.7 presenta una possible definició del mètode `trobaAreaAlimentPerDistancia`:



**Figura 23.11** — Trobar menjar comparant distàncies i girant. Esquerra: gira 120 graus. Dreta: gira 90 graus.



**Figura 23.12** — Trobar menjar comparant distàncies i girant. Esquerra: gira 45 graus. Dreta: gira 15 graus.

**Mètode 23.7** Un animal s'acosta a una font d'aliment provant de fer decréixer la seva distància al menjar.

#### **trobaAreaAlimentPerDistancia: rectangleMenjar**

```
| menjar moviment |
self capsas: rectangleMenjar.
moviment := 10.
menjar := rectangleMenjar center.
[ (rectangleMenjar containsPoint: self center)
or: [ self anyButtonPressed ] ] mentreFals:
[ (( self posicioSiVes: moviment) dist: menjar) > (self distanciaDesde: menjar)
siCert: [ self giraEsquerra: 15 ]
siFals: [ self ves: moviment ] ]
```

## Més experiments

Aquí teniu algunes idees per fer més experiments. Canvieu la posició del menjar o la direcció de l'animal al començament del seu recorregut a l'*Script 23.10*.

Milloreu el comportament implementat al Mètode 23.7 de manera que un cop l'animal s'adoni que s'està mouent en la direcció equivocada, no girarà ingènuament un angle aleatori sinó que comprovarà primer si girant d'una manera determinada millora la seva situació, i per tant girarà

cap al costat correcte. Per ajudar-vos en els vostres experiments, no dubteu a definir mètodes nous. Per exemple, podríeu definir un mètode posicioSiVes: unaDistancia iGira: unAngle que retorna la posició on el receptor estaria si girés unAngle i es mogués endavant unaDistancia (veieu el Mètode 23.8).

**Mètode 23.8** *Troba la posició en què acabaria el receptor si girés un determinat angle i es mogués una determinada distància.*

#### **posicioSiVes: unaDistancia iGira: unAngle**

“Retorna la posició en què el receptor acabaria si girés unAngle i es mogués endavant unaDistància.”

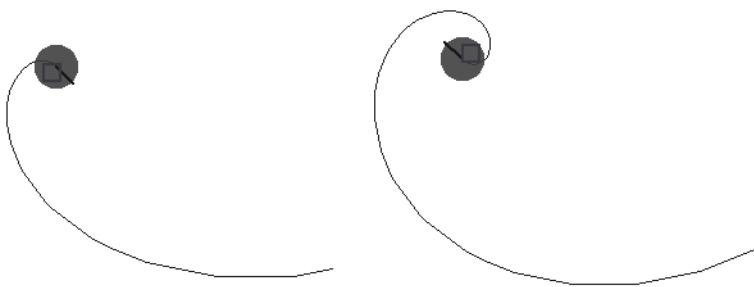
```
| posicio |
self gira: unAngle.
posicio := 10.
menjar := self posicioDireccioPerDistancia: unaDistancia.
self gira: unAngle negated.
^ posicio
```

Hem estat parlant de la distància entre la font d'aliment i l'animal, però és improbable que l'animal tingui cap manera de mesurar aquesta distància. Malgrat això, alguns animals poden calcular la distància d'una font d'aliment de diverses maneres, com la intensitat de l'olor del menjar. Per a aquests animals, reduir la distància al menjar és equivalent a incrementar la intensitat de l'olor.

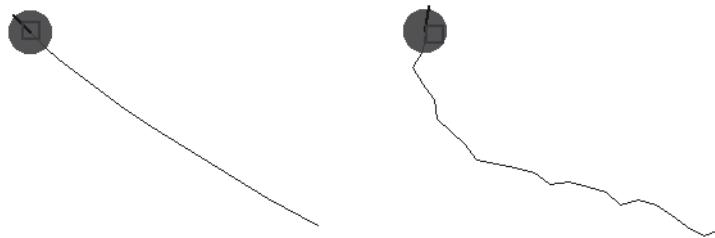
## Mantenir l'orientació

De fet, una bona manera per a un animal d'arribar a la font d'aliment és saber exactament on està localitzada. En aquest cas, la millor estratègia és “no deixar de mirar el premi” sense perdre de vista el menjar i movent-se en la seva direcció. Implementeu aquesta aproximació utilitzant el mètode apuntaA: i introduïu alguns moviments aleatoris per fer la simulació una mica més realista.

Ara podeu afegir la noció de velocitat i pertorbació de la trajectòria de l'animal. Definiu un mètode mantenirOrientació: unRectangle movent: unaDistancia girant: unAngle que manté l'animal sempre apuntant al centre de la font d'aliment movent-se i girant una quantitat constant. Les figures 23.13 i 23.14 mostren alguns resultats d'aquesta aproximació. Amb aquestes restriccions, podeu endevinar quina és la manera més eficient d'aconseguir el menjar: córrer molt i girar un angle gran o anar a poc a poc i girar un angle petit? Naturalment, aquesta simulació no considera que el menjar també es pot moure.



**Figura 23.13** — Trobar menjar sense perdre'l de vista. Esquerra: velocitat 5 píxels girant 45 graus. Dreta: velocitat 5 píxels girant 60 graus.

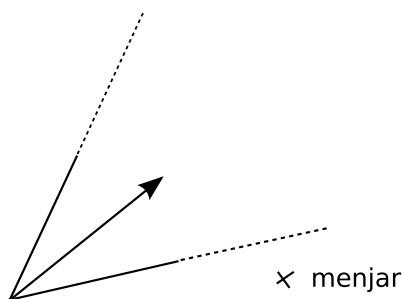


**Figura 23.14** — Trobar menjar sense perdre'l de vista. Esquerra: velocitat 15 píxels girant 5 graus. Dreta: velocitat 5 píxels girant 60 graus amb angle aleatori.

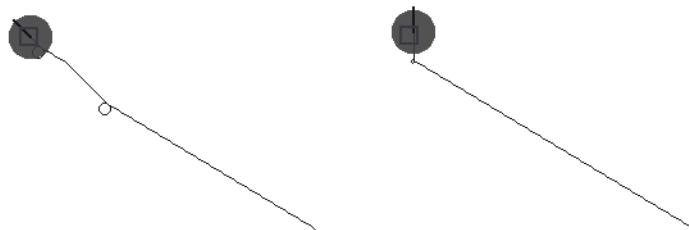
L'esquerra de la figura 23.14 mostra que ser capaç d'apuntar al menjar amb una petita pertorbació és una de les maneres més ràpides d'arribar-hi. La velocitat, però, ha de ser raonable. Feu alguns experiments amb velocitats altes per veure si això que acabem de dir és cert. Per aconseguir simulacions més realistes, introduïu aleatorietat a l'angle i la velocitat de manera que pugueu representar factors com el vent. Per exemple, hem introduït una mica d'aleatorietat a la dreta de la figura 23.14. A més de la introducció d'aleatorietat, implementeu la possibilitat que l'angle pugui variar en les dues direccions, tal com ja hem discutit en aquest capítol.

## Simular la visió

A l'experiment anterior, l'animal podia localitzar el seu menjar sense restriccions. Ara voldríem afegir una mica de realisme a la simulació. Quan l'animal identifica la font d'aliment amb la vista, té un cert angle de visió restringit que l'impedeix d'anar simplement directe cap al menjar. Imagineu que el nostre animal té ara un sol ull que és representat per un angle de visió dins del qual l'animal pot percebre el menjar, com veieu a la figura 23.15.



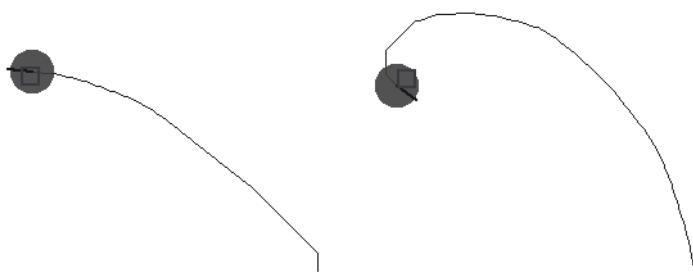
**Figura 23.15 —** El menjar està situat fora de l'angle de visió, i l'animal no el pot veure.



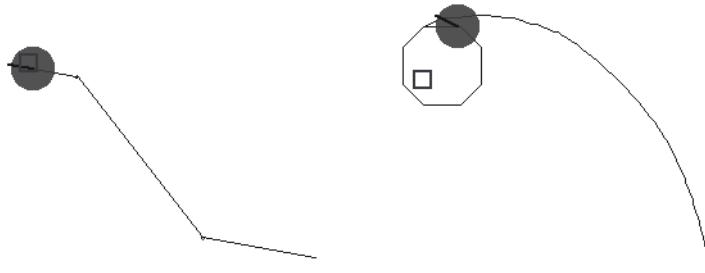
**Figura 23.16 —** Trobar menjar girant només quan el perdem de vista. Esquerra: angle de visió 10 graus, girant 15 graus. Dreta: angle de visió 40 graus, girant 60 graus.

Per definir aquest comportament, utilitzarem el mètode `anglePerApuntarA:`, que retorna l'angle que l'animal hauria de girar per apuntar cap a un punt donat. Aleshores podríeu decidir, per exemple, que si l'animal no veu menjar, giri, i si veu menjar hi vagi directament. Hauríeu de ser capaços de determinar si l'angle que l'animal ha de girar és inferior a la meitat del seu angle de

visió. Això és el que l'expressió (self anglePerApuntarA: unRectangle center) abs < (angleVisio / 2) fa al Mètode 23.9. L'expressió (self anglePerApuntarA: unRectangle center) abs retorna l'angle entre la direcció actual de l'animal i el seu menjar. Ara poseu totes aquestes idees juntes i definiu el mètode miraTrobaMenjarA: unRectangle angleVisio: angleVisio girant: unAngle que implementa aquest comportament. El Mètode 23.9 proporciona una possible solució, però proveu de trobar la vostra pròpia solució. Podeu veure algunes traces a les figures 23.16 i 23.17.



**Figura 23.17 —** Trobar menjar girant només quan el perdem de vista. Esquerra: angle de visió 15 graus, girant 2 graus. Dreta: angle de visió 35 graus, girant 3 graus.



**Figura 23.18 —** Trobar menjar girant només quan el perdem de vista. Esquerra: angle de visió 35 graus, girant 80 graus. Dreta: angle de visió 45 graus, girant 2 graus.

Aquesta aproximació és una mica ingènua, ja que pot girar indefinidament al voltant del menjar (figura 23.18, dreta). De fet, el canvi en l'angle no necessàriament porta a una millor situació. Us suggerim que feu que l'animal es mogui contínuament i no només giri quan no veu

el seu menjar.

#### Mètode 23.9

**mirarTrobaMenjarA: unRectangle angleVisio: angleVisio girant: unAngle**

```
[ (unRectangle containsPoint: self center)
  or: [ self anyButtonPressed ] ] mentreFals:
  [ ( self anglePerApuntarA: unRectangle center) abs < (angleVisio / 2)
    siCert: [ self ves: 15 ]
    siFals: [ self gira: unAngle ] ]
```

## Resum

Les diferents aproximacions al comportament animal presentades en aquest capítol són senzilles, però ja és possible obtenir alguns resultats interessants. En general, la introducció de perturbacions ajuda a simular el comportament real. Per a d'altres projectes, podríeu combinar diversos comportaments i provar de lligar-hi determinades entrades, com l'angle de visió. Molts altres aspectes del comportament animal poden ser modelitzats prenent com a punt de partida els exemples d'aquest capítol. Us desitgem molta diversió.

## **Part V**

# **Altres mons Squeak**



## Capítol 24

# Un recorregut per eToy

Aquest capítol presenta un breu recorregut pel sistema eToy. El sistema eToy proporciona una interfície per manipular objectes, per enviar-los missatges, comosar *scripts* gràfics i executar-los. S'utilitza a les escoles amb nens de 9 a 12 anys. Hi ha un llibre recent, *Powerful Ideas in the Classroom*<sup>1</sup>, que presenta detalladament com utilitzar eToy per ensenyar matemàtiques i ciència. Podeu trobar gran quantitat d'informació sobre eToy a <http://www.squeakland.org>.

En aquest capítol us ensenyarem com moure un avió amb un *joystick*, com crear animacions, com conduir un cotxe i finalment com programar un cotxe per seguir la carretera automàticament. Per a totes aquestes tasques haurieu d'obrir un projecte *morphic* utilitzant el menú del Món (**World**, opció **obrir**). Si esteu utilitzant l'entorn BotsInc. heu de fer un canvi molt simple per obtenir el menú complet d'Squeak. Trieu l'opció **ajuda...** i després **reinstal·lar el menú original**.

Ara obriu un projecte *morphic* obrint altre cop el menú **World** i triant **obrir...** seguit de **morphic project**. Haurieu d'obtenir una petita finestra. Feu clic per entrar dins el projecte. Podeu tornar utilitzant l'opció **projecte previ** del menú **World**. Un cop dins del nou projecte, haurieu de triar **pestanyes...** i instal·lar les solapes compartides per defecte (opció **default shared flaps**). Trigarà una miqueta, però obtindreu noves solapes, en particular unes anomenades *widgets* i *supplies* a la part de baix de la pantalla.

Si utilitzeu Squeak directament<sup>2</sup>, trieu **open...** del menú **World** i després l'opció **morphic project**, i entreu a la finestra petita que us apareixerà. Doneu un nom al vostre projecte i feu clic a sobre per entrar.

---

<sup>1</sup>B.J.Allen i Kim Rose, *Powerful Ideas in the Classroom: Using Squeak to Enhance Math and Science Learning* (Viewpoints Research Institute, 2003)

<sup>2</sup>Nota del Traductor: Aleshores us sortiran totes les opcions del menú en anglès, no en català amb algunes opcions en anglès, que és el que heu trobat si heu utilitzat l'entorn BotsInc. en català que estem utilitzant en aquest llibre.

## Pilotar un avió

Per pilotar un avió, primer heu de crear un avió. Després haureu d'obtenir un *joystick* i crear un *script* que connecti l'avió amb el *joystick*.

### Pas 1: Dibuixar un avió

Per obtenir un avió, el millor que podeu fer és dibuixar-lo vosaltres mateixos. Obriu la solapa blava anomenada *widgets* (figura 24.1) i arrossegueu la paleta o editor de dibuixos de la solapa a la pantalla. Aquesta acció obre una eina anomenada *Paint*, per pintar i dibuixar a la pantalla.



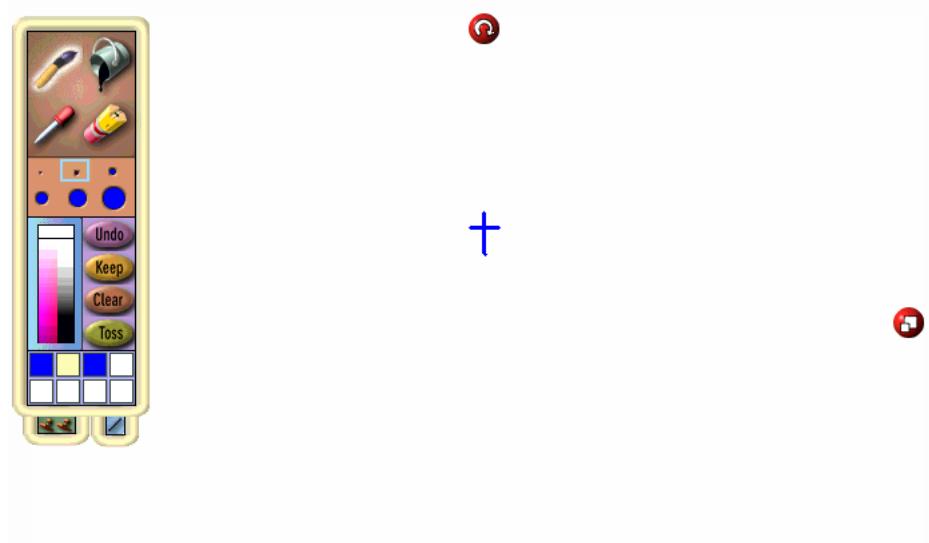
**Figura 24.1 — Obriu la solapa dels widgets per obtenir l'eina Paint**

Utilitzant *Paint*, dibuixeu un petit avió, com veieu a la figura 24.2. El nostre avió és una petita creu, però dibuixeu el vostre com us vingui de gust. Un cop heu acabat el dibuix, premeu el botó *Keep*. L'editor de dibuixos desapareixerà, i a la pantalla quedarà un esbós (*sketch*) que sembla un avió. L'avió que heu dibuixat s'anomena *jugador* en l'argot eToy.

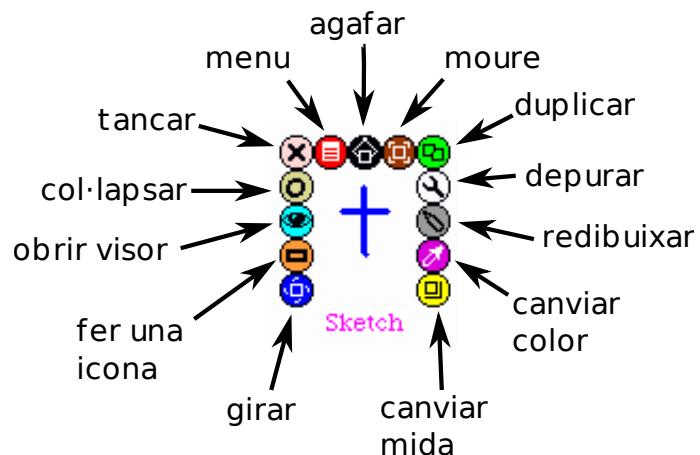
### Pas 2: Jugar amb l'halo

Ara, si feu clic sobre del vostre dibuix amb el botó dret del ratolí (o l'equivalent), veureu aparèixer un halo al voltant del dibuix, com es mostra a la figura 24.3. Cada nansa de l'halo té un color, una icona i una funció. Aquí les explicarem breument. Fixeu-vos que diferents tipus de dibuixos faran aparèixer diferents tipus de nanses. Podeu obtenir una descripció de la nansa aturant el ratolí al damunt durant un moment.

- La nansa rosa amb una creu destrueix el dibuix. En funció de les preferències, el dibuix anirà a la paperera o serà completament eliminat. Si va a parar a la paperera, podeu recuperar-lo i tornar-lo a utilitzar.
- La nansa vermella amb rectangles petits fa aparèixer el menú associat al dibuix. El menú depén del dibuix amb què esteu interaccionant. Proporciona moltes accions per manipular el dibuix.



**Figura 24.2 — Dibuixar un avió amb l'eina Paint**

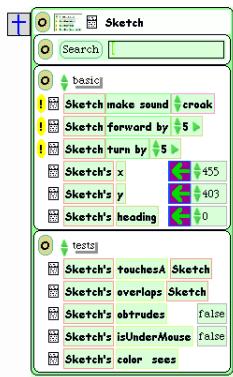


**Figura 24.3 — Un halo envolta un dibuix**

- La nansa negra selecciona el dibuix. Fixeu-vos que seleccionant un dibuix canvieu el seu *contenidor*. Uilitzeu la nansa marró per moure el dibuix dins del contenidor. Utilitzeu la nansa vermella per inserir el dibuix dins del dibuix que té a sota. La idea és que podeu inserir un *morph* dins del *morph* que té a sota obrint la nansa vermella i seleccionant l'opcio del menú **embed into**. Aquesta operació us ofereix un *morph* on inserir el vostre *morph*. Un cop ho heu fet, movent el *morph* on heu inserit el vostre *morph* es mouran els dos *morphs*. Amb la nansa negre podeu “des-inserir” el vostre *morph* del seu contenidor, mentre que amb la nansa marró, podeu moure el *morph* només dins del contenidor.
- La nansa marró amb un quadrat mou el dibuix sense canviar el seu contenidor.
- La nansa verda amb dos quadrats duplica el dibuix.
- La nansa gris amb una eina dibuixada ofereix possibilitats de depuració, que normalment són utilitzades per “squeakers” experts.
- La nansa de color gris fosc té un llapis amb el qual podeu repintar el dibuix.
- La nansa de color rosa fosc amb un comptagotes canvia el color del dibuix. No funciona amb dibuixos com el nostre aviò. Per canviar el color d'un dibuix creat per vosaltres, necessiteu la nansa gris fosc. Si trieu aquesta nansa, tornareu a l'eina *Paint*, on podeu fer qualsevol canvi sobre el vostre dibuix. La nansa rosa fosc és més apropiada per canviar els colors de les fonts, del rectangles o dels costats.
- La nansa groga amb un quadrat i una barra canvia la mida del dibuix.
- La nansa blava amb un quadrat petit serveix per girar el dibuix.
- La nansa de color taronja amb un rectangle petit produeix una icona que representa l'objecte pel sistema d'ícones d'eToy (més endavant explicarem més detalls).
- La nansa cian amb un ull obre un *visualitzador* (*viewer*) per al dibuix. El visualitzador presenta una representació gràfica dels mètodes i les variables d'instància de l'objecte.
- La nansa de color verd pà·lid amb un cercle col·lapsa el dibuix.

Per programar dins l'entorn eToy utilitzem un visualitzador. Així, obriu un visualitzador de l'aviò fent clic a la nansa cian amb un ull. Haurieu d'obtenir les eines que podeu veure a la figura 24.4

La figura 24.5 explica els principals components d'un visualitzador. A dalt de tot, el nom del dibuix (*sketch*) pot ser modificat. Per defecte el vostre dibuix s'anomena Sketch. Us suggerim que l'anomeneu “aviò” editant el nom al visualitzador A més de canviar el nom, podeu obrir un menú, com veieu a la figura. Amb aquest menú podeu afegir noves variables a l'objecte, obtenir un *script* buit o aconseguir una icona representant l'objecte.



**Figura 24.4 — Obrir un visualitzador**

Al visualitzador es mostren les categories. El nom d'una categoria es mostra a la dreta de dos triangles verds que apunten amunt i avall. A la figura 24.5 podeu veure les categories basic i tests. Podeu explorar-les utilitzant els petits triangles dobles. També podeu fer clic al mateix nom de la categoria per obtenir una llista de totes les categories.

Sota cada categoria podeu veure una llista de mètodes. Quan un mètode pot ser executat, hi ha un signe d'admiració groc al costat. Premeu el signe d'admiració davant del mètode forward i veure l'avió movent-se cap endavant. Hi ha també mètodes que només accedeixen a o canvien el valor d'una variable. Anomenem a aquests mètodes *d'accés*: són mètodes *getter*, per accedir als valors, o mètodes *setter* per modificar-los.

Fixeu-vos que podeu modificar el valor d'una variable directament utilitzant els triangles de color verd, o escrivint un valor directament sobre la capsula de la variable. Proveu, per exemple, de canviar la variable *x* amb el valor 800. Dins un *script*, com que no podeu canviar interactivament el valor d'una variable, haurieu d'utilitzar mètodes *getter* i *setter*. Podeu obtenir el mètode *getter* per a la variable *x* arrosegant la capsula *x* a l'escriptori. Per obtenir el mètode *setter*, haurieu d'arroseggar la fletxa verda gran, com veieu a la figura 24.6.

El nombre a la dreta de la fletxa verda gran és el valor de la variable. Si moveu l'avió utilitzant la nansa negra o marró, veureu les variables *x* i *y* canviar els seus valors.

També podeu tenir *observadors* (*watchers*), que proporcionen una manera d'espíar els valors de les variables. Per crear un observador, podeu fer clic a la icona petita de menú que hi ha a l'esquerra d'una variable del dibuix. Haurieu d'obtenir el menú que veieu a la figura 24.7. Després podeu triar crear un observador senzill, que mostrarà només el valor d'una variable, o un observador detallat, que podeu utilitzar per canviar el valor d'una variable espiada.

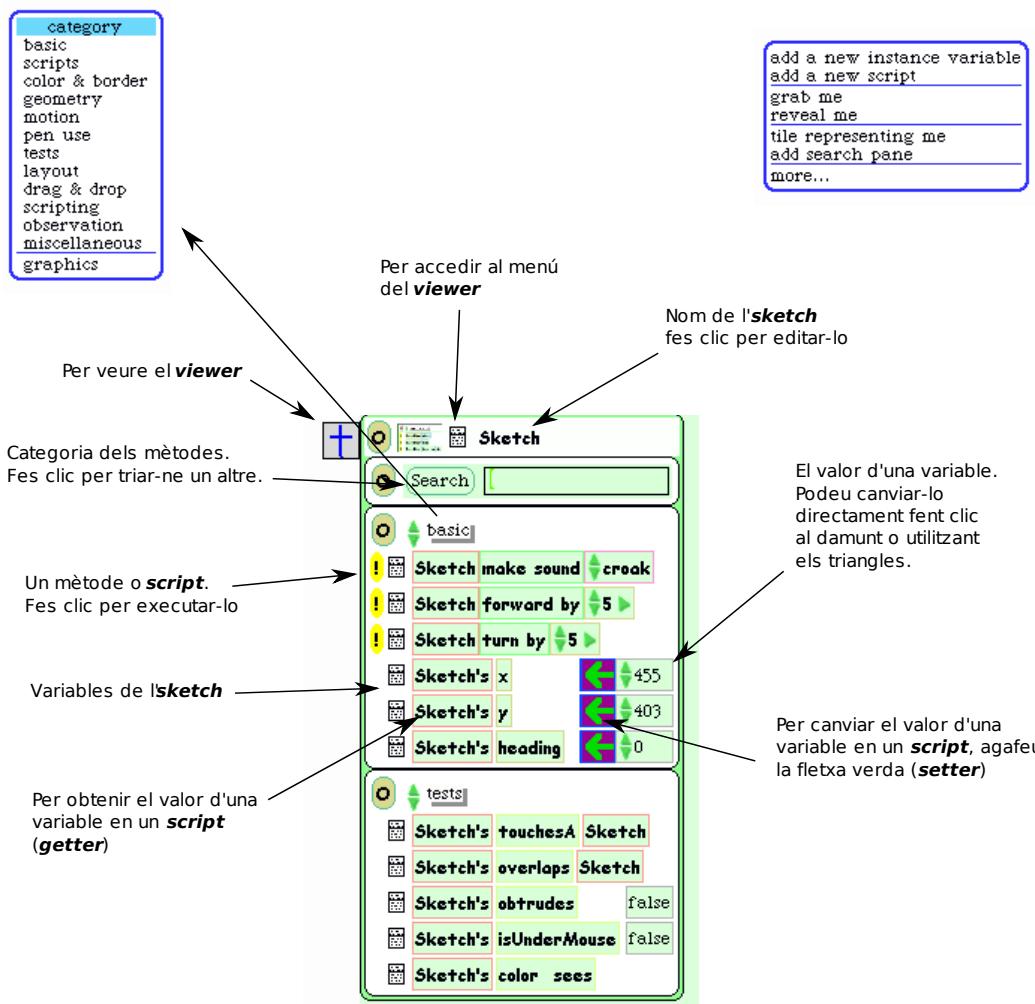
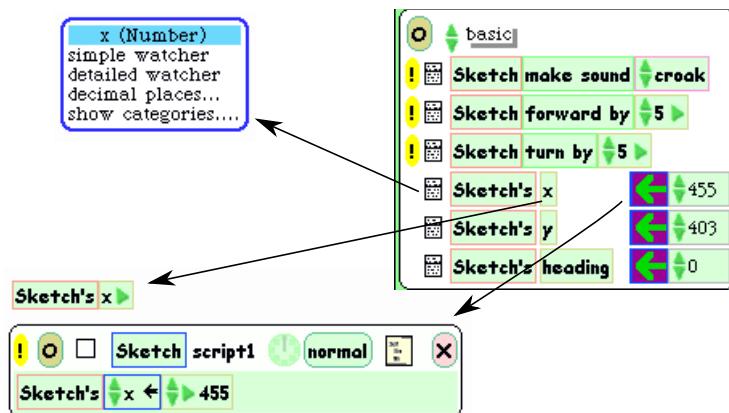


Figura 24.5 — Entendre un visualitzador



**Figura 24.6 — Accedir a les variables dins d'un script**



**Figura 24.7 — Els observadors: Espiar les vostres variables**

### Pas 3: Arrossegar i deixar un mètode per crear nous scripts

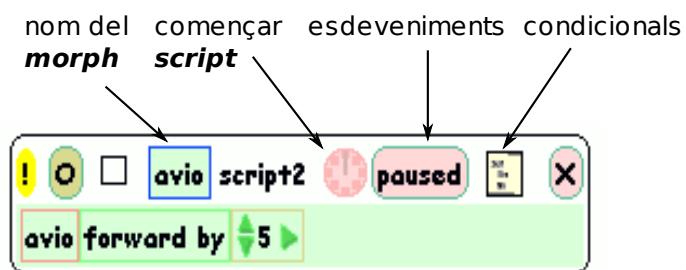
Si preneu un mètode com forward by del visualitzador i el deixeu sobre l'escriptori, ja teniu un *script*. Hauríeu d'obtenir, per exemple, l'*script* que veieu a la figura 24.8.



**Figura 24.8 — Creem un script**



**Figura 24.9 — Quan el rellotge està en marxa, l'script s'està executant**



**Figura 24.10 — Les diferents parts d'un script**

Per executar l'*script* només cal que feu clic sobre la icona del rellotge. El rellotge es posa en marxa (estat *ticking*, veure figura 24.9), la qual cosa vol dir que el vostre *script* s'executa a intervals de temps regulars.

La figura 24.10 mostra les diferents parts d'un *script*: podeu veure el nom del dibuix, el nom de l'*script*, un rellotge per començar i aturar l'execució de l'*script*, una llista d'esdeveniments als quals pot enllaçar-se un *script*, i la creació d'un condicional, o *test*.

#### Pas 4: Afegir mètodes

Quan executeu l'*script*, podeu veure que l'avió vola en línia recta. Sens dubte voldríeu que el vostre avió fos capaç de girar. Per aconseguir això, us cal afeir mètodes a l'*script*. Arrossegueu el mètode *turn by* del visualitzador i deixeu-lo anar dins de l'*script*. L'*script* us hauria de mostrar el lloc on quedàr el nou mètode quan el deixeu anar, ensenyant-vos unes capses de color verd. Deixeu anar el mètode en una de les capses. Hauríeu d'obtenir un *script* similar al que veieu a la figura 24.11.

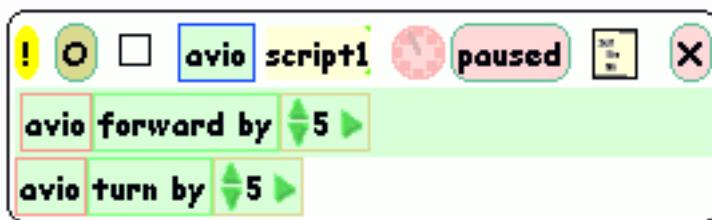
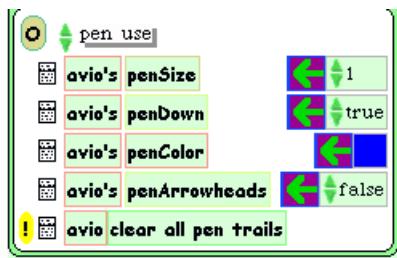


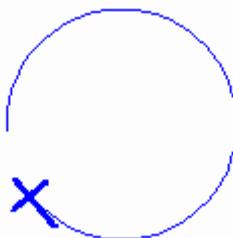
Figura 24.11 — El mètode *turn by* ha estat afegit a l'*script*

Ara, quan executeu l'*script*, l'avió hauria de girar en un cercle. De fet, un dibuix és similar a un robot, i podeu veure el que l'avió està fent dient-li que baixi el llapis. Per fer això, busqueu la variable *penDown* dins del visualitzador i poseu el seu valor a *true*, com a l'*script* que veieu a la figura 24.12. El resultat es mostra a la figura 24.13.

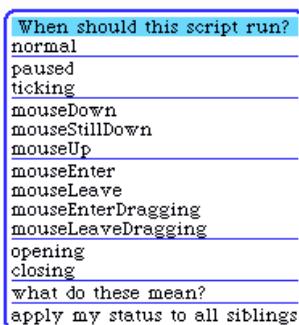
Fixeu-vos que podeu triar quan s'executarà l'*script* utilitzant el botó d'esdeveniments a l'*script*. La figura 24.14 mostra tots els esdeveniments que podeu fer servir. Obtindreu aquest menú fent clic dins el text a la dreta del rellotge.



**Figura 24.12** — Un cop el llapis és avall, l'avió dibuixarà una traça a l'escriptori, com veieu a la figura 24.13.



**Figura 24.13** — L'avió ha baixat el seu llapis, i podem veure que vola en un cercle.



**Figura 24.14** — Una llista d'esdeveniments disponibles.

## Joysticks en acció

Volar en cercles és divertit, però probablement us agradaría condir el vostre avió. Podeu variar l'angle de gir de l'avió, en lloc de girar sempre el mateix angle, associant-lo a l'estat d'un *joystick*.

### Pas 1: Crear un joystick

Primer creeu el *joystick* arrossegant-ne un de la solapa *supplies* i deixant-lo anar a sobre de l'escriptori. El cercle vermell representa el capdamunt del *joystick* (figura 24.15), i amb ell podeu indicar quina direcció i quina quantitat d'energia volem posar en el moviment de l'avió.



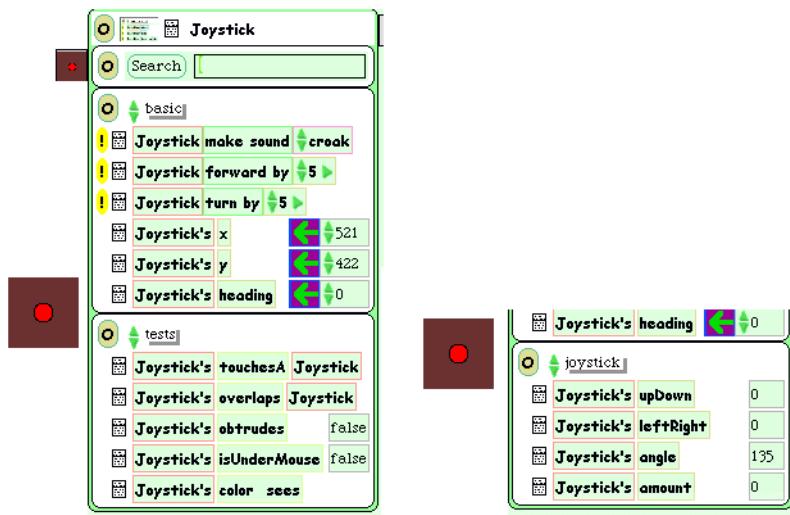
Figura 24.15 — Crear un joystick.

### Pas 2: Experimentar amb un joystick

Segon, obriu el visualitzador del *joystick* utilitzant la nansa apropiada. Després, explorant els mètodes, en trobareu alguns d'útils sota la categoria *joystick* (figura 24.16). Feu moviments amb el *joystick* i observeu les variables. La variable *amount* representa la quantitat d'energia invertida en el moviment; és a dir, podeu triar a més de la direcció en què es mou el *joystick*, la força del moviment. La variable *angle* representa la direcció en la qual esteu movent el *joystick*, i us deixarem endevinar el que representen les altres dues variables.

### Pas 3: Vincular el joystick i l'script

Per poder condir l'avió, heu de canviar el valor del mètode *turn by* a l'script pel valor donat pel *joystick*. Per a això, la variable *leftRight* sembla ser la que necessitem. Arrossegueu la variable directament sobre l'argument del mètode *turn by* a l'script. Pot costar-li un moment acceptar la variable, però hauríeu d'aconseguir l'script de la figura 24.17.



**Figura 24.16 — Mètodes sota la categoria joystick.**



**Figura 24.17 — Ara, la variable del joystick leftRight controlarà el gir de l'avió.**

Ara feu clic sobre el rellotge i conduïu el vostre avió amb el joystick. Com podeu veure, si controléssim la velocitat de l'avió en milloraríem la conducció. Proveu de trobar-hi una solució. Considereu la variable amount; us pot ajudar.

## Crear una animació

La idea per crear una animació és la següent: primer dibuixeu els marcs individuals de l'animació i els poseu en un contenidor d'animacions. Després creeu un dibuix senzill, la representació gràfica del qual serà sustituïda pels elements de l'animació. Per fer això podeu escriure un *script* que fa que el dibuix sembli els diferents marcs que hi ha al contenidor.

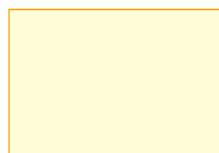
### Pas 1: Crear el contenidor

Per crear una animació, heu de crear un contenidor (*Holder*). Un contenidor és un objecte gràfic que pot contenir altres objectes gràfics. També sap quin és l'objecte seleccionat d'entre tots els que conté. Per crear un contenidor, arrossegueu-lo de la solapa anomenada *supplies* (figura 24.18) i deixeu-lo anar sobre l'escriptori.



**Figura 24.18** — Podem fer servir la solapa vermella per crear un contenidor.

Es crea un contenidor buit (figura 24.19).



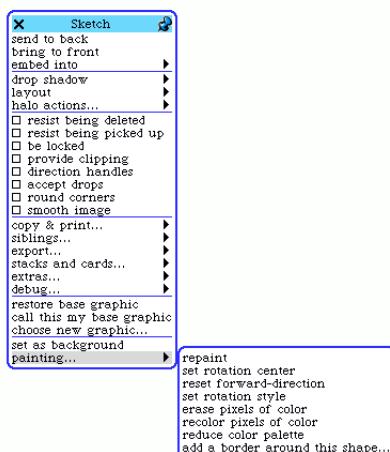
**Figura 24.19** — Un contenidor buit.

## Pas 2: Dibuixar els elements de l'animació

Pel segon pas, hauríeu de dibuixar allò que vulgueu animar utilitzant l'editor de dibuixos *Paint* que ja hem vist abans. Us recomanem que primer feu un dibuix, i el dupliqueu amb la nansa verda. Després seleccioneu la nansa gris fosc per redibuixar el que heu fet. D'aquesta manera, podeu crear diversos dibuixos modificant-ne un pas a pas. Ara pintarem un cuc en dues posicions diferents (figura 24.20).



**Figura 24.20 — Un cuc en dues posicions diferents.**

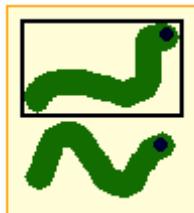


**Figura 24.21 — El menú de pintar de la nansa vermella.**

Fixeu-vos que també podeu utilitzar l'opció de pintar del menú de la nansa vermella (figura 24.21), però us suggerim que utilizeu les nances proporcionades tant com us sigui possible.

### Pas 3: Deixar els dibuixos dins del contenidor

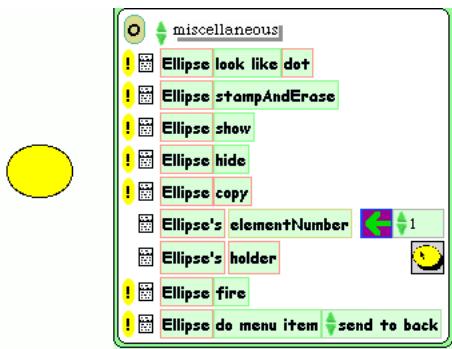
El proper pas és simplement posar els dibuixos dins del contingidor. Un rectangle negre (figura 24.22) representa el dibuix actualment seleccionat dins del contingidor.



**Figura 24.22** — El cuc del rectangle negre és el dibuix seleccionat.

### Pas 4: Crear un dibuix senzill com a base de l'animació

Ara us cal un dibuix que s'utilitzarà com a receptacle de l'animació. Per tant, haurieu de crear un dibuix senzill, com una el·lipse, que agafareu i arrossegareu de la solapa *supplies*. Obriu després un visualitzador d'aquest dibuix, com veieu a la figura 24.23.



**Figura 24.23** — Un visualitzador del receptacle del dibuix.

### Pas 5: Crear un *script* amb lookLike

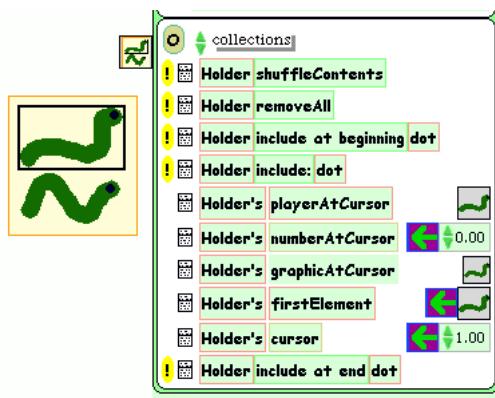
A partir del dibuix senzill, aquí l'el·lipse, podeu crear un nou *script* arrossegant el mètode lookLike dot (mostrat a la figura 24.23 del visualitzador a l'escriptori. Aquesta acció hauria d'haver creat l'*script* descrit a la figura 24.24.



**Figura 24.24 — Un script amb lookLike dot**

### Pas 6: Mostrar l'element seleccionat de l'animació

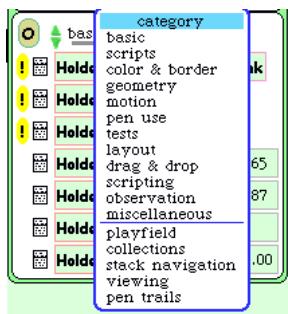
Ara hauríeu d'indicar que l'el·lipse s'hauria d'assemblar a l'element actualment seleccionat del contingidor. Seleccioneu el contingidor i obriu-lo en un visualitzador, com veieu a la figura 24.25.



**Figura 24.25 — El contingidor obert en un visualitzador.**

Busqueu la categoria collection triant “basic”, com veieu a la figura 24.26.

De la llista de mètodes arrossegueu el mètode anomenat playerAtCursor, que retorna el jugador, és a dir, l'element gràfic dins el contingidor que és a la posició actual. Deixeuh-lo anar dins



**Figura 24.26 — Prement el botó “basic” us permet accedir a collection.**

del quadrat amb un punt al mig (aquesta capsula representa l’argument del mètode lookLike). Obtindreu l’script de la figura 24.27.



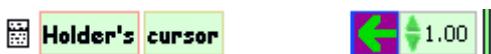
**Figura 24.27 — Ara l’el·lipse hauria de tenir l’aparença de l’element apuntat pel cursor.**

### Pas 7: Canviar l’element seleccionat d’un contenidor

Si executeu l’script previ utilitzant el botó *ticking*, l’script canviàrà la forma de l’el·lipse per l’element gràfic actualment seleccionat. Encara, però, no tenim cap animació. Per a això, ens cal trobar una manera de canviar l’element seleccionat actualment d’un contenidor. De fet, un contenidor té un índex, anomenat *cursor*, que representa el nombre associat a l’element seleccionat. N’hi ha prou amb incrementar aquest índex per fer que el contenidor seleccioni un altre element gràfic.

Per canviar el valor de la variable anomenada *cursor*, arrossegueu la fletxa (mostrada a la dreta de la figura 24.28) i poseu-la a la línia següent de l’script que ja hem vist. Obtindreu l’script de la figura 24.29, que diu que la variable conté el valor 1.

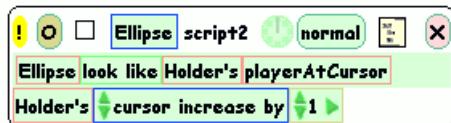
Ara, si feu clic als triangles dobles de color verd del *setter* a l’script, veureu que podeu incrementar el valor del *cursor* en una certa quantitat (figura 24.30). Ara l’animació hauria de funcionar.



**Figura 24.28 — El cursor dins el jugador.**



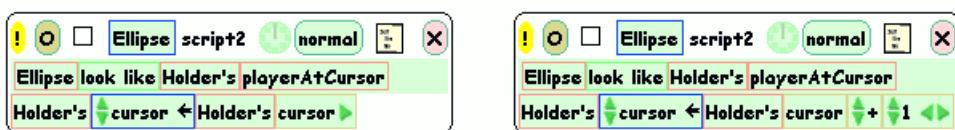
**Figura 24.29 — Introduir una assignació.**



**Figura 24.30 — Incrementar el cursor.**

## Una altra manera

Per aconseguir el mateix efecte, podeu simular l'expressió `cursor := cursor + 1`, que mourà el cursor al proper element. Per tant, arrossegueu la part esquerra del cursor des del visualitzador fins el lloc de l'`1` a l'`script`, i obtindreu l'`script` de la figura 24.31.



**Figura 24.31 — Utilitzar l'expressió `cursor := cursor + 1`.**

Ara només us cal fer clic al triangle verd i petit al final de la capsula del cursor a l'`script` per fer aparèixer l'expressió `+ 1`, com es veu en l'`script` final. La primera solució, però, és més senzilla.

Us recomanem que els lectors interessats a utilitzar eToy a classe llegiu el llibre esmentat

al principi del capítol. Presenta molts aspectes pedagògics que poden utilitzar-se amb aquest exemple.

## Cotxes i conductors

Ara ens agradaria ensenyar-vos com podeu construir un *script* per controlar un cotxe amb un volant. Aquest exercici pot ser útil als mestres per ensenyar l'ús de la divisió com a desmultipliador, tal com es fa en cotxes reals, però també es pot fer servir per diversió.

### Pas 1: Dibuixar un cotxe i un volant

Utilitzant l'editor de dibuixos, dibuixeu un cotxe i un volant, com els de la figura 24.32.



**Figura 24.32 — Dibuixeu un cotxe i un volant.**

### Pas 2: Girar el cotxe en un cercle

Obriu un visualitzador per al cotxe, canvieu-li el nom pel de “cotxe” i arrossegueu i deixeu anar el mètode forward by tal com ja heu fet abans. S'hauria d'obtenir l'*script* mostrat a la figura 24.33.



**Figura 24.33 — Arrossegueu i deixeu anar el mètode forward.**

Després arrossegueu i deixeu anar el mètode turn sota el mètode anterior a l'*script* que acabeu de crear. Hauríeu de tenir l'*script* de la figura 24.34, que quan s'executa fa que el cotxe doni voltes en un cercle.



**Figura 24.34 — Arrossegueu i deixeu anar el mètode turn.**

### Pas 3: Utilitzar la direcció del volant

Ara heu de vincular l'angle que gira el cotxe amb la posició del volant. Hem de trobar una manera de girar el volant i determinar quina quantitat ha girat. El primer problema és fàcil de resoldre: feu aparèixer l'halo, feu clic sobre la nansa blava i gireu (figura 24.35). Això fa que giri el volant. Per resoldre el segon problema, obriu un visualitzador del volant, canvieu-li el nom per "volant", i busqueu la variable heading dins el visualitzador (l'expressió serà volant's heading). Quan gireu el volant amb la nansa, el valor de la variable canvia. Aquesta variable representa l'angle de rotació respecte del dibuix original.



**Figura 24.35 — El volant amb el seu halo de nanses.**

Ara que ja teniu totes les peces del trencaclosques, heu de dir-li al cotxe que giri no una quantitat determinada, sinó el que digui la direcció del volant. Per fer això, arrossegueu l'expressió volant's heading des del visualitzador fins al nombre 5, argument del mètode turn de l'script anterior. hauríeu de tenir l'script de la figura 24.36.

Si fem clic sobre el petit rellotge de l'script, el cotxe corre i podem controlar-lo des de la nansa blava de l'halo del volant. Us adoneu, però, que el seguiment no és perfecte. Els mestres haurien de proposar que els estudiants analitzessin el problema i suggerissin solucions.

La qüestió és que el valor de la direcció del volant s'hauria de dividir de manera que l'usuari pogués tenir un control més precís sobre el gir. Per fer-ho, feu clic sobre el petit triangle verd de



**Figura 24.36 — Ara ja podem girar el cotxe amb el volant.**

més a la dreta dins del bloc de direcció del volant. Això afegeix més capses. Feu clic per triar la divisió  $/$ . Ara teniu un *script* similar al de la figura 24.37.



**Figura 24.37 — Dividint el valor de la direcció del volant aconseguim un control més precís.**

El cotxe, el volant i els halos apareixen tots a la figura 24.38. Divertiu-vos controlant el cotxe!



**Figura 24.38 — Divertiu-vos amb el cotxe!**

Ara voldríem programar el cotxe de manera que pugui anar sol, automàticament, per una carretera. La idea és equipar al cotxe amb sensors que li diguin si marxa de la carretera. Un cop

el cotxe hagi estat equipat amb sensors i convenientment programat, el posarem en una carretera, a veure què tal la segueix.

No us mostrarem una solució completa d'aquest problema ja que pensem que us agradarà experimentar vosaltres mateixos. És més, la nostra solució no funciona gaire bé.

### Pas 1: Sensors

A eToy podeu preguntar si un determinat color d'un dibuix pot veure un altre color que tingui a sota. Aquesta possibilitat pot ser utilitzada per construir un sensor. Un sensor serà un simple punt d'un color que li dirà al cotxe si està passant per sobre d'un altre color. Per equipar el vostre cotxe amb sensors, redibuixeu el cotxe i afegiu dos punts de colors, com veieu a la figura 24.39.



**Figura 24.39 — Hem afegit dos sensors al cotxe.**

### Pas 2: La carretera

Utilitzant l'editor de dibuixos, feu una carretera d'un sol color. Després podeu provar de posar-hi diverses bandes de colors.

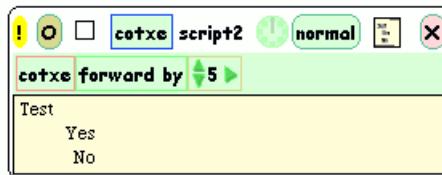
### Pas 3: Condicions i tests a eToy

Obriu el visualitzador del cotxe i arrossegueu i deixeu anar el mètode forward by sobre l'escriptori. Això crea un *script* com aquells que ja esteu acostumats a crear.

Ara necessiteu una manera d'expressar diferents tipus de comportament en funció del valor del sensor. Per a això, hauríeu de poder expressar una condició, o test. Per obtenir un test, hauríeu d'arrossegar el quadrat petit (segona capsella per la dreta al capdamunt de l'*script*, prop del botó amb una creu), com veieu a la figura 24.41. Arrossegueu i deixeu anar aquesta petita icona de test dins de l'*script*. Hauríeu d'aconseguir un *script* similar al de la figura.



**Figura 24.40 — La carretera.**



**Figura 24.41 — Afegir un test a l'script.**

Ara us cal trobar una manera d'activar els vostres sensors. Trobeu el mètode color sees a la categoria test, com veieu a la figura 24.42. Aquest mètode retorna true o false en funció de si una part del dibuix d'un determinat color passa sobre un altre color.

Arrossegueu el mètode color sees dins de l'script al costat de la paraula Test. Hauríeu d'obtenir un script com el de la figura 24.43.

#### Pas 4: Personalitzar els tests basats en colors

Com definirem el color que hauria d'utilitzar el test? És fàcil. Només cal que feu clic al damunt del quadradet de color dins del mètode color sees. Això obre automàticament un triador de color (figura 24.44). Amb el triador de color, podeu triar el color d'un sensor. El primer color de l'script hauria de canviar per reflectir aquesta tria de color. Feu la mateixa operació per a la segona capsula de color, però aquest cop trieu el color de la carretera (figura 24.45).

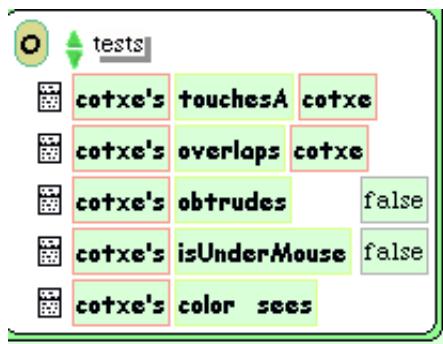


Figura 24.42 — Afegir un test a l'script.

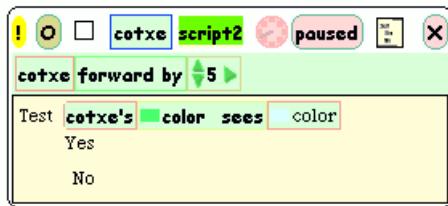


Figura 24.43 — Afegir un mètode color sees a l'script.

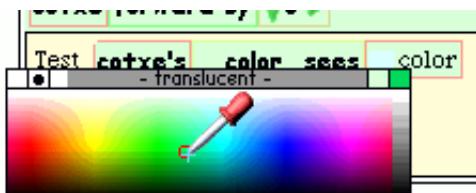
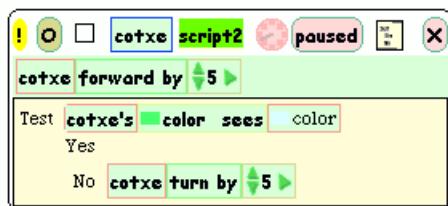


Figura 24.44 — Utilitzar el triador de color.

### Pas 5: Afegir accions

Ara podeu especificar que el cotxe hauria de girar quan el sensor no vegi el color de la carretera. Només cal que arrossegueu i deixeu anar el mètode turn by al costat de la paraula No a l'script.



**Figura 24.45 — Trieu el color de la carretera pel color que el cotxe veu.**

Ara podeu seleccionar el cotxe, posar-lo a la carretera i prémer el rellotge per executar l'script. Veieu la figura 24.46.

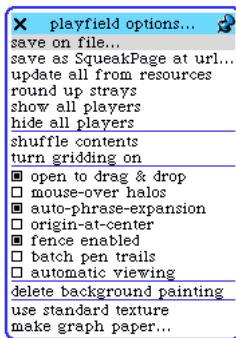


**Figura 24.46 — El cotxe segueix la carretera!**

És clar que el comportament del cotxe no és perfecte, i us deixem a vosaltres la possibilitat de canviar-lo o de trobar una solució pròpia.

## Alguns trucs

Ens agradaria ensenyar-vos alguns aspectes d'eToy que us poden ajudar. L'entorn eToy pot ser personalitzat, així que obriu el menú **playfield options...** (figura 24.47), deixeu que el ratolí s'aturi sobre les diferents opcions i llegiu-vos la bafarada d'ajuda. Algunes de les coses que podeu fer són:



**Figura 24.47 — El menú *playfield options...***

### Executar diversos *scripts*

Si creeu diversos *scripts*, s'executarán en paral·lel. Per controlar tots els *scripts* disponibles a l'escriptori, podeu fer servir el giny<sup>3</sup> anomenat “All Scripts”, que podeu obtenir de la solapa *widgets*. Un cop deixeu anar aquest giny sobre del escriptori, obtenuïu un tauler (figura 24.48) que us permet executar i aturar tots els *scripts* actualment a l'escriptori.



**Figura 24.48 — El tauler de control d'scripts.**



**Figura 24.49 — Els mètodes start script i scripting.**

<sup>3</sup>Nota del Traductor: Traducció de *widget* segons softcatalà.

Fixeu-vos que podeu utilitzar també el mètode start script i el mètode relacionat scripting a la mateixa categoria per començar, fer una pausa o aturar scripts (figura 24.49).

És interessant veure com un *script* pot invocar l'execució d'altres *scripts*. Això permet crear *scripts* complexos.

## Eliminar

Podeu eliminar tots els camins dibuixats pels vostres robots amb el mètode clear all pen trails, a la categoria pen use. També podeu eliminar les traces dels jugadors des de l'escriptori. Per a això, utilitzeu la darrera opció del menú **appearance**, que podeu obtenir del menú **World**.

## Crear una icona

Si voleu escriure un *script* que vinculi dos objectes, necessitareu referir-vos a aquests objectes. En aquest cas, us cal una *icona* (*tile*) que representi l'objecte a vincular per a que pugueu deixar-la anar sobre el vostre *script*. Per exemple. Supposeu que teniu dos avions, un de blau i un altre de vermell, i voleu que l'avió vermell segueixi l'avió blau. Per fer això, podeu utilitzar el mètode move toward, mostrat a la figura 24.50.



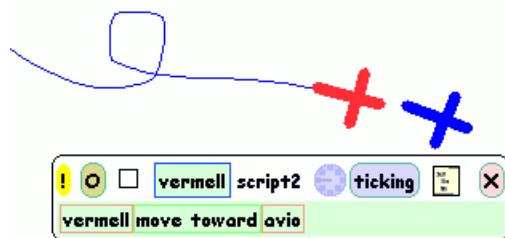
**Figura 24.50 — El mètode move toward.**

Com que voleu que l'avió vermell es mogui cap a l'avió blau, i no cap a un dot com a l'*script* anterior, primer utilitzeu la nansa taronja de l'avió blau per aconseguir una icona que el representi (figura 24.51).



**Figura 24.51 — Una icona que representa l'avió blau.**

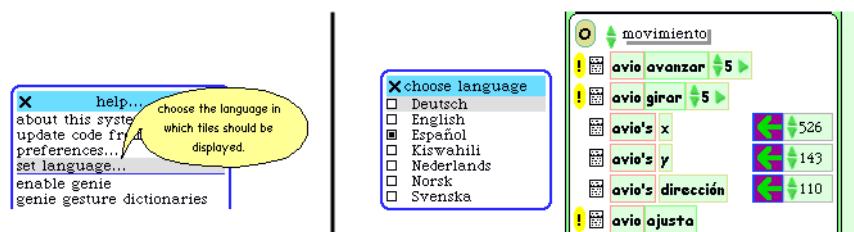
Després deixeu anar aquesta icona dins de l'*script*. Ara l'avió vermell segueix l'avió blau, com es pot veure a la figura 24.52.



**Figura 24.52 — El tauler de control d'scripts: l'avió vermell segueix l'avió blau.**

## Internacionalització

Si voleu utilitzar eToy amb nens petits, voldreu que tot aparegui en la seva llengua materna. L'interfície completa d'Squeak ha estat traduïda a uns quants idiomes. Obriu el menú **World** i trieu l'opció **help...** seguida de **set language...** (figura 24.53).



**Figura 24.53 — Esquerra: El menú per triar idioma. Dreta: Tria l'espanyol.**

## Resum

Aquest capítol ha presentat una petita mostra de les moltes possibilitats oferides per eToy. Us suggerim que visiteu <http://www.squeakland.org> i exploreu el material que hi podreu trobar.

## Capítol 25

# Un Recorregut per Alice

Alice és un entorn per construir mons 3D en Squeak. Alice és també un projecte de recerca l'objectiu del qual és proporcionar un entorn prou abstracte, fàcil d'aprendre i d'utilitzar per principiants. Squeak Alice és una versió d'Alice per a Squeak feta per Jeff Pierce. Podeu trobar una descripció detallada del sistema en un dels capítols del llibre *Squeak: Open Personal Computing and Multimedia*<sup>1</sup>. Squeak Alice està construït sobre Balloon, el motor 3D d'Squeak, que funciona en qualsevol plataforma, sense requeriments especials de maquinari. Aquest capítol presenta Squeak Alice, però dins el context d'aquest llibre.

Squeak Alice ve amb un entorn complet per manipular objectes 3D. Per desenvolupar *scripts* i interactuar amb objectes 3D, podeu crear un entorn nou, com explicarem després en aquest capítol, o utilitzar l'entorn que es proporciona per defecte dins l'entorn Squeak. Per començar de pressa, us suggerim que utilitzeu primer l'entorn ja definit. Després, podeu obrir i utilitzar els vostres caràcters 3D. Aquesta és l'estrategia que seguirem en aquest capítol.

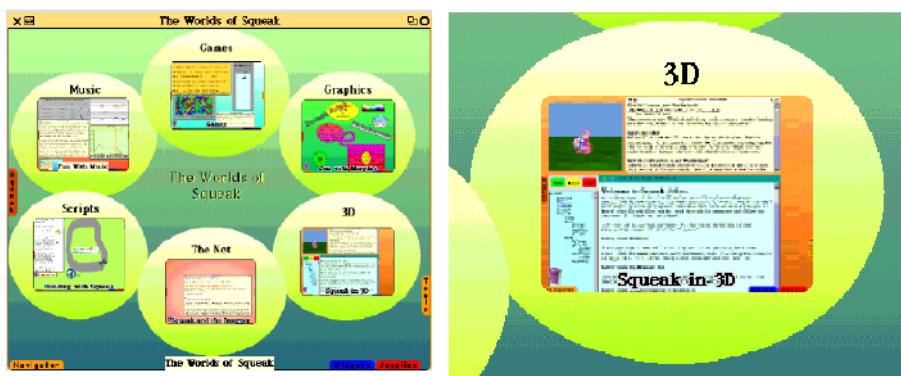
### Començar amb Alice

Quan comenceu a utilitzar un entorn Squeak estàndard, apareix una finestra anomenada *The Worlds of Squeak*<sup>2</sup>, com veieu a la figura 25.1. Si feu clic en aquesta finestra, anireu a parar a un lloc que conté diverses finestres petites. Cada una d'aquestes finestres representa una demostració d'algun aspecte d'Squeak.

---

<sup>1</sup>Mark J. Guzdial i Kimberly M. Rose, *Squeak: Open Personal Computing and Multimedia* (Prentice Hall, 2001).

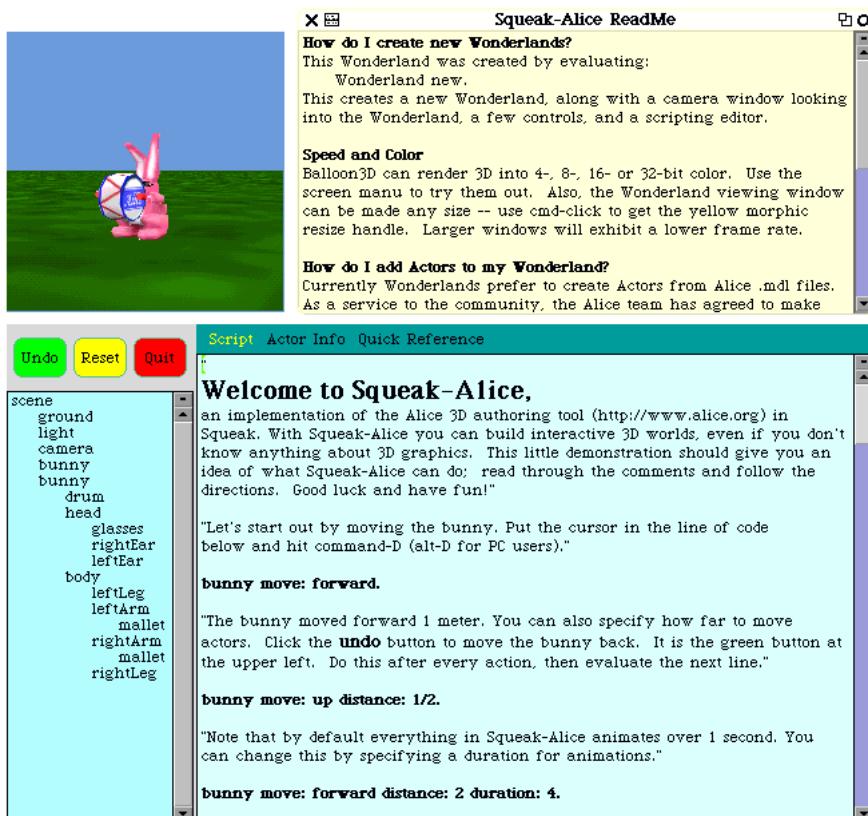
<sup>2</sup>Nota del Traductor: Això passava quan es va escriure aquest llibre, als voltants de 2005. La versió actual d'Squeak, la 3.9 quan estic escrivint això, ja no té aquesta finestra inicial. Podeu, però, utilitzar versions anteriors d'Squeak, la 3.6 per exemple, si voleu jugar amb el que s'explica en aquest capítol. Les podeu trobar a <http://ftp.squeak.org>



**Figura 25.1 —** Hi ha diversos entorns per jugar amb les característiques multimèdia d'Squeak.

Si feu clic a la finestra anomenada 3D, arribareu a l'entorn Squeak Alice que ve donat dins d'Squeak, com veieu a la figura 25.2. Hi ha quatre finestres a l'escriptori: la subfinestra superior dreta és una finestra que conté algunes anotacions que ens indiquen on trobar els objectes 3D i altra informació. La subfinestra superior esquerra, que conté un conill en 3D, és el món 3D en què els objectes 3D, anomenats actors, evolucionen. La subfinestra inferior dreta és l'editor d'*scripts* d'Squeak Alice. Aquesta és la finestra que farem servir per crear objectes 3D i definir *scripts* per controlar aquests objectes. L'editor d'*scripts* ja conté una llarga llista d'*scripts* interessants que us suggerirem de provar després. La subfinestra inferior esquerra mostra una llista jeràrquica de tots els actors que existeixen actualment al món 3D.

Abans que comenceu, us suggerim que comproveu la pantalla per veure quina profunditat de color hi ha definida al vostre entorn. La profunditat de color representa el nombre de colors que teniu disponibles. Per canviar-la, seleccioneu **appearance...** al menú *World* i després **set display depth...** Us recomanem que proveu l'opció 32, però el resultat pot dependre de les capacitats de les vostres targetes gràfiques. Quan la profunditat de color triada no és la de la targeta gràfica, Squeak ha de transformar sempre la renderització dels objectes 3D, de manera que Alice va més lent. Un cop heu triat la profunditat de color, us suggerim que feu una mica més gran la finestra esquerra amb l'opció de la nansa groga de l'halo. Fixeu-vos que la finestra completa és anomenada “*camera*” quan feu aparèixer l'halo; això és perquè aquesta finestra mostra el que una càmera observaria al món 3D.



**Figura 25.2 — L'entorn Squeak Alice tal com ve definit.**

## Interactuar directament amb els actors

Amb Squeak Alice podeu interactuar directament amb els objectes 3D, com el conill, que existeixen al món 3D:

- Per moure un actor horitzontalment, feu clic al món 3D. Això us permet portar el conill més lluny o més a prop dins del món.
- Per moure verticalment un actor, premeu Shift + Clic (veieu la figura 25.3, esquerra).
- Per girar verticalment un actor, premeu Ctrl + Clic (veieu la figura 25.3, mig).

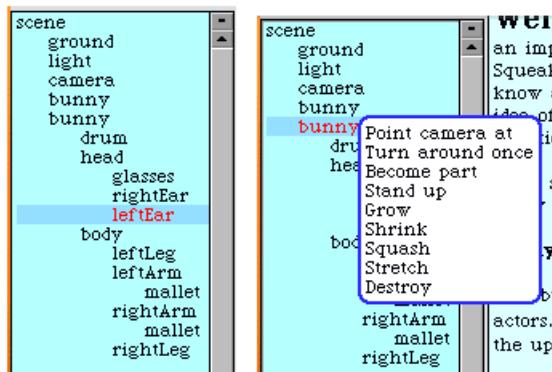
- Per girar lliurement un actor, premeu Shift + Ctrl + Clic (veieu la figura 25.3, dreta).



**Figura 25.3** — Podeu moure el conill amunt i avall (Shift + Clic), girar-lo verticalment (Ctrl + Clic), i girar-lo lliurement (Shift + Ctrl + Clic).

Fixeu-vos que si voleu que un actor torni a una posició estable, utilitzeu el mètode `standUp`. Això és molt útil per experimentar amb accions realitzades en paral·lel.

També podeu interactuar amb els actors via la llista dels actors disponible dins del món, com es veu a les figures 25.4 i també 25.2. La llista conté tots els actors. Podeu veure que la llum, la càmera i el terra són tots actors. Podeu aplicar als actors determinades accions ja definides, com fer-los créixer, fer-los més petits, estrènyer-los, tot seleccionant l'actor i obtenint el menú emergent (figura 25.4, dreta).



**Figura 25.4** — Una llista dels actors i la seva estructura jeràrquica.

Els actors es componen d'altres objectes. Les parts d'un actor estan estructurades jeràrquicament. Per exemple, el conill està compost d'un cap i un cos. El cap està compost d'ulleres i orelles. Les parts apareixen també a la llista, i també hi podeu realitzar accions a sobre. La figura 25.5 mostra el conill després de la transformació següent: hem fet més petit el tambor, hem fet més gran el cap i hem tallat l'orella esquerra.



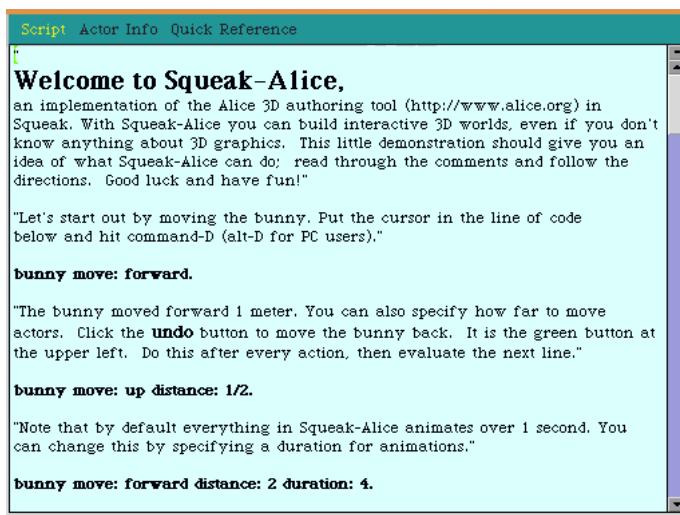
**Figura 25.5 — Un conill transformat.**

## L'entorn

Com ja hem mencionat, l'entorn està format principalment de tres components gràfics: l'escena, o finestra de càmera, que mostra el món 3D (finestra superior esquerra a la figura 25.2); la llista d'actors (figura 25.4); i l'editor d'*scripts* (l'àrea a la dreta de la llista d'actors a la figura 25.2). Ara discutirem l'editor d'*scripts* en detall. Hi ha tres botons al capdamunt: **Script**, **Actor Info** i **Quick Reference** (veure figura 25.6).

- El botó **Script** us permet editar i executar *scripts*
- El botó **Actor Info** mostra informació relacionada amb l'actor seleccionat a la llista d'actors (figura 25.7).
- El botó **Quick Reference** llista totes les accions possibles i constants definides per defecte per a cada tipus d'acció. Aquest és un ajut força útil i fàcilment accessible.

Quan el botó **Script** està seleccionat, podeu definir *scripts* i executar-los amb l'acció típica **do it** del menú, o bé amb Command + D o Alt + D. De fet, l'editor d'*scripts* és un *workspace*



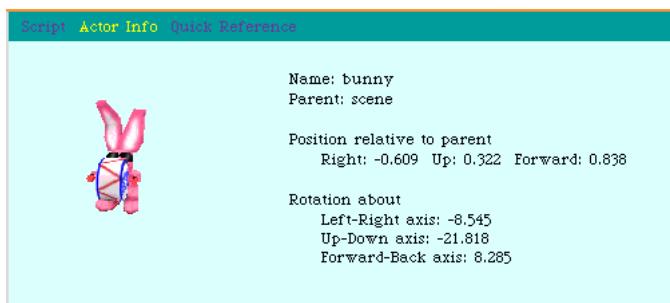
**Figura 25.6 — L'editor d'scripts.**

ampliat dedicat a l'execució d'Alice. Aquest *workspace* ampliat conté variables ja definides, tal i com s'explica a la **Quick Reference**. Per a la nostra exploració heu de saber només que camera es refereix a la càmera per defecte, *cameraWindow* al *morph* de l'escena, i *w* al *Wonderland*, és a dir, el sistema Alice complet.

Fixeu-vos que Squeak Alice s'executa sense acceleració del maquinari, que està desactivat per defecte. Com a tal, Squeak Alice s'executa en qualsevol plataforma. Si voleu activar l'acceleració del maquinari, feu aparèixer el menú (nansa vermella) sobre el *morph* de la càmera i seleccioneu l'opció amb l'original nom de **hardware acceleration**.

## Scripts

Abans de continuar, obriu els controls de la càmera, com mostrem a l'*Script 25.1*. D'aquesta manera, podreu seguir l'actor si és present dins l'angle de visió.

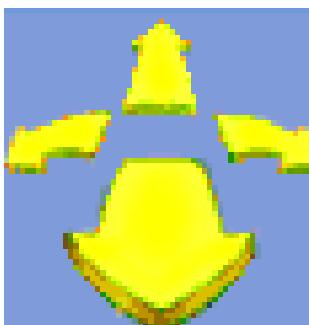


**Figura 25.7 — Informació dels actors.**

**Script 25.1** *Afegir els controls de la càmera.*

```
cameraWindow showCameraControls
```

Podeu moure la càmera fent clic sobre el giny mostrat a la figura 25.8. Fixeu-vos que podeu moure la càmera amunt i avall apretant la tecla Shift mentre moveu el ratolí a sobre del giny cameraControls. Si per accident premeu el botó reset i no voleu tornar a arrencar el sistema, podeu carregar el conill com expliquem a la secció “El Vostre Propi Wonderland” d'aquest capítol.



**Figura 25.8 — Afegint els controls de càmera amb cameraWindow showCameraControls.**

## Analitzar el primer *script*

Per poder compondre accions i canviar la seva velocitat d'execució, els autors d'Alice van alterar el model d'execució de missatges. El model d'execució dels actors és diferent del que hem vist fins ara a la resta del llibre. Fins i tot si la sintaxi és la mateixa, múltiples missatges enviats a un actor no s'executen en seqüència, sinó que s'executen combinats. Compareu les diferents execucions de l'*Script 25.2* que obtenuïu executant-lo línia per línia, o executant-lo tot de cop.

### **Script 25.2** Algunes accions senzilles.

```
bunny move: forward.  
bunny turn: left.  
bunny move: back.  
bunny roll: left.
```

Per executar un *script* format per una seqüència d'accions, utilitzeu el mètode `dolnOrder:`, com veieu a l'*Script 25.3*.

### **Script 25.3** Executar una seqüència de missatges un darrera l'altre.

```
w dolnOrder: {  
    bunny move: forward.  
    bunny turn: left.  
    bunny move: back.  
    bunny roll: left. }
```

Com veieu, la diferència entre tots els efectes succeint a l'hora i succeint en seqüència és força important. Un altre aspecte important és que l'entorn *Wonderland* proporciona algunes constants definides per programar actors, com per exemple `left`, `back` i `forward`, que han estat utilitzades dins l'*Script 25.3*. Aquí teniu una llista d'algunes de les constants disponibles pel moviment, tal i com es presenten a la subfinestra *Quick Reference*. No farem servir accions relacionades amb la localització en aquest capítol.

- direcció: `left`, `right`, `up`, `down`, `forward` i `back`.
- duració: `rightNow` i `eachFrame`
- estil: `gently`, `abruptly`, `beginGently` i `endGently`
- posició: `asls`
- localització: `onTopOf`, `below`, `beneath`, `inFrontOf`, `inBackOf`, `behind`, `toLeftOf`, `toRightOf`, `onFloorOf` i `onCeilingOf`

Aquestes constants s'utilitzen per especificar variacions dels mètodes per manipular actors. Llegiu la *Quick Reference* per veure les combinacions possibles.

## Moure, girar i rodar

Els actors poden ser manipulats per moure's, girar i rodar utilitzant els mètodes `move:`, `turn:` i `roll:`, com veieu a l'*Script 25.3*. La subfinestra *Quick Reference* mostra que aquests mètodes poden ser parametritzats per donar resultats diversos. Aquí teniu alguns exemples, però us suggerim que llegiu el capítol sobre Alice al llibre que hem esmentat al principi d'aquest capítol i la *Quick Reference* per aprendre totes les possibilitats. Fixeu-vos que hi ha algunes inconsistències entre la descripció i la implementació, per tant no dubteu a experimentar.

L'*Script 25.4* presenta la llista tal i com es presenta a la *Quick Reference*.

### **Script 25.4** Variacions sobre move:

```

move: aDirection
move: aDirection distance: aNumber
move: aDirection distance: aNumber
move: aDirection distance: aNumber duration: aNumber
move: aDirection distance: aNumber duration: aNumber style: aStyle
move: aDirection asSeenBy: anActor
move: aDirection distance: aNumber asSeenBy: anActor
move: aDirection distance: aNumber duration: aNumber asSeenBy: anActor
move: aDirection distance: aNumber duration: aNumber asSeenBy: anActor style: aStyle

move: aDirection speed: aNumber
move: aDirection speed: aNumber for: aNumber
move: aDirection speed: aNumber until: aBlock
move: aDirection speed: aNumber asSeenBy: anActor
move: aDirection speed: aNumber asSeenBy: anActor for: aNumber
move: aDirection speed: aNumber asSeenBy: anActor until: aBlock

```

Aquí teniu algunes explicacions: Primer, podeu especificar una *distància* utilitzant `distance:`. Després haurieu de saber que, per defecte, una animació triga un segon en executar-se. Per canviar aquest comportament per defecte podeu especificar una altra *duració* amb `duration:` i donar el nombre de segons que l'animació hauria de durar. De fet, fins i tot si definiu una duració de zero, pot no ser executat instantàniament pel *Wonderland*. Si realment voleu animacions instantànies, utilitzeu la constant `rightNow`. També podeu especificar un *estil*, utilitzant `style:` i les constants associades `gently`, `abruptly`, `beginGently` i `endGently` que descriuen com hauria de començar o acabar l'animació.

Les accions són normalment dependents del temps, la qual cosa vol dir que tenen un començament i un final. També podeu crear accions persistents, amb `speed:`, que especifica que els actors s'haurien de moure a una velocitat constant. Pareu atenció que si utilitzeu `speed:` però ometeu

la distància, l'actor es mourà per sempre. L'argument de velocitat pel mètode move: són metres per segon, mentre que pel mètode turn: és el nombre de girs per segon. L'argument especificat per for: us permet especificar una duració per al missatge quan utilitzeu velocitat; until: us permet especificar una condició, expressada mitjançant un bloc, que determinarà si l'acció continua. Fixeu-vos que podeu aturar l'animació amb el missatge stop.

Per defecte, accions com move: i turn: prenen com a referència l'actor mateix. Per tant, quan dieu bunny turn: left el conill girarà cap a la seva esquerra. De vegades voldreu especificar algun altre marc de referència, i en aquest cas hauríeu d'utilitzar asSeenBy: que us permet especificar un altre marc de referència, com veieu als exemples de l'*Script 25.5*.

#### **Script 25.5** Exemples de variacions sobre missatges.

```
bunny move: forward distance: 3 duration: rightNow style: endGently
bunny move: forward distance: 3 duration: 0
bunny move: forward distance: 3 duration: rightNow
bunny move: forward distance: 5 speed: 1
bunny move: left distance: 3 duration: 3 asSeenBy: camera
bunny turn: left turns: 3 speed: 1
bunny roll: right turns: 2
```

## Parts dels actors

Els actors estan formats per parts en una relació jeràrquica pare-fill. Les parts pertanyen només a un sol progenitor. Aquesta relació és important ja que les accions enviades a un actor afecten les seves parts. Per exemple, quan demaneu al conill que mogui el cap, les ulleres, que formen part del cap, també es mouen. Les parts no són res d'especial. Senzillament són actors als quals podeu enviar missatges, com il·lustrem a l'*Script 25.6*.

#### **Script 25.6** Enviar missatges a parts.

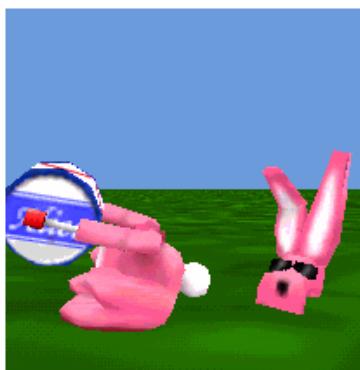
```
bunny drum roll: left bunny drum roll: left speed: 1

w doInOrder: {
    bunny head glasses move: forward.
    bunny head glasses move: back }

bunny drum stop
```

De fet, tots els actors són parts d'un super-progenitor anomenat l'*escena* (*scene*). Si observeu la llista jeràrquica que mostra tots els actors del Món, veieu l'escena, i a sota, amb sagnat, el conill, però també el terra, el llum i la càmera.

De vegades necessiteu enviar un missatge a un objecte afectant només algunes de les seves parts. Per exemple, voleu canviar el color del conill sense canviar el color de la seva orella esquerra; així doncs, heu de fer l'orella independent de la resta del conill. Per fer això, Alice introduceix la noció d'objecte de *primera classe*. Una part de *primera classe* pertany al progenitor però no es veu afectada per canvis al progenitor. A la figura 25.9 podeu veure com hem separat el cap del conill de la resta.



**Figura 25.9 —** Un cos independent per fer dibuixos animats.

Dos mètodes us permeten controlar si un objecte és part d'un altre. El mètode `becomeFirstClass` fa que el receptor es torni un objecte de primera classe, mentre el mètode `becomePart` fa que el receptor sigui part del seu progenitor. Executeu l'*Script 25.7* línia a línia per entendre la diferència.

**Script 25.7** Exemples de parts de primera classe.

```
bunny setColor: green.  
bunny head becomeFirstClass.  
bunny setColor: red.  
bunny head becomePart.  
bunny setColor: pink
```

També podeu canviar la relació progenitor-fill entre objectes utilitzant el mètode `becomeChildOf` (veieu l'*Script 25.8*).

**Script 25.8** *Enviar missatges a parts.*

```
bunny head becomeChildOf: ground
bunny move: forward
ground head turn: left
```

## Altres operacions

Els actors entenen molts més missatges del que us hem ensenyat fins ara. Aquí teniu una descripció senzilla dels altres mètodes.

### Fer-se gran

El mètode `resize:` canvia la mida del receptor. Existeix en moltes variacions, com `resize:duration:`, `resizeTopToBottom:leftToRight:frontToBack:`, i `resizeLikeRubber:dimension:`, com veieu a l'*Script 25.9*.

**Script 25.9** *Experiments canviant la mida.*

```
bunny resize: 1/2
bunny resizeTopToBottom: 2 leftToRight: 1 frontToBack: 3
bunny resizeLikeRubber: 2 dimension: topToBottom
```

### Moviments quantificats

El mètode `nudge:` mou un actor en múltiples de la seva longitud, amplada o alçada en funció de la direcció triada.

**Script 25.10** *Experiments amb nudge:.*

```
bunny nudge: up distance: 2 duration: 2
```

### Mantenir-se dret

Els mètodes `standUp` i `standUpWithDuration: unNombre` permeten que un actor es mogui a una posició vertical, la qual cosa pot ser útil després de fer experiments.

### Acolorir

El mètode `setColor:` canvia el color del receptor. Existeix de maneres molt variades, com `setColor:duration:` i `setColor:duration:style:`. Fixeu-vos en l'*Script 25.7* per veure un exemple.

## Destrucció

El mètode `destroy` destrueix un actor amb una animació divertida. Afortunadament, l'entorn *Wonderland* té un mecanisme potent per desfer (*undo*) accions.

## Visibilitat

Els mètodes `hide` i `show` gestionen la visibilitat d'un objecte.

## Moviments absoluts i rotacions

Fins ara hem utilitzat només accions que canviem la posició o la direcció d'un actor. El mètode `moveTo:` mou el receptor a un lloc donat, i el mètode `turnTo:` fa que el receptor apunti a una posició determinada. La posició i la direcció poden ser un triplet de la forma `{ dreta . amunt . endavant }` o bé `unActor`. Els valors del triplet poden ser nombres o bé `asls` (per exemple, `{ asls . 0 . asls }`). Fixeu-vos que el triplet descriu una posició al marc de referència del progenitor de l'actor. Per tant `bunny moveTo: { 1 . 1 . 0 }` significa que el conill s'hauria de moure 1 metre a la dreita i 1 metre per sobre de l'origen de l'escena, que és el progenitor de l'actor conill. El mateix triplet a l'expressió `bunny head moveTo: { 1 . 1 . 0 }` es refereix a la posició 1 metre a la dreita i 1 metre per sobre de la posició origen del conill.

Compareu les accions de `moveTo:` i de `move:` a l'*Script 25.11*.

### **Script 25.11** Experiments amb moviments absoluts

```
bunny moveTo: {0 . 0 . 0}.
bunny head moveTo: {0 . 1 . 0}.
bunny head moveTo: {0 . -1 . 0}
bunny head move: up.
bunny head move: down
bunny head turnTo: camera duration: 1 style: abruptly
bunny turnTo: camera duration: 1 style: abruptly
```

Fixeu-vos que el mètode `alignWith: unActor` és equivalent a `turnTo:`

## Apuntar

El mètode `pointAt: unObjectiu` permet que els actors es vegin entre ells. Un objectiu pot ser un actor, un moviment `{ dreta . amunt . endavant }` o una posició de píxel `x@y`.

**Script 25.12** *Experiments amb pointAt:.*

```
bunny pointAt: camera.
bunny turn: left.
bunny move: forward.
camera pointAt: bunny.
```

**Posicionament relatiu dels actors.**

Finalment, els actors poden ser col·locats en posicions relatives respecte d'altres actors, utilitzant el mètode place: unLloc object: unActor. Les posicions s'especifiquen amb les constants onTopOf, below, beneath, inFrontOf, inBackOf, behind, toLeftOf, toRightOf, onFloorOf i onCeilingOf. Proveu de carregar molts actors, tal i com explicarem més tard, i jugueu amb les expressions de l'*Script 25.13*.

**Script 25.13** *Col·locar els actors.*

```
camera place: inFrontOf object: bunny.
camera move: up.
bunny move: back distance: 2.
camera pointAt: bunny.
bunny head place: toRightOf object: bunny
```

**Accions relacionades amb el temps**

També podeu definir accions que es relacionen amb el pas del temps utilitzant la constant eachFrame com a argument de duration:, com podeu veure a l'*Script 25.14*. Utilitzeu el mètode stop per aturar l'animació (veieu la propera secció).

**Script 25.14** *Utilitzar la constant eachFrame.*

```
bunny head pointAt: camera duration: eachFrame.
bunny move: forward
```

També podeu especificar la durada d'una acció en segons quan l'acció utilitza l'opció eachFrameFor: unNombre, com en l'*Script 25.15*.

**Script 25.15** *Restringir una acció una determinada durada.*

```
bunny moveTo: {asls . 0 . asls} eachFrameFor: 10
```

El mètode `eachFrameFor: unNombre` fa que les accions es repeteixin un cert nombre de segons. El mètode `eachFrameUntil: unBloc` repeteix les accions fins que el bloc retorna `true`.

Fixeu-vos que `asls` és una constant especial que manifesta que el mètode executat no modifícarà el valor actual. Deixa el valor “tal com està” (“*as is*”). Altres mètodes, però, poden canviar aquest valor. Aquí l’*script* restringeix el conill a seguir el terreny. Fixeu-vos també que el triplet significa `{ dreta . amunt . endavant }` i per tant el conill no es pot moure amunt o avall per un període de 10 segons.

## Animació

Per definir una animació, només cal assignar-la a una variable. Per exemple, declarem a l’*Script 25.16* que `spin` fa girar dues vegades el conill cap a l’esquerra durant 2 segons.

**Script 25.16** *Una animació senzilla.*

```
spin := bunny turn: left turns: 2 duration: 2.
```

Podeu aturar temporalment l’animació (`spin pause`), tornar a posar-la en marxa (`spin resume`), aturar-la del tot (`spin stop`) o començar-la altre cop (`spin start`).

Les animacions també poden repetir-se utilitzant els mètodes `loop` i `loop: unNombreDeRepeticions`, o aturar les repetitions amb `stopLooping`.

**Script 25.17** *Una animació senzilla*

```
flip := bunny turn: forward turns: 1 duration: 2.
```

Ara podeu compondre animacions amb el mètode `doInOrder:`, que executa una seqüència de missatges en seqüència, o `doTogether:`, que executa una seqüència de missatges combinant-los.

**Script 25.18** *Dues animacions senzilles executades en seqüència.*

```
w doInOrder: {spin start . flip loop:2}
```

**Script 25.19** *Dues animacions senzilles compostes.*

```
w doTogether: {spin start . flip loop:2}
```

Fixeu-vos que hi ha combinacions que no funcionen i us podeu trobar amb un *walkback* (una finestra que us notifica que hi ha un problema). Una composició pot rebre un nom i formar part d’altres animacions.

**Script 25.20** *Dues animacions senzilles compostes en seqüència.*

```
bla := w doInOrder: {spin start . flip loop:2}. bla start
```

Recordeu que el mètode `standUp` fa que l'actor romangui vertical de peu, la qual cosa és útil després de resultats inesperats en compondre accions.

## El vostre propi *Wonderland*

Podeu crear el vostre propi *Wonderland* executant l'*Script 25.21*.

**Script 25.21** *Obrir un nou Wonderland.*

```
Wonderland new
```

Un cop hagieu creat el vostre propi *Wonderland*, o després de prémer el botó de `reset`, hauríeu de carregar alguns objectes 3D. L'equip que desenvolupa Alice proporciona un fitxer ZIP ple de personatges 3D a <http://www.cs.cmu.edu/~jpierce/squeak/SqueakObjects.zip> (o busqueu-los a <http://www.apress.com>). Aquests objectes es representen en un format antic, i per tant provar de carregar els nous objectes Alice no funcionarà a Squeak.

Podeu crear un avió senzill, com a simple experiment, amb l'expressió `w makePlaneNamed: 'aviaMeu'`. Aquí teniu com carregar objectes 3D en un PC (*Script 25.22*) o en Mac OS X (*Script 25.23*).

**Script 25.22** *Carregar nous objectes 3D en un PC.*

```
w makeActorFrom: 'Objects\Animals\Bunny.mdl'
```

**Script 25.23** *Carregar nous objectes 3D en Mac OS X.*

```
w makeActorFrom: ':Objects:Animals:PurpleDinosaur.mdl'
```

“si teniu problemes al mac utilitzeu el nom complet”

```
w makeActorFrom: 'OSX:Users:ducasse:Alice:Objects:Animals:PurpleDinosaur.mdl'
```

Després de carregar diversos personatges, com el ninot de nou o el dinosaure porpra (figura 25.10), podeu escriure i executar els *Scripts 25.24* i *25.25*.



**Figura 25.10 — Els tres amics.**

**Script 25.24** Script amb diversos actors.

```
bunny turn: left
bunny turn: left asSeenBy: snowman
snowman place: inFrontOf object: purpleDinosaur.
```

**Script 25.25** Una mirada ràpida al conill.

```
w doInOrder: {
    purpleDinosaur head pointAt: bunny
    purpleDinosaur head alignWith: purpleDinosaur }
```

## Múltiples càmeres i altres efectes especials

Tenir diverses càmeres pot fer que la renderització 3D d'Alice sigui més lenta, però val la pena entendre com funciona per construir animacions. Quan es crea una nova càmera, també es crea una nova visualització. Igual que un objecte 3D nou que representa la càmera. Canviar la localització de la càmera fent clic a sobre automàticament canvia la visualització que mostra allò que veu la càmera. A més, canviar la posició de la càmera utilitzant el giny dels controls de la càmera modifica la posició de l'objecte càmera.

A la figura 25.11, hi ha tres càmeres. Hem triat la càmera de la dreta per tenir una visió panoràmica de l'escena, mentre que les dues càmeres de l'esquerra s'han situat per proporcionar diferents plans propers del conill. L'*Script 25.26* mostra com afegir una altra finestra de càmera.

**Script 25.26** *Crear una altra finestra de càmera.*

```
w makeCamera
```

Fixeu-vos que una càmera és un actor com qualsevol altre objecte 3D, i per tant el podeu moure utilitzant els missatges mostrats a l'*Script 25.27*.

**Script 25.27** *Moure una càmera.*

```
w doInOrder: {
    camera roll: left.
    camera move: back distance: 4.
    camera standUp.}
```

## Alarms

Cada *Wonderland* manté un seguiment del temps gràcies a un objecte planificador de tasques (*scheduler*). Quan es crea un *Wonderland*, es posa el comptador de temps a zero, i el planificador comença a actualitzar aquest comptador de temps a cada marc. Podeu obtenir el temps actual utilitzant l'expressió *scheduler getTime*. El que és interessant és que podeu programar una *alarma* per executar certes accions en un moment determinat utilitzant el mètode *do:at:inScheduler:*, o, un cop ha passat un determinat període de temps, utilitzant el mètode *do:in:inScheduler:*. L'*Script 25.28* defineix dues alarmes.

**Script 25.28** *Dues alarmes.*

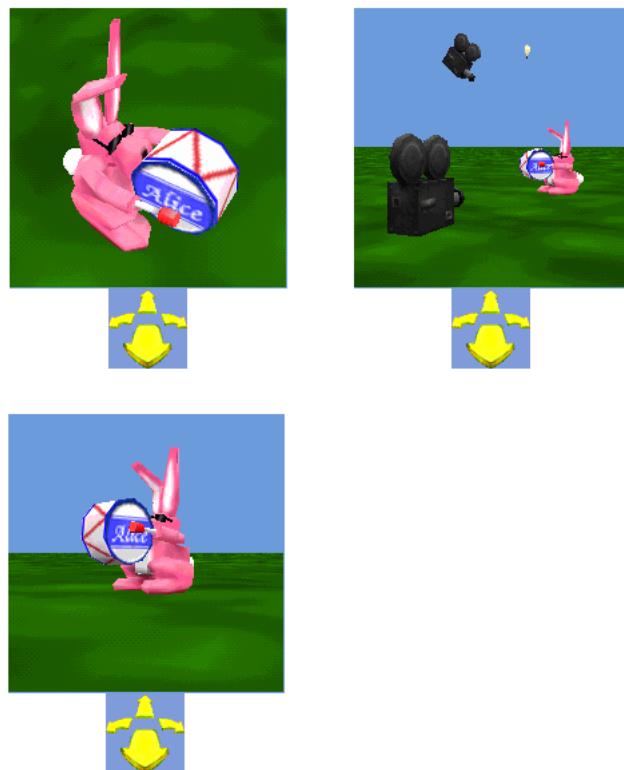
Alarm

```
do:
    [ bunny head turn: left turns: 3.
      bunny setColor: red ]
  at: (scheduler getTime + 5)
  inScheduler: scheduler.
```

Alarm

```
do: [bunny setColor: pink]
in: 8
inScheduler: scheduler.
```

Podeu enviar els següents missatges a una alarma: *checkTime*, que retorna el temps en què l'alarma s'hauria d'executar, i *stop*, que aturará l'alarma si encara no s'ha executat del tot.



**Figura 25.11** — Les finestres de l'esquerra mostren el que les càmeres de la finestra de la dreta veuen.

## Introduir la interacció amb l'usuari

En aquest moment podeu programar animacions, però encara no podeu definir interaccions amb l'usuari. Squeak Alice us permet vincular accions als actors quan succeeixen determinats esdeveniments, com els clics del ratolí. Per exemple, l'*Script 25.29* fa que el conill giri el cap quan fem clic al damunt amb el botó dret del ratolí.

**Script 25.29** *Definir una acció associada amb un clic del botó dret del ratolí.*

```
bunny respondWith: [:event | bunny head turn: left turns: 1] to: rightMouseClick
```

Els tres mètodes addResponse: unBloc to: tipusEsdeveniment, removeResponse: unBloc to: tipusEsdeveniment i respondWith: unBloc to: tipusEsdeveniment gestionen la definició de les accions. Les accions s'expressen utilitzant blocs i s'associen amb tipus d'esdeveniments, entre els quals podem trobar els següents: keyPress, leftMouseDown, leftMouseUp, leftMouseClick, rightMouseDown, rightMouseUp i rightMouseClick.

La diferència entre els mètodes addResponse:to: i respondWith:to: és que el primer us permet definir diverses accions associades al mateix tipus d'esdeveniment, mentre que el segon elimina les accions prèviament definides i en defineix una de nova. El mètode removeResponse:to: elimina les accions corresponents. Un exemple utilitzant removeResponse:to: apareix a l'*Script 25.30*.

**Script 25.30** *Definir una acció associada amb un clic del botó esquerre del ratolí.*

```
bunny
respondWith:
  [:event |
  bunny head turn: left turns: 2 duration: 2.
  w doInOrder: {
    bunny head move: up.
    bunny head move: down}] to: leftMouseClick
```

A l'*Script 25.31* hem afegit dues reaccions i després hem eliminat la primera de manera que només s'executa la segona quan feu clic a sobre del conill.

**Script 25.31** *Gestionar les respostes.*

```
reaction := bunny
  addResponse: [ :event | bunny head turn: left turns: 1 ]
  to: rightMouseClick.
bunny
  addResponse: [ :event | bunny head pointTo: { 0 . 0 . 0 } ]
  to: rightMouseClick.
bunny removeResponse: reaction to: rightMouseClick
```

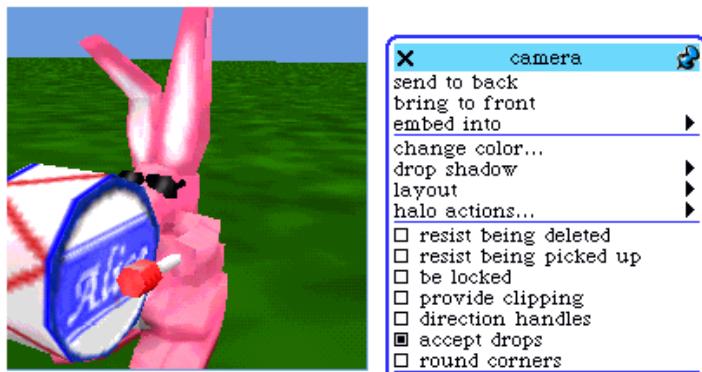
## Aspectes ocults d'Alice i Pooh

Ens agradaria acabar aquesta presentació d'Squeak Alice mostrant-vos alguns aspectes divertits que il·lustren també la potència d'Alice.

## Projectar *morphs* 2D en 3D

Podeu posar *morphs* 2D en objectes 3D. El procés és el següent:

- Primer feu aparèixer la nansa vermella **menu** a la finestra de la càmera i seleccioneu l'opció **accept drops**, com veieu a la figura 25.12.



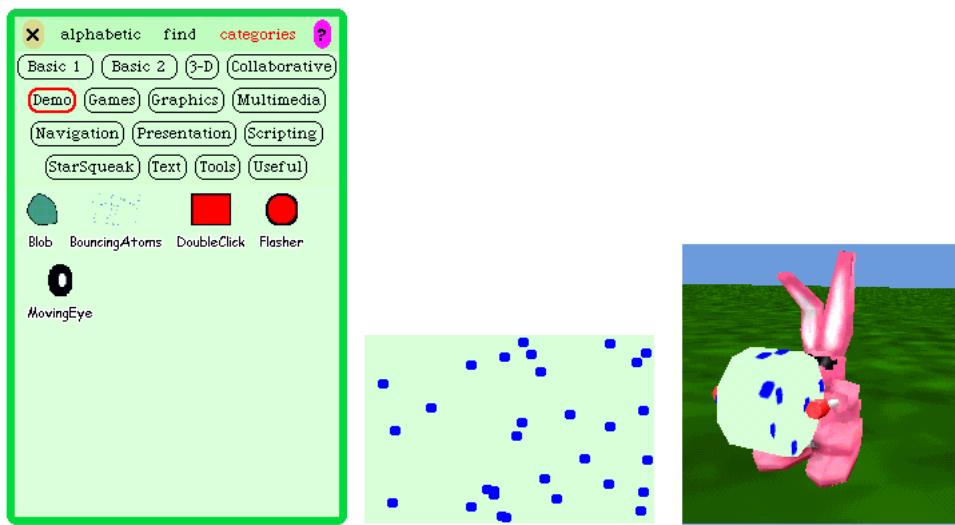
**Figura 25.12 —** Esquerra: *El conill abans*. Dreta: *Diem a la càmera que accepti objectes deixats anar sobre el conill*

- Després creeu un *morph* actiu; per exemple, feu aparèixer la finestra dels objectes utilitzant l'opció **objects (o)**, o Command + Q, del menú principal d'Squeak i creeu un *morph* d'àtoms que reboten com veieu a la figura 25.13.
- Deixeu anar el *morph* que acabeu de crear sobre alguna part del conill. Usualment, el *morph* s'hauria de projectar a l'objecte 3D, com veieu a la figura 25.13 (dreta).

Finalment, l'*Script* 25.32 mostra un aspecte divertit i potent d'Alice i Squeak. Crea un mur a través del que podeu veure el que està apuntant el ratolí.

### Script 25.32 Diversió amb Alice.

```
w makePlaneNamed: 'test'.
test
doEachFrame:
[ test setTexturePointer:
  (Form fromDisplay: ((Sensor mousePoint) extent: 50@50)) asTexture ]
```

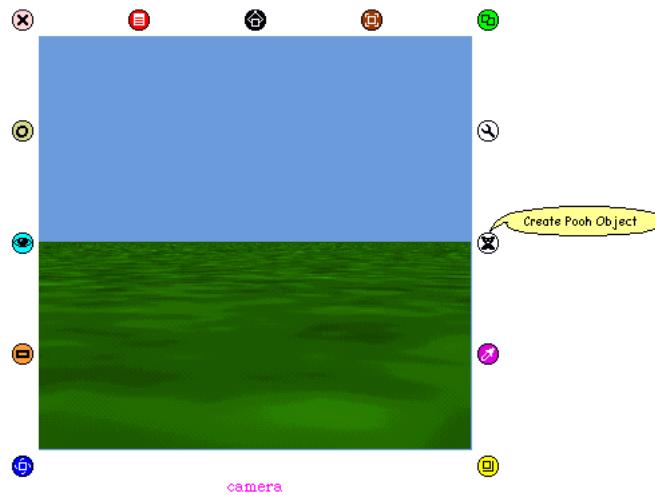


**Figura 25.13 —** Esquerra: La finestra dels objectes. Mig: Un morph d'àtoms que reboten. Dreta: Un morph actiu projectat sobre un objecte 3D.

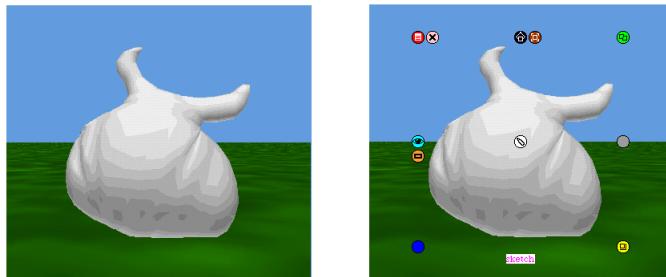
## Pooh: generar formes 3D a partir de 2D

Pooh és un sistema que us permet generar formes 3D dibuixant formes 2D dins d'un món Alice. Si teniu curiositat, proveu de fer el següent:

- Obriu un nou món *Wonderland* (*Wonderland new*).
- Feu aparèixer l'halo de la finestra de la càmera, com veieu a la figura 25.14
- Seleccioneu l'halo blanc del mig a la dreta amb la icona petita de l'ós.
- Dibuixe una corba tancada directament a la finestra de la càmera. Quan acabeu, Pooh genera una forma 3D, com podeu veure a la figura 25.15 (dreta)
- Ara podeu dibuixar dins la forma obtenint l'halo de la nova forma i triant la nansa del llapis que teniu al centre, com podeu veure a la figura 25.15. Això obre un editor de dibuixos. Quan acabeu, premeu el botó *Keep* de l'editor de dibuixos. La figura 25.16 mostra la forma anterior dibuixada. També mostra que podeu girar aquesta forma, i qualsevol altre.

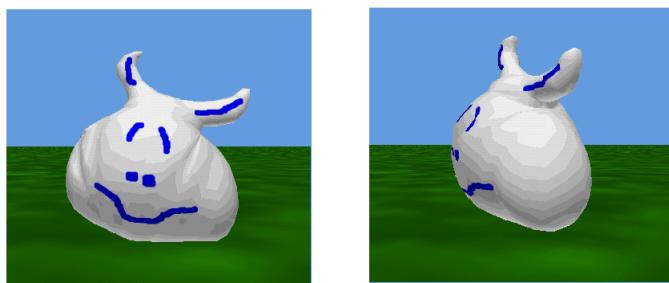


**Figura 25.14** — Obrir l'halo de la càmera d'Alice per obtenir accés a Pooh

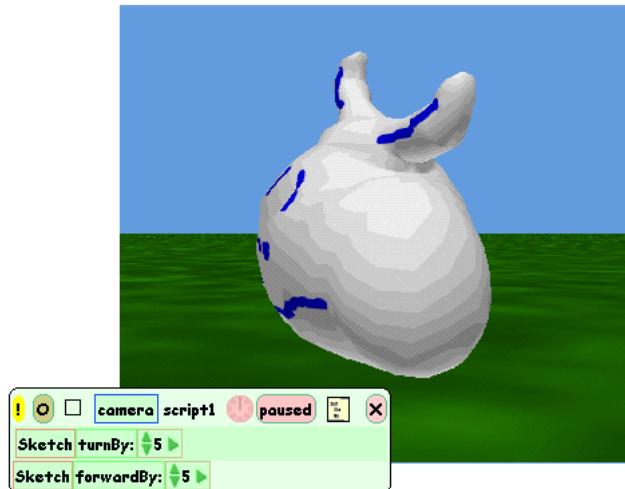


**Figura 25.15** — Esquerra: Obtenir una forma 3D. Dreta: Obtenir l'editor de dibuixos sobre una forma 3D.

Finalment ens agradaria mostrar-vos alguns aspectes experimentals d'Squeak. Podeu utilitzar eToy, presentat en el capítol anterior, amb els objectes 3D d'Alice. Podeu obtenir un visualitzador amb la nansa de l'ull de l'objecte 3D utilitzant les mateixes tècniques que vam explicar al capítol 24. La figura 25.17 mostra un senzill *script*.



**Figura 25.16** — Esquerra: *Una vaca dibuixada.* Dreta: *La mateixa vaca girada.*



**Figura 25.17** — Utilitzar un script eToy per controlar un objecte 3D.

## Resum

Alice és un entorn molt potent. Us hem mostrat només uns quants dels aspectes més importants. El lector interessat hauria de llegir el capítol dedicat a Alice dins del llibre sobre Squeak mencionat al començament d'aquest capítol.

# Índex alfabètic

- ' (cometes simples), utilitzades en cadenes de caràcters, 229
  - " (cometes), utilitzades en comentaris, 167
  - () (parèntesis)
    - errors provocats per, 274–278
    - inclosos per alterar l'ordre d'execució, 146
    - utilitzats amb punts, 283
  - . (punt)
    - oblidar un, 31
    - utilitzar en missatges i *scripts*, 24
  - // (divisió), triar per conduir el volant d'eToy, 349
  - : (dos punts)
    - utilitzar en mètodes i múltiples arguments, 196
    - utilitzar en mètodes i paràmetres, 189
    - utilitzar en paràmetres, 192
    - utilitzar en selectors de missatge, 184
  - := (expressió d'assignació)
    - parts dreta i esquerra, 119–120
    - utilitzar en traces, 237
    - utilitzar en variables, 107–108
    - utilitzar per fer traces, 235
  - @ mètode, efecte del, 284
  - [] (claudàtors)
    - utilització de, 267
    - utilitzar en blocs condicionals, 246
    - utilitzar en repeticions, 94, 265
  - & (conjunció) missatge
    - error de parèntesi relacionat amb, 277
    - exemples, 271–274, 278
  - ^, retornar valors amb, 169
  - | (disjunció) missatge
    - error de parèntesi relacionat amb, 277
  - examples, 271–274
  - || (barres verticals) posar variables entre, 107
  - 20 + (2 \* 5) expressió, descomposició, 153
  - 20 + 2 \* 5 expressió, descomposició, 152
  - 30 graus, girar, 47
  - 45 graus, girar, 51
  - (50@60) + (200@400), descompondre, 284
  - 50@60 + 200@400, descompondre, 284
  - 65@325 extent: 134@100 missatge, descompondre, 150
- A
- dibuixar, 41–42
  - forma de, 104
  - utilitzar moviments absoluts amb, 293–294
  - utilitzar variables, 108–109
  - variacions, 104–106
- Abandon, botó; a la finestra del depurador, efecte, 203
- absolut
- moviment, versus moviment relatiu, 285–288
- absoluta
- orientació, versus orientació relativa, 48–50
- abstracció, construint sobre els detalls de la definició, 178
- accés (mètodes de), mostrar per les categories en eToy, 333
- accent circumflex (^), retornar valors amb, 169
- accions
- afegir per a un cotxe a eToy, 352
  - component a Alice, 364
  - crear a Alice, 365
  - definir a Alice, 375

eliminar a Alice, 376  
 accions BotsInc... submenú, descripció de, 61  
 accions relacionades amb el temps, definir a Alice, 370–371  
 aconsegueixImatgeDeClasse mètode, descripció i exemple, 88  
 Actor Info (botó) a l'editor d'*scripts* d'Alice, 361  
 actors a Alice  
     estructura jeràrquica, 359  
     interactuar directament amb, 359–361  
     mostrar informació, 361  
     moure, girar i rodar, 365–366  
     parts de, 366–367  
     posicionament relatiu, 370  
 addResponse:to: i respondWith:to: mètodes a Alice, comparar, 375  
 agulles de rellotge, moure, 52–53  
 alarma, programar dins un *Wonderland*, 374  
 algunes capses, Experiment, 179  
 Alice, entorn de creació personalitzada, *Vegeu també Wonderland*  
     accedir, 357  
     accions relacionades amb el temps, 370–371  
     aconseguint el temps per *Wonderland*, 374  
     aturar animacions, 365  
     canviar de mida els receptors, 368  
     canviar la relació progenitor-fill dels objectes, 367  
     compondre accions, 364  
     components, 361–362  
     crear *Wonderland*, 372–373  
     definir animacions, 371–372  
     editor d'*scripts*, 361  
     efectes especials, 373–374  
     eliminar accions, 375–376  
     especificant la durada d'una acció, 370  
     executar missatges, 364  
     interacció amb l'usuari, 375–376  
     interactuant amb els actors, 359–361  
     múltiples càmeres, 373–374  
     mostrar per pantalla, 358  
     moure, girar i rodar, 365–366

objectes de primera classe, 367  
 projectar *morphs* 2D en 3D, 377  
 propietats de, 357  
 utilització d'*scripts*, 362–366  
 utilització d'eToy amb, 378  
 utilització del mètode *destroy*, 369  
 utilització del mètode *nudge*, 368  
 utilització del mètode *place*, 370  
 utilització del mètode *pointAt*, 369  
 utilització del mètode *setColor*, 368  
 utilització dels mètodes *hide* i *show*, 369  
 utilització dels mètodes *moveTo*: *i move*, 369  
 utilització dels mètodes *standUp*, 368  
 variacions de missatges, 367  
 una altra escala estranya, Experiment, 131  
 una altra espiral, Experiment, 216  
 altura, variable; relació amb l'amplada i la mitja altura, 104  
 amplada, variable; relació amb l'altura i la mitja altura, 104  
 anglePerApuntarA: mètode  
     definició per simular comportament animal, 325  
     descripció i exemples de, 305  
     diagrama de i codi de, 304  
 angles  
     aleatoris, per simular comportament animal, 309  
     canviar els valors dels, 53  
     girar, 50–54  
     per missatges referents a direccions absolutes, 300  
     representar, 50  
     versus temps, 54  
     vincular amb la posició del volant a eToy, 348–350  
 animacions  
     aturar, dins d'Alice, 365  
     crear contenidors per a, 341  
     crear dibujos senzills com a base de, 343  
     crear parts independents per a, 366  
     definir dins d'Alice, 371–372

- aparenta\* mètodes, descripció i exemple, 88  
 aparentaBot missatge, efecte de, 77  
 aparentaCercle missatge, aplicat als robots, 77, 82  
 aparentaImatge missatge, efecte de, 85  
 aparentaTriangle missatge, efecte de, 77–79  
 aplicació (Squeak), fitxer de, resoldre problemes, 12  
 apuntaA: mètode  
     utilitzar en simulacions de comportament animal, 322  
     codi de, 303–304  
 Apuntar i Seleccionar amb el ratolí, 63  
 aranya.frm fitxer de gràfic, contingut de, 83  
 area: mètode, descripció i exemple, 88  
 area: missatge, canviar la mida del robot amb, 78  
 arguments  
     en missatges de paraula-clau, 145  
     i paràmetres, 193–196  
     indicar pels receptors dels missatges, 142  
     per a mètodes, 189  
     utilitzar en mètodes, 184  
     variables com a, 195  
 art abstracte, Experiment, 40  
 asIs constant, utilitzar a Alice, 371  
 aspecte... submenú, descripció de, 61  
 assignació, introduir a eToy, 345  
 asString missatge, exemple de, 233  
 aviò  
     afegir mètodes, 337  
     arrosgregar i deixar un mètode per crear nous *scripts*, 335  
     crear *joystick* per, 339–340  
     dibuixar, 330  
     experimentar amb el *joystick*, 339  
     fer una icona, 355  
     jugar amb l'halo, 330–337  
     volar, 337  
 bacteris, exemple  
     canviar direcció, 316  
     incrementar velocitat, 316  
 bafarada, mostrant missatges mitjançant els, 7  
 barres verticals (| |) posar variables entre, 107  
 basic (botó), prémer a eToy, 344  
 binaris, missatges, *Vegeu també* missatges  
     exemples de, 139–141, 144  
     explicació de, 139–141  
     prioritat, 152  
 bloc condicional, exemple de, 246  
 blocs, sagnat de, 95  
 Bot nou patro4, execució de, 200  
 botó central del ratolí, conjunt d'operacions associades amb, 63  
 botó dret del ratolí, conjunt d'operacions associades amb, 63  
 botó esquerre del ratolí, conjunt d'operacions associades amb, 63  
 botons del ratolí  
     explicació, 63–65  
     i combinacions de tecles, 63  
 botons del ratolí, operacions associades als, 63  
 brúixola, apuntar els robots en les direccions principals de la, 38  
 bucle controlat per les dades d'entrada  
     proporcionades per l'usuari, afegir a una escala interactiva, 266  
 bucles, *Vegeu també* bucles imbricats  
     aturar, 263–264  
     combinar variables amb, 131–134  
     definir, 264  
     experimentar amb, 99–101  
     introduir variables als, 134  
     repetir seqüències de missatges en, 93–95  
     i translacions, 295–296  
     utilitzar en un *script* per dibuixar una piràmide, 97  
 bucles condicionals, *Vegeu* repeticions condicionals  
 bucles imbricats, exemples de, *Vegeu també* bucles, 225–227  
 bucles infinits, aturar, 263–264  
 cadenes de caràcters  
     i caràcters, 231–233  
     i nombres, 233, 235  
     panorama general, 229

- camins dibuixats per robots, eliminar a eToy, 355  
canviar de direcció, 38–40  
canviar la direcció de referència, Experiment, 54  
capses, *Vegeu també* comportament d'un animal atrapat dins d'una capsula  
afegint una sortida a, 315  
moure robots dins de, 302–303  
captura i guarda imatge, opció del menú, accedir, 67  
capturar pantalla (opció menú), accedir, 66–67  
carpetes, explorar, 65  
carregaImatge: missatge  
descripció i exemple, 88  
efecte de, 83–87  
carretera  
acolorir a eToy, 351  
crear a eToy, 350  
cas d'estudi del missatge d'error provocat pel missatge not, 275  
casa, dibuixar, 56  
cascada, *Vegeu també* missatges  
enviar múltiples missatges amb, 15  
usos de, 25  
categories  
crear pels mètodes, 163–164  
mostrar en el visualitzador d'eToy, 333  
center (missatge), descripció i exemple, 298  
center (missatge), descripció i exemples, 305  
centre versus posició, 305  
circular, forma; aplicar als robots, 77  
classe Bot  
associar gràfics amb, 84–87  
carregar i guardar gràfics associats amb, 83  
crear el robot aranya amb, 81–83  
significat de, 26  
classes, *Vegeu també* fàbriques de robots  
com a fàbriques, 35  
crear objectes, 35  
inicialitzar gràfics per defecte, 84  
obtenir nous objectes a partir de, 26  
paper dins la programació orientada a objectes,  
20  
claudàtors ([])
- utilització de, 267  
utilitzar en blocs condicionals, 246  
utilitzar en repeticions, 94, 265  
clear all pen trails mètode, utilitzar a eToy, 355  
codi, sagnat de, 95  
collection categoria, buscar, 344  
col·locació de l'increment dins del bucle,  
Experiment, 131  
color (objectes), obtenir, 74  
color (tests), personalitzar, 351  
Color classe, efecte de, 27  
color del text, canvis del, 31  
Color r:g:b: expressió, crear colors amb, 76  
color sees mètode, triar color en l'exemple del cotxe,  
351  
Color, fàbrica, propietats de, 76  
color, missatges  
efecte de, 9  
executar, 70  
noms de, 76  
color: mètode, descripció i exemple, 88  
colorLlapis: mètode, descripció i exemple, 88  
colorLlapis: missatge, efecte de, 74–75  
colors  
canviar-los en els robots, 37, 43, 243–246,  
248–250  
canviar-los en quadrícules, 282  
crear, 76  
dels llàpisos, 74–76  
demantar a un robot, 70  
relació amb la direcció del robot, 250–252  
triar-los amb el mètode fromUser, 76  
comentaris, incloure dins dels mètodes, 167  
cometes ("'), utilitzades en comentaris, 167  
cometes simples ('), utilitzades en cadenes de  
caràcters, 229  
compondre mètodes, definició de, *Vegeu també*  
mètodes, 176  
compondre solucions, significat de, 213  
comportament d'un animal atrapat dins d'una  
capsa, *Vegeu també* capses  
afegir una sortida a la capsula, 315

- direcció aleatòria, 314
- panorama general, 312–313
- resseguir els costats, 313–314
- volar al costat oposat, 314
- compostes, expressions booleanes, utilitzar, 271–274
- compostes, expressions, descomponent considerant els parèntesis, 150
- condicionals mètodes, triar, 248
- conill
  - moure a Alice, 359
  - script* de mostra, 373
  - transformar dins Alice, 361
- conjunció (&) missatge
  - error de parèntesi relacionat amb, 277
  - exemples, 271–274, 278
- constants pel moviment a Alice, exemples de, 364
- contenidor, elements de; canviar a eToy, 345–346
- contenidors
  - crear per a animacions, 341
  - deixar els dibuixos dins del, 343
  - obrir dins del visualitzador, 344
  - variables com a, 118–119
- contenidors de dibuixos, crear per les animacions d'eToy, 343
- controlar els costats del polígon, Experiment, 115
- controls de càmera, afegint i movent a Alice, 363, 373–374
- convació direccional, significat de, 47
- copyUpTo: mètode, exemple de, 232
- costats variable, utilitzar en polígons, 115
- costats, examinar en el comportament d'un animal tancat en una capsa, 313–314
- cotxe
  - afegir accions per a , 352
  - afegir sensors a, 350
  - dibuixar a eToy, 347
  - expressar diferents comportaments per, 350
  - girar en cercles a eToy, 347
- crear i moure un robot, Experiment, 37
- una creu suïssa, Experiment, 99
- creu, dibuixar amb paràmetres, 187
- creuRecorregut1:recorregut2: mètode, definir, 191
- cuc, animar, 342
- cursor, canviar a eToy el valor del, 346
- daly, crear robot, 35
- Debug, botó; a la finestra del depurador, efecte, 203
- declaració de variables, relació amb els missatges, 24
- depurador (*debugger*), Vegeu també errors; errors de programa
  - corregir errors amb parèntesis amb el, 275
  - definició del, 197
  - escriure arguments des del, 207
  - invocar, 202–203
  - tancar, 206
  - utilitzar als bucles infinitis, 263
- depurador, obrir la finestra del, 204
- descompondre problemes, significat de, 213
- destroy mètode a Alice, utilitzar, 369
- detectorDistancia mètode
  - amb una traça, 246
  - canviar els colors amb, 244
- dibuix, eina de; obrir, 79
- dibuixar
  - ABC del, 41
  - capturar, 66–67
  - elements de l'animació, 342
  - escala, 128–131
  - estrella, 91–95
  - figures, 170–172
  - figures geomètriques, 96–97
  - hexàgon, 57
  - pentàgon, 57
  - polígons regulars, 56
  - quadrats, 160–161, 168
  - robots, 79–81
  - triangle equilàter, 55
  - ulleres de llarga vista, 75
  - una casa, 56
- dibuixos de robots, guardar i carregar, 84
- dibuixos, deixar-los dins de contenidors, 343
- dibuixos, girar i fer zoom, 80
- direcció
  - aleatòria pel comportament animal, 314

apuntar cap a una, 303–304  
 apuntar en una, 300–301  
 canviar de, 38–41, 43, 59  
 obtenir pels robots, 299  
 direcció de referència, canviar, 54  
 direcció nord, moure el robot en la, 259, 265  
 direcció mètode  
     descripció i exemples de, 305  
     exemple de, 303–305  
 direccions absolutes  
     angles associats amb, 301  
     significat de, 38–39  
 disjunció (!) missatge  
     error de parèntesi relacionat amb, 277  
     exemples, 271–274  
 disjunció (or) missatges, exemple de, 271, 274, 278  
 un disseny abstracte senzill, Experiment, 171  
 dist: mètode, obtenir distància amb, 320  
 distància  
     comparar a les simulacions de comportament animal, 319–321  
     especificar a Alice, 365  
 distanciaDesde: missatge  
     codi per, 301  
     descripció i exemple de, 305  
 divisió (/ /), triar per conduir el volant d'eToy, 349  
 dobleMarc mètode, definir, 177–179  
 dobleMarcSenseCridarPatro mètode, definir, 179  
 doesNotUnderstand mètode, efecte de, 209  
 dos punts (:)  
     utilitzar en mètodes i múltiples arguments, 196  
     utilitzar en mètodes i paràmetres, 189  
     utilitzar en paràmetres, 192  
     utilitzar en selectors de missatge, 184  
 drawGrids mètode, utilitzar, 283  
 duplicar el marc, Experiment, 177  
 duració, especificar pels actors a Alice, 365  
 durada d'una acció, especificar a Alice, 370  
 eachFrame constant, utilitzar a Alice, 371  
 editor d'scripts a Alice, propietats del, 362  
 editor de text Espai de Treball Bot

guardar *scripts* amb, 65  
 obtenir, 62  
 propietats, 17  
 editor de text, utilitzar l'Espai de Treball Bot com a, 17  
 efectes especials, utilitzar a Alice, 373  
 elements de l'animació  
     dibuixar, 342  
     mostrar, 344–345  
 Eliminar Bots (botó), efecte de, 65  
 Eliminar camins (botó), efecte de, 65  
 Eliminar-ho tot (botó), efecte de, 65  
 el·lipses  
     al voltant dels missatges, significat de, 141  
     canviar l'aparença a eToy, 345  
 enrajar quadrats, 224–227  
 entorn  
     components del, 7  
     identificar robots, 7  
     instal·lar, 3  
     obrir, 5–7  
     sortir i guardar, 10  
 errors, *Vegeu també* depurador; errors de programa  
     aprendre de, 250–252  
     corregir, 209–211  
     generar-ne manipulant punts, 283–285  
 errors en el programa, *Vegeu també* depurador;  
     errors  
         detectar amb el color del text, 32–33  
         escriure malament el nom d'una variable, 28  
         escriure malament els selectors d'un missatge, 28  
         majúscules versus minúscules, 29  
         oblidar un punt, 31  
         panorama general, 27  
         variables no utilitzades, 29  
 escala  
     dibuixar, 128–131  
     dibuixar amb esglaons, 235–237  
     versió interactiva, 266  
 una escala, Experiment, 99  
 una escala de mà, Experiment, 101

- escala sense alçadors, Experiment, 100  
escala, Experiment, 40  
escapant:ambSortida: mètode, crear, 316  
escriu-ho (p) opció menú, seleccionar, 69  
escriure  
    arguments des del depurador, 207  
    resultat d'una expressió, 71  
esdeveniments, menú; mostrar dins el sistema eToy, 337  
esgraons  
    dibuixar una escala amb, 235–237  
    mesurar per dibuixar una escala, 129–131  
espai, caràcter; representar, 232  
una espiral a partir de línies, Experiment, 218  
una espiral amb quatre paràmetres, Experiment, 217  
una espiral d'angle constant, Experiment, 216  
espiral d'angle constant, definició, 216  
espiral, dibuixar, 132  
espirals a partir de quadrats, Experiment, 218  
espirals amb distància constant, Experiment, 217  
espirals, crear, 216  
Esquerra a Dreta, ordre d'execució, exemples de, 151–154  
est missatge, exemple de, 38  
est, indicar la direcció, 51  
estrella  
    dibuixar, 91–95  
una estrella, Experiment, 180  
una estrella amb seixanta branques, Experiment, 95  
eToy, sistema  
    botó basic, 344–345  
    canviar els elements d'un contenidor, 345  
    condicions i tests a, 350–351  
    cotxes i conductors, exemple, 347–353  
    crear animacions a, 341–347  
    crear icones a, 355–356  
    crear nous *scripts* amb, 335  
    crear sensors, 350  
    crear una carretera, 350  
    eliminar camins dibuixats pels robots, 355  
    executar *scripts*, 354  
    exemple pilotant un avió, 330–339  
    internacionalització, 356  
    modificar noms dels dibuixos, 332  
    mostrar el menú d'esdeveniments, 337  
    obrir un visualitzador, 332  
    personalitzar, 353–356  
    personalitzar tests basats en colors, 351  
    propietats del, 329  
    utilitzar *joystick* per l'exemple de l'avió, 339–340  
    utilitzar amb objectes 3D a Alice, 379  
    utilitzar observadors, 333  
    vincular objectes, 355  
execució de programes, utilitzar el *Transcript* per a la, 234–235  
Experiments, Vegeu també *Scripts*  
    algunes capses, 179  
    una altra escala estranya, 131  
    una altra espiral, 216  
    art abstracte, 40  
    canviar la direcció de referència, 54  
    col·locació de l'increment dins del bucle, 131  
    controlar els costats del polígon, 115  
    crear i moure un robot, 37  
    una creu suïssa, 99  
    un disseny abstracte senzill, 171  
    duplicar el marc, 177  
    escala, 40  
    una escala, 99  
    una escala de mà, 101  
    escala sense alçadors, 100  
    una espiral a partir de línies, 218  
    una espiral amb quatre paràmetres, 217  
    una espiral d'angle constant, 216  
    espirals a partir de quadrats, 218  
    espirals amb distància constant, 217  
    una estrella, 180  
    una estrella amb seixanta branques, 95  
    fletxes, 294  
    frAnkenstein, 105  
    girar el quadrat, 49  
    una grapa, 100  
    hexàgon regular, 97

- un senzill laberint, 132
- un mètode pel marc art nouveau, 172
- un mètode per dibuixar un hexàgon, 186
- un mètode per dibuixar una creu, 187
- moure les agulles del rellotge, 53
- ones quadrades en un estany quadrat, 214
- un passadís llarg, 133
- un passadís, 215
- pentàgon regular, 97
- PICA, 41
- una pinta, 100
- piràmide, 136
- piràmide amb deu esglaons, 98
- una piràmide amb un nombre variable d'esglaons, 113
- una piràmide amb una mida variable de l'esglaó, 113
- la piràmide esglaonada de Saqqara, 40
- Una Piràmide Rectangular, 225
- una piràmide triangular, 226
- posar una traça dins del bucle, 236
- quadrat, 39
- un quadrat relatiu, 48
- un quadrat trencat, 49
- un quadrat utilitzant un bucle, 96
- quadrats, 136
- quadrats que fan tombarelles, 101
- Quadrats Russos, 215
- quadrats russos, 133
- rectangle 1, 288
- rectangle 2, 288
- el rectangle d'or, 111
- rectangles d'or incrementals, 224
- un rellotge "real", 55
- una roda, 177
- scripts* misteriosos, 47
- Scripts* que no funcionen, 112
- SOS, 38
- tauler de dames, 226
- traslladar un robot a un punt, 296
- Triangle 1, 289
- triangle 2, 292
- trieu vosaltres, 180
- utilitzar el mètode trasllada: 1, 297
- utilitzar el mètode trasllada: 2, 297
- Explorador de la classe Bot
  - propietats de, 161
  - utilitzar el, 162
- expressió Bot nou est, descomposició, 151
- expressió Bot nou, executar, 70
- expressió d'assignació (:=)
  - parts dreta i esquerra, 119–120
  - utilitzar en variables, 106–108
- expressions
  - components de, 140
  - equivalents amb parèntesis, 154
  - escriure els resultats de, 71
  - executar, 72
  - exemples de, 23
  - per dibuixar una escala, 131
  - relació amb els mètodes, 167
  - relació amb els programes, 19, 21
  - self halt, 202–203
  - utilitzar claudàtors ([] amb, 267
  - utilitzar variables dins de, 121
- expressions booleanes
  - combinar, 271–274
  - errors, 275
  - exemples de, 269–271
- expressions condicionals
  - amb una sola bifurcació, 247–248
  - components de, 246
  - imbricar, 248–250
  - utilitzar per canviar els colors d'un robot, 243–245
  - utilitzar traces amb, 245–246
- fàbriques
  - classes com a, 36
  - fàbrica Color, 76
  - relació amb objectes manufacturats, 19
- fàbriques de robots, *Vegeu també* classes
  - enviar missatges a les, 10
  - re-equipar, 82–83

- false i true, objectes; retornar amb expressions booleanes, 270  
Fes-ho (botó), efecte de, 65  
fes-ho (d) missatge, efecte de, 69  
Fes-ho tot (botó), efecte de, 17–19, 65  
figures  
    dibuixar, 170–172  
    exemples de, 176–178  
figures geomètriques, dibuixar, 96–97  
finsA100 mètode  
    afegir una traça a, 260–262  
    efecte de, 258  
finsA100Infinit, mètode; executar, 264  
first mètode, exemple de, 232  
fitxer *changes*, resoldre problemes amb, 12–13  
fitxer de l'aplicació Squeak, resoldre problemes relacionats amb, 12  
fitxer imatge, resoldre problemes amb, 12–13  
fitxers  
    importar, 67  
    problemes amb, 12–13  
fletxes, Experiment, 294  
for: mètode; utilitzar a Alice, 366  
forma dels robots, canviar, 77–79  
forward mètode, arrossegar i deixar a eToy, 347  
frAnkenstein, Experiment, 105  
fromUser mètode, triar colors amb, 76
- getter*, mètodes; mostrar per a les categories a eToy, 333  
giraA: mètode; efecte de, 300–301, 305  
giraA: unAngleAbsolutEnGraus, missatge; codi del, 300  
giraDreta: missatge, efecte de, 45–47, 52–54  
giraEsquerra: missatge, efecte de, 9, 45–47  
girar a l'esquerra 45 graus, resultat de, 51  
girar dibuixos, 80  
girar dins Alice, panorama general, 365–366  
girar el quadrat, Experiment, 49  
girar:, mètode; efecte de, 47  
girDeColors: mètode  
    definir, 250
- depurar, 251–252  
girs, examinar en el comportament animal, 310–314  
Glossari per a mètodes, 173  
gràfics  
    associar amb la classe Bot, 85–87  
    carregar, 81  
    carregar i guardar, 84  
    carregar i guardar per a la classe Bot, 84  
    dibuixar i preservar pels robots, 79–81  
    guardar i restaurar, 81–87  
    utilitzar *scripts* amb, 83–87  
una grapa, Experiment, 100  
guardaImatge mètode, descripció i exemple, 88  
guardaImatge: missatge, efecte de, 84  
guardar el contingut, opció del menú, accedir, 65  
guardar l'entorn Squeak, 10
- halo de nanses  
    accedir, 67, 71  
    aconseguir-ne informació, 72  
    explicació, 63–65  
    manipular per l'avió, 330–337  
    obtenir en dibuixar robots, 67, 79  
    per conduir el volant a eToy, 348  
*heading* valor, per conduir el volant a eToy, 348  
heptàgon, dibuixar amb el mètode poligon100:, 188  
hexàgon regular, exemple de, 114  
hexàgon regular, Experiment, 97  
hexàgons  
    dibuixar, 57  
    dibuixar amb paràmetres, 186  
    exemple de, 114  
hide i show mètodes a Alice, utilitzar, 369
- icona, crear dins d'eToy, 355  
idiomes, tria a eToy, 356  
ifFalse:; mètode, *Vegeu siFals:*  
ifTrue:ifFalse:; mètode, *Vegeu siCert:siFals:*  
imatges  
    aplicar als robots, 87  
    canviar, 85  
    guardar, 84  
    restaurar les classes, 84

importar fitxers, 67  
 indicar els arguments per a, 142  
 inicialització de variables, 134  
 inicialitzalmatge (missatge de la classe Bot), efecte de, 84  
 instal·lació  
     ajuts a la, 7  
     en sistemes Windows i Macintosh, 3  
     problemes amb, 12–13  
 interactiva, aplicació; exemple de, 265–266  
 internacionalització, implementar a eToy, 356  
 interpreta: mètode, definir, 252  
 interpretar un petit llenguatge, 252  
 Into, botó; utilitzar al depurador, 206  
 invisibilitat i visibilitat, aplicar als robots, 42  
**joystick**  
     crear per a un avió, 339  
     experimentar amb, 339  
     vincular a un *script*, 339–340  
 un senzill laberint, Experiment, 132  
 laberints, crear, 214–215  
 Lindenmeyer sistemes, significat de, 252  
 líniaDivisoria: mètode, desenvolupar per a rectangles daurats, 222  
 línies  
     crear espirals a partir de, 218  
     dibuixar i acolorir, 74–75  
     dibuixar per a una estrella, 92  
 línies de referència, dibuixar, 53  
 llapis, mida i color, 74–75  
 llenguatge de programació orientat a objectes, Smalltalk com a, 19  
 llenguatge petit, interpretar, 252–254  
 llenguatges de programació, 19  
 lletra A  
     dibuixar, 41–42  
     forma de, 104  
     utilitzar moviments absoluts amb, 293–294  
     utilitzar variables, 108–109  
     variacions, 104–106  
 llista de fitxers (*file list*), carregar *scripts* amb, 65

llocs web  
     eToy, 329  
     Squeak, 3  
 longitud, examinar la influència en el moviment de vagar de la, 309  
 longitudCami, variable  
     canviar dues vegades, 122  
     declarar, 120–123  
     definir variable amb, 123  
     escriure i llegir, 120  
     inicialitzar, 123–124  
 longitudTotal variable, utilitzar en polígons, 115  
 lookLike mètode, crear *scripts* amb, 344  
 màquina virtual (MV), fitxer d'aplicació Squeak com a, 12  
 mètodes d'*scripting a eToy*, 354  
 Macintosh  
     instal·lar Squeak en un, 4  
     obrir l'entorn en un, 5  
 majúscules, lletres; errors relacionats amb, 29  
 mantenirOrientació: mètode, definir, 322  
 menjar, trobar-ne a les simulacions de comportament animal, 319–326  
**mentreCert: bucle**  
     components de, 259  
     convertir a mentreFals:, 262  
     descripció i exemple de, 268  
     efecte de, 257  
**mentreFals: bucle**  
     components de, 259  
     descripció i exemple de, 268  
     efecte de, 257  
 menú principal, opcions del, 61  
 menú, opcions del; explicacions, 62  
 menús contextuais, accedir a, 65  
 menús, mostrar per Món, 11  
 mètode capsula, definir mètode estrella amb, 180  
 un mètode pel marc art nouveau, Experiment, 172  
 un mètode per dibuixar un hexàgon, Experiment, 186  
 un mètode per dibuixar una creu, Experiment, 187

- mètodes, Vegeu també compondre mètodes  
afegir per a l'avió, 337–339  
afegir traces, 260–263  
arroseggar i deixar anar per crear nous *scripts*  
per a l'avió, 335  
compilar i comprovar, 166  
components de, 166–168  
cridar altres mètodes amb, 178  
definició dels, 25  
definir amb el mètode quadrat, 179  
definir amb l'Explorador de la classe Bot, 162,  
165–166  
definir amb múltiples arguments, 184  
definir amb múltiples paràmetres, 189  
definir per dibuixar rectangles d'or, 219–221  
depurar, 207  
entrar sense executar, 207–209  
mostrar pel *joystick* de l'avió, 339  
mostrar per a les categories d'*eToy*, 333  
per dibuixar quadrats, 184–185  
recompilar amb el depurador, 209  
relació amb les expressions, 167  
utilitzar arguments amb, 184, 189  
utilitzar i reutilitzar, 166, 178  
valor retornat per, 246  
variables en, 187–189  
versus *scripts*, 160–161, 168–169
- mètodes, categories de, 163–164  
mètodes, definicions; modificar amb el depurador,  
209, 211  
mètodes, execució, 195–196  
mida dels robots, canviar, 77–79  
mida, variable; declarar, 198  
midaCostat paràmetre  
utilitzar a l'experiment dels quadrats russos,  
133  
utilitzar al mètode quadrat:, 185  
midaEsglaons variable, utilitzar en piràmide, 113  
midaLlapis mètode, descripció i exemple, 88  
minúscules, lletres; errors relacionats amb, 29  
missatge Bot nou ves: 100 + 20, descomposició, 149  
missatge, enviaments de
- efecte de, 142  
prioritat dels, 153  
missatge, examinar l'execució del, 199–200  
missatge, selectors de  
Color, per a la classe, 74  
escriure malament els, 28  
utilitzar dos punts (:) amb, 184
- missatges, Vegeu també binaris, missatges; cascades;  
paraula-clau, missatges; ordre d'execució;  
unaris, missatges  
components dels, 139  
enviar a fàbriques de robots, 10  
enviar als robots, 7–10  
enviar amb cascades, 15  
enviar parts dels actors, a Alice, 367–368  
executar, 207  
executar, a Alice, 364  
exemples comuns de, 63  
exemples de, 7, 23–24  
identificar, 141–143  
oblidar punts entre, 31  
ordre d'execució dels, 276, 284  
resultats dels, 67  
tipus de, 139–141  
utilitzar bucles amb, 93–95  
utilitzar en expressions booleanes, 270–274  
variacions a Alice, 366
- missatges compostos  
definició de, 9  
enviar, 9  
exemples de, 9–10  
representar, 147–148
- mitja altura, variable; relació amb l'amplada i  
l'altura, 104
- Món (*World*) menú  
canviar la mida de la pantalla amb, 283  
mostrar, 11  
obrir projectes *morphic* amb, 329  
*morphic*, projecte; obrir, 329  
*morphs*, crear a Alice, 377  
Morse, codi; dibuixar missatge SOS en, 37  
moure les agulles del rellotge, Experiment, 53

move toward mètode, utilitzar a eToy, 355–356  
 move: mètode, utilitzar a Alice, 365–366  
 moveTo: i move: mètodes a Alice, utilitzar, 369  
 moviment, *Vegeu també* robots en moviment  
     a Alice, 365–366  
     relatiu versus absolut, 285–288  
 moviments absoluts  
     fer, 285  
     significat de, 292–294  
 moviments de robot, seguiment dins dels *scripts*, 235  
 MV (màquina virtual), fitxer d'aplicació Squeak  
     com a, 12

n vegadesRepetir: [], efecte de, 93, 101  
 nances, *Vegeu* halo de nances  
 nances de l'halo, descripció de, 330  
 negació (not) missatge, exemple de, 271–274, 278  
 negated mètode, utilitzar en punts, 295  
 negreta (tipus de lletra), significat de, 200  
 nil valor, significat de, 197–199, 210–211  
 nom dels dibuixos, modificar a eToy, 332  
 només lectura, identificar fitxers de, 12  
 nombreDeCostats variable, utilitzar en mètodes, 188  
 nombreEsglaons variable, utilitzar en piràmide, 113  
 nombres i cadenes, panorama general, 233, 235  
 nord missatge, exemple de, 38  
 nordEst missatge, exemple de, 38  
 nordOest missatge, exemple de, 38  
 not (negació) missatge  
     error amb parèntesis, 275  
     exemple de, 271–274, 278  
 nou, missatge; efecte de, 10  
 nudge: mètode, utilitzar a Alice, 368

o (!) missatge  
     error de parèntesi relacionat amb, 277  
     exemple de, 271, 273–274, 278

o (disjunció) missatge  
     exemple de, 273–274, 278

objectes  
     classes com a fàbriques de, 274  
     comportament a Smalltalk, 67  
     crear per classes, 36

en missatges binaris, 144  
 manipular dins d'Alice, 367  
 obtenir de classes, 26  
 relació amb missatges binaris, 140  
 vincular dins d'eToy, 355–356  
 obre... submenú, descripció de, 61  
 observadors, utilitzar a eToy, 333  
 oest missatge, exemple de, 38  
 ones quadrades en un estany quadrat, Experiment, 214  
 operadors booleans, exemples de, 274  
 oques volant, patrons de, 291, 295–296  
 ordre d'execució, *Vegeu també* missatges  
     de missatges, 277, 283–285  
     panorama general, 146–147  
     Regla 1, 146–150, 154  
     Regla 2, 146, 150–151  
     Regla 3, 146, 151–154  
 origen, missatge; efecte de, 301  
 Over, botó; utilitzar al depurador, 207

pablo, variable; declarar, 118–119  
 Paint, eina; obrir per dibuixar un avió, 330  
 pantalla  
     canviant la mida de la, 283  
     mostrant informació a la, 230–231  
     posicionar robots al centre de la, 301  
 pantalla completa, mode; canviar a, 283  
 pantalla, capturar regions de la, 67  
 parèntesi ()  
     errors provocats per, 274–278  
     inclosos per alterar l'ordre d'execució, 146, 154  
     utilitzats amb punts, 283

paral·lel, moviment; exemple de, 286  
 paràmetres  
     i arguments, 193–196  
     declarar, 192  
     definició de, 184  
     definir mètodes amb, 188  
     dibuixar quadrats amb, 192  
     donar nom als, 190  
     no es pot assignar valors als, 192

- propietats dels, 192  
utilitzar, 186  
utilitzar en polígons, 187  
utilitzar amb espirals, 216  
i variables, 191–193
- paràmetres, canviar el valor dels, 192
- paraula-clau, missatges, *Vegeu també* missatges  
exemples de, 139–141, 145  
explicació de, 139–141
- paraules, seleccionar en un *script*, 69
- un passadís llarg, Experiment, 133
- un passadís, Experiment, 215
- passalimatgeAClasse mètode, descripció i exemple, 88
- passaImatgeAClasse missatge, efecte de, 82
- patro, mètode  
definir, 199  
depurar, 209  
examinar, 207  
recompilar amb el depurador, 211  
seleccionar a la finestra del depurador, 204
- patro4, mètode  
definir, 176–178, 199  
seleccionar dins del depurador, 205
- patroInclinat mètode, significat de, 178
- patrons, *Vegeu figures*
- pentàgon regular, exemple de, 114
- pentàgon regular, Experiment, 97
- pentàgon, dibuixar amb el mètode *poligon100:*, 189
- pentàgon, exemple de, 114
- pica color, expressió; executar, 70
- pica color: Color groc, descomposició de l'execució de, 147
- pica midaLlapis: pica midaLlapis + 2 expressió, descomposició de, 149
- pica ves: 100 + 20 expressió, descomposició de, 148
- PICA, Experiment, 41
- pica, robot  
assignar un gràfic a, 86  
com a variable, 108  
crear, 35
- pica, variable; declarar, 118–122
- pila d'execució, exemple de, 200
- pila, depurador; anar pas a pas per la, 206
- una pinta, Experiment, 100
- pintar amb la nansa vermella, utilitzar en animacions, 342
- piràmide amb deu esglaons, Experiment, 98
- una piràmide amb un nombre variable d'esglaons, Experiment, 113
- una piràmide amb una mida variable de l'esglaó, Experiment, 113
- la piràmide esglaonada de Saqqara, Experiment, 40
- Una Piràmide Rectangular, Experiment, 225
- una piràmide triangular, Experiment, 226
- piràmide, Experiment, 136
- píxels  
determinar pel moviment endavant del robot, 121  
significat de, 22
- place: mètode a Alice, utilitzar, 370
- playerAtCursor mètode, arrossegant a eToy, 345
- playfield options menú, mostrar a eToy, 353
- pointAt: mètode d'Alice, 369
- polígon100, 188
- polígon  
automatitzar amb variables, 114–115  
de mida fixa, 115  
dibuixar, 56–57  
utilitzar variables i paràmetres amb, 187
- polígon regular  
de mida fixa, 115  
dibuixar, 56–57
- poligon:mida: mètode, definir, 189
- Pooh, generar formes 3D a partir de figures 2D amb el sistema, 378–379
- PopUp menú, exemple de, 230–231
- posar una traça dins del bucle, Experiment, 236
- posició  
determinar la, 305  
especular amb, 301  
representació com a punts, 280  
saltar a una, 285  
versus centre, 305

- vincular amb l'angle del volant a eToy, 348–350
- posició actual dels robots, determinar, 305
- posició missatge, descripció i exemple de, 305
- posicioSiVes: mètode, codi per, 302
- Primer els Parèntesis, ordre d'execució; exemples de, 150–151
- primera classe, objectes de; descripció a Alice, 367
- Proceed, botó; utilitzar al depurador, 203, 206, 211
- profunditat de color, comprovar i modificar, 358
- programes
  - definició, 19
  - escriure i executar, 20
- punt (.)
  - oblidar un, 31
  - utilitzar en missatges i *scripts*, 24
- punt1 \* nombre missatge, descripció i exemple, 298
- punt1 + punt2 missatge, descripció i exemple, 298
- punt1 negated missatge, descripció i exemple, 298
- punts
  - apuntar cap a un, 303–304
  - distància des d'un, 301
  - i moviments absoluts, 292–294
  - panorama general, 280–282
  - per simular visió en comportaments animals, 319–322
  - traslladar robots a, 296
  - utilitzar negated amb els mètodes setX:setY;, 294
- quadrícula, utilitzar, 282–283
- quadrat, Experiment, 39
- un quadrat relatiu, Experiment, 48
- un quadrat trencat, Experiment, 49
- un quadrat utilitzant un bucle, Experiment, 96
- quadrat, *script*, utilitzar variable amb, 191
- quadrat: mètode
  - definir mètodes amb, 179
  - execució de, 195
  - reproduir figures amb, 224–227
  - retornar el receptor del missatge amb, 170
  - utilitzar el paràmetre midCostat amb, 185
- quadrats, *Vegeu també* quadrats centrats; quadrats concèntrics
  - dibuixar, 48, 96, 160–161
  - dibuixar amb mètodes, 184–187
- quadrats centrats, crear, 214–215, *Vegeu també* quadrats
- quadrats concèntrics, dibuixar, 133, *Vegeu també* quadrats
- quadrats que fan tombarelles, Experiment, 101
- Quadrats Russos, Experiment, 215
- quadrats russos, Experiment, 133
- quadrats, Experiment, 136
- Quick Reference, botó dins l'editor d'*scripts* d'Alice; descripció de, 361
- reaccions, tractant-les dins Alice, 376
- receptors
  - amagar i mostrar, 59
  - enviar missatges a, 169
  - explicació de, 142
  - moure si va a parar dins d'un rectangle, 302
  - retornar amb el mètode quadrat, 170
  - trobar-ne la posició en simulacions de comportament animal, 319
- receptors dels missatges, *Vegeu missatges*
- rectangle 1, Experiment, 288
- rectangle 2, Experiment, 288
- el rectangle d'or, Experiment, 111
- rectangleAmplada:altura: mètode, definició, 190
- rectangleDaurat: mètodes, desenvolupar i exemples de, 221–224
- rectangles, *Vegeu també* capses; rectangles d'or
  - centrats en animals, 312
  - com a regions segures, 317
  - com a sortides d'un animal atrapat dins d'una capsà, 315
  - dibuixar, 16, 302
  - moure robots dins de, 302–303
- rectangles d'or, *Vegeu també* rectangles
  - dibuixar, 219–220
  - imbricar, 219, 222
  - un-rectangle-per-línia, 220–224

- rectangles d'or incrementals, Experiment, 224  
Rectangles, experiments, 288  
regió segura, imaginar un rectangle com a, 317  
Regla 1 de l'ordre d'execució  
    conseqüències, 154  
    explicació de, 146  
    Unari > Binari > Paraula-clau, 147–150  
Regla 2 de l'ordre d'execució  
    explicació de, 146  
    primer els parèntesis, 150–151  
Regla 3 de l'ordre d'execució  
    d'esquerra a dreta, 151–154  
    explicació de, 146  
relatiu  
    moviment, versus moviment absolut, 285–288  
relativa  
    orientació, versus orientació absoluta, 48–50,  
        279  
un rellotge “real”, Experiment, 55  
rellotge robot, crear, 55  
rellotge, crear, 54  
repeticions condicionals  
    components de, 258–260  
    definir, 262  
    efecte d'executar només un cop, 263  
    exemple de, 257–260  
request:initialAnswer: missatge, efecte de, 231  
resize: mètode, utilitzar dins Alice, 368  
resoldre problemes d'instal·lació d'Squeak, 12–13  
Restart, botó; utilitzar al depurador, 206  
resultats d'un missatge, significat de, 67  
*return* caràcters, representar, 232  
robot aranya, crear amb la classe Bot, 81–83  
robots  
    acolorir, 248–250  
    aplicar imatges als, 87  
    apuntar en les direccions principals de la, 38  
    canviar colors, 43, 76, 243–245  
    canviar els colors, 37  
    canviar l'aparença, 85  
    canviar la direcció dels, 59  
    canviar la forma i la mida, 77–79  
    crear, 10, 26, 35, 43  
    crear picas, 17  
    determinar els colors dels, 70  
    determinar la posició actual, 305  
    dibuixar, 79–81  
    enviar missatges a, 7  
    fer moviments absoluts amb, 285  
    girar un angle determinat, 46  
    identificar dins l'entorn, 7  
    interactuar amb, 9  
    moure, 42, 72  
    moure al nord, 259, 266  
    moure dins de, 302–303  
    obtenir direcció pels robots, 299  
    obtenir informació sobre, 7  
    pintar, 79  
    tornar al centre de la pantalla, 301  
    tornar invisible, 42  
    traslladar a un punt, 297  
una roda, Experiment, 177  
rodar a Alice, panorama general, 365–366  
rotació, determinar pel volant d'eToy, 348–350  
  
salta: mètode, utilitzar a l'Experiment PICA, 41  
salta: missatges  
    efecte de, 37  
    efecte en moviments absoluts, 285  
saltaA: mètode versus vesA: unPunt, 285  
saltaA: missatge, descripció i exemple, 298  
salts, dibuixar la lletra A amb, 293  
scheduler getTime expressió, utilitzar dins  
    Wonderland, 374  
script amb diversos actors a Alice, exemple de, 373  
scripts, Veieu també Experiments  
    afegir traces als, 237  
    analitzar, 21–22, 120–124  
    analitzar a Alice, 364  
    carregar, 65  
    crear amb el mètode lookLike, 344  
    crear per a l'avio, 335  
    declarar variables pels, 43  
    definició de, 15

escriure, 17–18  
 executar, 69–71  
 executar a eToy, 354  
 guardar amb l’Espai de Treball Bot, 65  
 pel rectangle d’or, 111  
 relació amb les expressions, 20  
 seleccionar el text complet de, 69  
 utilitzar dins d’Alice, 362–366  
 utilitzar en operacions gràfiques, 83–84  
 versus mètodes, 160–161, 168–169  
 vincular el *joystick* de l’avió amb, 339–340  
*scripts* amb nom, Vegeu mètodes  
*scripts* misteriosos, Experiment, 47  
*Scripts* que no funcionen, Experiment, 112  
 secció àuria, calcular, 219  
 segments de línia, dibuixar, 37  
 seleccionar amb el ratolí, 65  
 selectors de missatges matemàtics, prioritats, 154  
 self halt expressió, obrir el depurador amb, 202–203  
 self variable, relació amb els mètodes, 168, 196  
 sensors pel cotxe, crear a eToy, 350  
 set language... menú, mostrar a eToy, 356  
 setColor: mètode, utilitzar a Alice, 368  
*setter*, mètodes; mostrar per a les categories a eToy, 333  
 setX:setY: mètode, utilitzar en punts, 295  
 siCert:siFals: mètode  
     efecte de, 244  
     utilitzar bloc condicional buit amb, 247  
 siFals: mètode, efecte de, 247–248  
 simulacions de comportament animal  
     atrapat dins una capsà, 312–316  
     decreix la velocitat, 319  
     trobar menjar, 319–326  
     vagar, 307–312  
     visió, 324–326  
 sistema de coordenades a Smalltalk, descripció de, 281  
 sistemes de coordenades matemàtic, comparació amb Smalltalk, 281  
 size mètode, exemple de, 232  
 Smalltalk

anomenar les variables, 117  
 blocs (*block*), 94  
 comportament dels objectes a, 67  
 donar forma al codi en, 96  
 llenguatge de programació, 36  
 selectors de missatges matemàtics, 154  
 significat de, 19–20  
 sistema de coordenades, 281  
 solapa  
     definició de, 7  
     obtenir Espai de Treball Bot, 62  
 sortir de l’entorn Squeak, 10  
 SOS missatge en codi morse, dibuixar, 37  
 SOS, Experiment, 38  
 sources, fitxer; resoldre problemes amb, 12  
 speed: mètode, utilitzar a Alice, 365  
 Squeak  
     *downloading*, 3  
     fitxers necessaris per a, 12  
     instal·lació, 3–5  
     interaccionar amb, 63  
     i el llenguatge de programació Smalltalk, 19–20  
     sortir i guardar l’entorn, 10  
 standUp mètode, aplicar a actors dins Alice, 360  
 start script mètode, utilitzar a eToy, 354  
 subfinestres de l’Explorador de la Classe Bot,  
     explicació de, 162  
 submenú, explicació de, 61  
 subratllats, receptors de missatge; significat de, 141–143  
 sud missatge, exemple de, 38  
 sudEst missatge, exemple de, 38  
 sudOest missatge, exemple de, 38  
 tab caràcter, representar, 232  
 tallar i enganxar, generar polígons regulars amb, 57  
 tauler de dames, Experiment, 226  
 tauler, construcció de, 136  
 tecles, combinacions i botons del ratolí,  
     explicacions, 63  
 temps  
     conèixer dins del *Wonderland* a Alice, 374

- versus angles, 55
- text, color; canvis del, 31
- timesRepeat:, Vegeu vegadesRepetir:
- tornarInvisible i tornarVisible mètodes, efectes de, 45, 59, 305
- traces
  - afegir als mètodes, 260–263
  - endevinar les generades pel moviment dels animals, 309
  - fer que l'avió dibuixi, 337
  - generar, 235–237
  - per simular la visió en el comportament animal, 324–326
  - utilitzar en expressions condicionals, 245–246
- Transcript*, eina
  - generar traces de programes amb, 235–237
  - utilitzar al mètode finsA100, 261
  - utilitzar el, 234–235
  - utilitzar en expressions condicionals, 245–246
- translacions
  - i bucles, 295–296
  - d'oques volant, 291–292
  - panorama general, 288–290
  - de triangles, 290–291
- traslladar un robot a un punt, Experiment, 296
- tres radis, dibuixar figura amb, 58
- Triangle 1, Experiment, 289
- triangle 2, Experiment, 292
- triangle equilàter, dibuixar, 55
- triangles
  - dibuixar, 55
  - traslladar, 290–291
- triangular, forma; aplicar als robots, 77
- trieu vosaltres, Experiment, 180
- trobaAreaAlimentPerDistancia: mètode;
  - implementar, 320–321
- true i false, objectes retornats amb expressions booleans, 269–271
- turn: mètode
  - afegir per a un avió, 337
  - arrossegar i deixar a eToy, 347
  - utilitzar dins Alice, 365–366
- ulleres de llarga vista, dibuixar, 75
- unaCondició (mètode), descripció i exemple de, 255
- Unari > Binari > Paraula-clau ordre d'execució, exemples de, 147–150
- unaris, missatges, Vegeu també missatges
  - exemples de, 141, 143
  - explicació de, 139–141
  - ordre d'execució, 146
- unDrawGrids mètode, utilitzar, 283
- until: mètode; utilitzar a Alice, 365
- usuari, comunicació amb, 230–231
- usuari, interacció amb; implementar dins Alice, 375–376
- utilitzar el mètode trasllada: 1, Experiment, 297
- utilitzar el mètode trasllada: 2, Experiment, 297
- vagar, simular, 307–312
- valor dels arguments, visualitzant-los amb el depurador, 207
- valorMida paràmetre, exemple de, 190
- valors
  - assignar a variables, 107
  - canviar el d'una variable, 134
  - excloure d'una variable, 123
  - modificar els de les variables a eToy, 333
  - relació amb les variables, 107
  - representar amb nil, 197–199
  - retornar, 169–170
  - retornat pels mètodes, 246
- valors booleans, exemples de, 269
- variable angle
  - utilització amb polígons, 115
  - utilització en mètodes, 188
- variable, nom de
  - escriure malament, 28
  - importància, 118
  - self, 168
- variable, valor; canviar, 193
- variables
  - anomenar, 117
  - com a argument, 194
  - assignar valors a, 107

- automatitzar polígons amb, 114–115
- canviar el valor de les, 134
- combinar amb bucles, 131–134
- com a contenidors, 118–119
- declarar, 107
- declarar i assignar, 116
- declarar per *scripts*, 42
- declarar, inicialitzar i utilitzar, 120–122
- definir, 106
- definir amb la variable longitudCami, 123
- errors relacionats amb, 28–29
- experimentar amb, 111–114
- expressar relacions entre, 110–111
- inicialitzar, 106–107, 134
- introduir en bucles, 134
- en mètodes, 187–189
- modificar els valors a eToy, 333
- i paràmetres, 191–193
- per a la longitud en el camí de pica, 121
- potència de, 110
- referir-nos a les, 108
- utilitzar, 134
- utilitzar dins d'expressions, 121
- utilitzar en dibuixar una escala, 129
- utilitzar en l'*script* del quadrat, 192
- utilitzar en l'Experiment de la piràmide de Saqqara, 113
- utilitzar en la lletra A, 108–109
- utilitzar sense valor, 122
- valor per defecte de, 197–199
- vegadesRepetir:** mètode
  - arguments de, 94
  - efecte de, 93, 101
  - mostrar en el depurador, 205
- ves missatge**
  - avançar píxels, 37
  - dibuixar rectangles amb, 16
  - efecte de, 9
- ves:siDinsCapsa:** mètode, codi per al, 302
- vesA:** mètode
  - descripció i exemple, 297
  - versus saltaA: unPunt, 285
- viewer, Vegeu** visualitzador
- visió, simular en el comportament animal, 324–326
- visibilitat del robot, controlar, 42
- visibilitat i invisibilitat, aplicar als robots, 42
- visualitzador (*viewer*)**
  - de la base de l'animació, 343
  - mostrar categories en el, 333
  - obrir dins eToy, 333
  - obrir un contingut en un, 344
  - parts de, 334
- vocabulari, subfinestra; llistar missatges a la, 62
- volant
  - dibuixar a eToy, 347
  - utilitzar la direcció del, 348–350
- w makePlaneNamed:** expressió; utilitzar a Alice, 372
- whileFalse:** bucle, Vegeu mentreFals: bucle
- whileTrue:** bucle, Vegeu mentreCert: bucle
- widgets,** solapa; obrir per dibuixar avió, 330
- Windows**
  - instal·lar Squeak a, 4
  - obrir l'entorn a, 5
- Wonderland, Vegeu també** Alice, entorn de creació personalitzada
  - alarmes, 374
  - crear dins Alice, 372–373
  - obrir per utilitzar amb Pooh, 378
- The Worlds of Squeak** finestra, mostrar, 357
- x@y missatge,** descripció i exemple, 298
- yertle,** robot; aparició de, 86
- zoom,** fer amb els dibuixos, 80