Racket et POO

Exemple : une classe Pile (classe pile%) :

-implémente des objets modélisant des piles et qui réagit aux message :

. Vide?

. sommet

. empiler

. depiler

Racket et POO

```
1  (define pile%                          ; une sous-classe de object%
2    (class object%                       ; variable d'instance privée
3      (define L '())
4      (define/public (vide?)             ; méthode publique
5        (null? L))
6      (define/public (sommet)            ; méthode publique
7        (if (null? L) (error "Pile vide !") (car L)))
8      (define/public (empiler x)         ; méthode publique
9        (set! L (cons x L)))
10     (define/public (depiler)           ; méthode publique
11       (if (null? L) (error "Pile vide !") (set! L (cdr L))))
12     (super-new)))                      ; le constructeur de la classe mère
```

Racket et POO

```
> (define p (new pile%))
> (send p empiler 'a)
> (send p empiler 'b)
> (send p empiler 'c)
> p
#(struct:object:pile% ...)
> (send p vide?)
#f
> (send p sommet)
c
> (send p depiler)
> (send p sommet)
b
```

Racket et POO

Toutes les méthodes sont-elles publiques?

. Oui la plupart

. mais on peut définir des méthodes privées (non utilisables par un utilisateur mais que l'on peut invoquer dans le texte de la classe

Racket et POO

• Peut on initialiser des champs (attributs)? Oui

Dans le cas de la classe pile%, il n'y a qu'une variable L privée initialisée à la liste vide lors de la création.

• Sinon possibilité de définir des valeurs initiales lors de l'instantiation

(exemple classe balle% avec deux champs a et b :

(define b (new balle% (a 10) (b 0)))

Variante de new : (define b (make-object balle% 10 0))

Racket et POO

```
26 (define balle%
27   (class object%
28     (init-field (a 0) (b 0))      ; deux champs initialisables a et b
29     (define L (cons a b))         ; une variable privée L
30     (define/public (pos)          ; accesseur aux champs
31       (list a b))
32     (define/public (move dx dy)       ; translation de la position
33       (set! a (+ a dx)) (set! b (+ b dy)))
34     (define/public (reset)            ; remise à la position initiale
35       (set! a (car L)) (set! b (cdr L)))
36     (super-new)))
```

```
37 > (define b1 (new balle% (a 10)))
38 > b1
39 #(struct:object:balle% ...)
40 > (send b1 pos)
41 (10 0)
```

```
42 > (send b1 move 5 2)
43 > (send b1 pos)
44 (15 2)
45 > (send b1 reset)
46 > (send b1 pos)
47 (10 0)
```

Racket et POO

```
48 > (define b2 (new balle% (a 10)))
49 > b2
50 #(struct:object:balle% 10 0 (10 . 0))      ; a, b publics, et L privé
51 > (send b2 move 5 2)
52 > b2
53 #(struct:object:balle% 15 2 (10 . 0))      ; c'est pratique...
```

Racket et POO

Sous-Classes et héritage

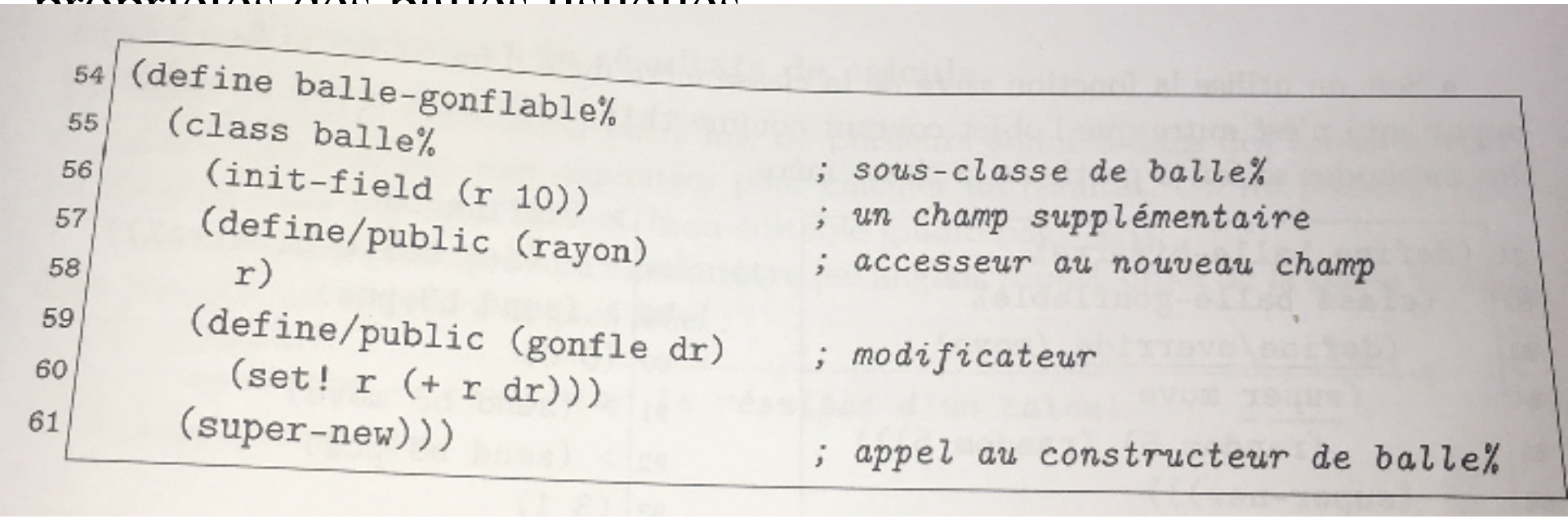
Evidemment une classe peut avoir des sous-classes

Exemple typique : toute classe est sous-classes de la classe object%

Autre exemple : création d'une sous-classe de balle pour décrire une balle ayant un rayon r.

Racket et POO

Cette nouvelle classe (disons balle-gonflable%) héritera des propriétés des balles usuelles



```
54 (define balle-gonflable%
55   (class balle%                          ; sous-classe de balle%
56     (init-field (r 10))                  ; un champ supplémentaire
57     (define/public (rayon)               ; accesseur au nouveau champ
58       r)
59     (define/public (gonfle dr)           ; modificateur
60       (set! r (+ r dr)))
61     (super-new)))                        ; appel au constructeur de balle%
```

Racket et POO

Ajout :

. nouveau champ rayon initialisante

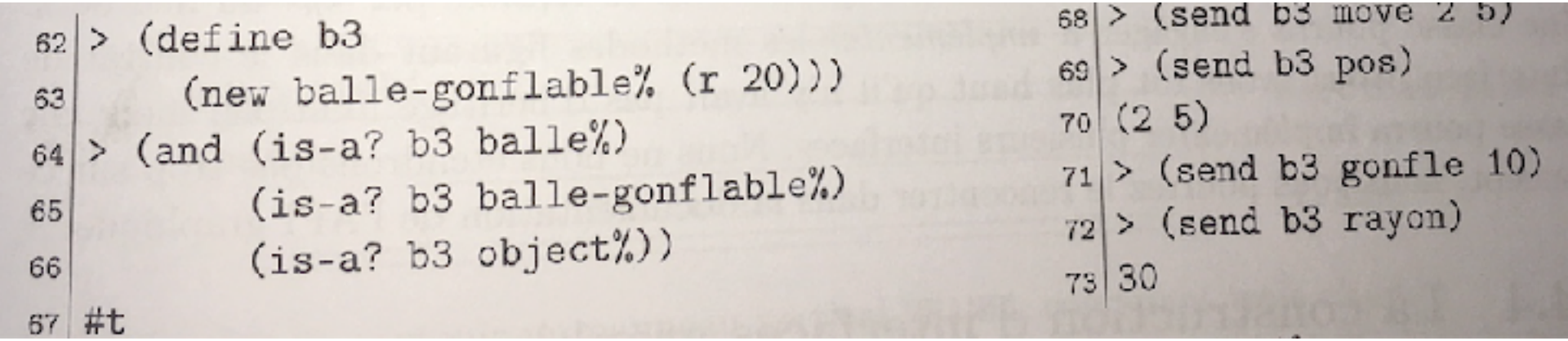
. Accepteur rayon

. Modificateur (méthode) gonfle.

Une balle gonflable étant une balle, on peut lui envoyer les méthodes de la classe balle%.

C'est la notion d'heritage. (en Racket comme en Java l'héritage est simple).

Racket et POO



```
> (define b3
    (new balle-gonflable% (r 20)))
> (and (is-a? b3 balle%)
       (is-a? b3 balle-gonflable%)
       (is-a? b3 object%))
#t
> (send b3 move 2 5)
> (send b3 pos)
(2 5)
> (send b3 gonfle 10)
> (send b3 rayon)
30
```

Racket et POO

Dans une sous-classe on peut aussi redéfinir des classes

de la classe mère :

. avec define/override.

Exemple : on veut par exemple ,défnir des balles

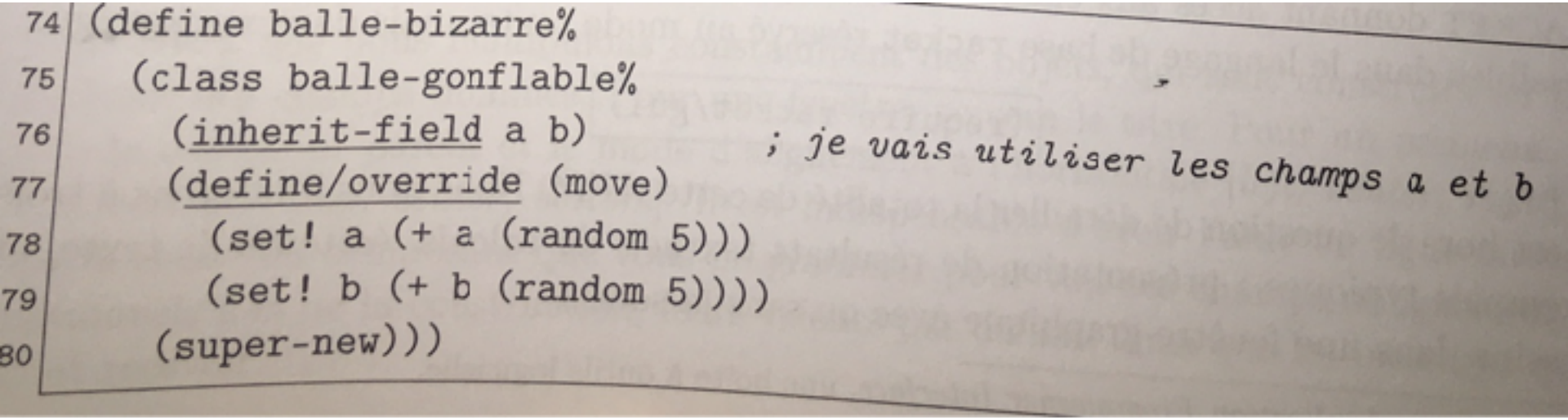
bizarres dont la méthode move sans arguments

provoque un petit déplacement aléatoire

2 facons

Racket et POO

Soit on modifie explicitement les champs a et b (il faut alors en demander l'héritage pour introduire ces nouvelles variables dans le texte de la sous-classe



```
74 (define balle-bizarre%
75   (class balle-gonflable%
76     (inherit-field a b)          ; je vais utiliser les champs a et b
77     (define/override (move)
78       (set! a (+ a (random 5)))
79       (set! b (+ b (random 5))))
80     (super-new)))
```

Racket et POO

Soit on utilise la fonction move de la classe mère mais il faut alors utiliser l'objet super.

L'objet super est l'objet courant comme this mais pour lequel la recherche de méthode se fait à partir de la

```
81 (define balle-bizarre%
82   (class balle-gonflable%
83     (define/override (move)
84       (super move
85         (random 5) (random 5)))
86     (super-new)))
```

```
87 > (define b3
88        (new balle-bizarre%))
89 > (send b3 pos)
90 (0 0)
91 > (send b3 move)
92 > (send b3 pos)
93 (3 1)
```

TP N°3