OPTION PF2 - Automne 2016
Jean-Paul Roy
L2-Info&Maths, Faculté des Sciences

http://deptinfo.unice.fr/~roy

Cours n°5

La Programmation par Objets

("soft" et "hard")

p. 313-318

Vers les objets...

- **Système logiciel** : un ensemble de mécanismes permettant certaines **actions** sur certaines **données**.

- La structure du système peut choisir de mettre l'accent :

  - sur les actions et/ou les calculs qui vont traiter les données :
    - soit par manipulation explicite de la mémoire via l'exécution d'**instructions** : **programmation impérative**.
    - soit de manière implicite en privilégiant l'évaluation d'expressions : **programmation fonctionnelle**.

  - sur les données et d'y encapsuler les calculs :
    - c'est le cas de la **programmation par objets**.

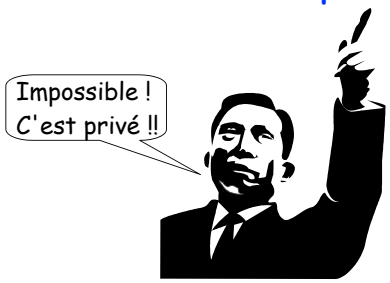
Peut-on ré-initialiser une mémoire privée ?

- Revenons au générateur d'entiers pairs :

```
> (define fermat (make-gpair))
> (for ([i (in-range 1 11)])
    (printf "~a " (fermat)))
0 2 4 6 8 10 12 14 16 18
```

- Comment puis-je remettre `fermat` à 0 **sans le redéfinir** ?

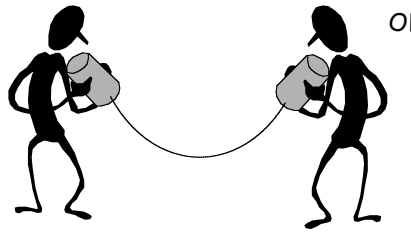
- Comment puis-je **modifier de force la mémoire privée d'une fermeture** ?



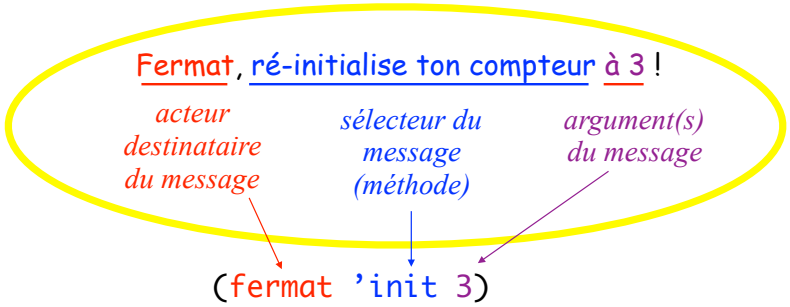
- Nous allons implémenter en Scheme pur l'idée d'envoi de message.

Les objets "softs" [Scheme pur]

Ô Fermat, ré-initialise ton compteur à 3 !



UN ENVOI DE MESSAGE AVEC ARGUMENT(S) :



Nos générateurs deviennent des objets en Scheme pur !

• Un **objet** sera une fonction prenant un argument obligatoire [la méthode] et des arguments optionnels [les arguments du message] :

```
(define (make-gpair)
  (let ((n -2))
    (define (this methode . Largs)
      (case methode
        ((reset) (set! n -2))
        ((init)  (set! n (car Largs)))
        ((next)  (set! n (+ n 2)) n))
        (else    (error "Méthode inconnue" methode))))
    this))    ; l'objet courant this (le self de Python)
```

Un objet est une fonction qui analyse un message !

```
> (define fermat (make-gpair))
> (fermat 'next)
0
> (fermat 'next)
2
> (fermat 'next)
4
```

```
> (fermat 'init 100)
> (fermat 'next)
102
> (fermat 'reset)
> (fermat 'next)
0
> (fermat 'next)
2
```


• Tout ce que nous avons fait au niveau "soft" est portable et ne dépend que du noyau Scheme pur. Mais la POO procure des mécanismes bien plus sophistiqués !

• Les grands initiateurs : SIMULA et SMALLTALK.

1967 1972 → *www.squeak.org*

• La vraie couche-objet de Racket permet de définir des **classes**, de construire des **objets** comme **instances de classes**, et d'envoyer des **messages** [méthodes d'instance] à des objets. Champs **publics** et **privés**.

• Tout ceci en symbiose avec le Scheme usuel. On dispose ainsi des trois paradigmes majeurs de programmation :

fonctionnel + impératif + objets

• Il s'agit simplement d'une librairie écrite en Scheme qui est chargée lors de la directive `#lang racket` dans un module.

Un exemple de classe avec la vraie couche-objet Racket

```
(define pile%                    ; la classe des piles
  (class object%                 ; une sous-classe de object%
    (define L '())               ; variable d'instance privée
    (define/public (vide?)       ; méthode publique
      (null? L))
    (define/public (sommet)      ; méthode publique
      (if (null? L) (error "Pile vide !") (car L)))
    (define/public (empiler x)   ; méthode publique
      (set! L (cons x L)))
    (define/public (depiler)     ; méthode publique
       (if (null? L) (error "Pile vide !") (set! L (cdr L))))
    (super-new)))                ; le constructeur de la classe mère
```

• Tout comme un module, la classe `pile%` exporte des noms publics. Par contre L est une variable privée pour chaque pile !

• Les **fonctions** exportées sont vues comme des **méthodes**.

• Il est nécessaire d'invoquer le constructeur de la classe-mère !

• L'envoi de message passe par la macro **send** :

`(fermat 'init 3)`

Objets "soft"

```
> (define P (new-pile))
> (P 'vide?)
#t
> (P 'empiler 3)
> (P 'empiler 5)
> (P 'vide?)
#f
> (P 'sommet)
5
> (P 'depiler)
> (P 'sommet)
3
```

`(send fermat init 3)`

Objets "hard"

```
> (define P (new pile%))
> (send P vide?)
#t
> (send P empiler 3)
> (send P empiler 5)
> (send P vide?)
#f
> (send P sommet)
5
> (send P depiler)
> (send P sommet)
3
```

- Exemple : des **points 2D**

```
(define nbPoints 0)      ; variable globale...
                                  (n'est pas attachée à
                                  un objet particulier !)
(define point2d%
  (class object%
    (init-field (x 0) (y 0))
    (define/public (pos)
      (list x y))
    (define/public (move dx dy)
      (set! x (+ x dx)) (set! y (+ y dy)))
    (define/public (toString)
      (concat-infos))
    (define (concat-infos)          ; private !
      (format "point2d[~a,~a]" x y))
    (set! nbPoints (add1 nbPoints))
    (super-new)))
```

initialisation de l'instance
[le constructeur en Python/Java]

- **Instanciation d'une classe** avec l'opérateur new. Il s'agit de construire un nouvel objet de la classe en initialisant ses champs :

```
(define a (new point2d% (x 10) (y 20)))
```

- Si l'on omet certains champs, ils seront initialisés par défaut :

```
(define b (new point2d% (y 20)))    ; ⇒ x = 0
```

- Si l'on omet les noms des champs, il faut fournir toutes les valeurs dans l'ordre [par exemple dans la classe pen% de l'API] :

```
(define c (make-object point2d% 10 20))
```

```
> nbPoints
3
> a
#<struct:object:point2d%...>
> (send a toString)
"point2d[10,20]"
```

```
> (object? b)
#t
> (is-a? c point2d%)
#t
> (is-a? c object%)
#t
```

- On peut envoyer à un objet de la classe `point2d` un message public :

```
> (send a pos)
(10 20)
> (send a move 2 5)
> (send a toString)
"point2d[12,25]"
```

- On peut aussi lui envoyer une ***cascade*** de messages publics :

```
> (send* a (move -1 0) (pos))
(11 25)
```

- Il est bien entendu <u>impossible</u> d'utiliser de l'extérieur de la classe une méthode *privée* [non déclarée publique] :

```
> (send a concat-infos)
```
ERROR : no method concat-infos for class point2d%

En Python 3 :

Comparez avec le code Scheme de la page 9...

```
class Point2d :       # Python 3
    nbPoints = 0
    def __init__(self,x=0,y=0) :
        self.x = x
        self.y = y
        Point2d.nbPoints++
    def getX(self) :                # inutile ? x est public
        return self.x
    def getY(self) :
        return self.y
    def move(self,dx,dy) :
        self.x = self.x + dx
        self.y = self.y + dy
    def __concat_infos(self) :      # private ?
        return 'point2d[{},{}]'.format(self.x,self.y)
    def __repr__(self) :
        return self.__concat_infos()
```

Exécution au toplevel de Python :

```
>>> p = Point2d(3,5)
>>> q = Point2d(100)
>>> (p.getX(),p.getY())
(3, 5)
>>> (q.getX(),q.getY())
(100, 0)
>>> q
point2d[100,0]
>>> nbPoints
NameError: name 'nbPoints' is not defined
>>> Point2d.nbPoints
2
>>> q.__concat_infos()
AttributeError: 'Point2d' object has no attribute '__concat_infos'
```

private ?

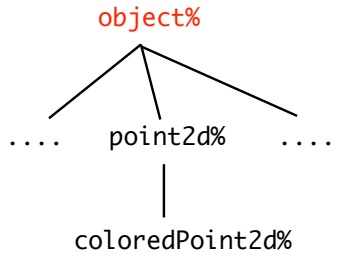
- En réalité, le coup du __ en préfixe de concat_infos est un leurre :

```
>>> dir(q)                              # oups !
['_Point2d__concat_infos', ...]
>>> p._Point2d__concat_infos()
point2d[3,5]
```


La hiérarchie des classes - L'héritage

```
(define point2d%
  (class object%
    ...))
```

- Toute classe est une **sous-classe** de object%, qui est donc à la racine de l'arbre des classes :




```
(define coloredPoint2d%
  (class point2d%
    ...))
```

La classe coloredPoint2d% *:*
- *étend la classe* point2d%
- *est une sous-classe de* point2d%
- *admet* point2d% *comme classe-mère*

Java ⋯⋯▸ `class ColoredPoint2d extends Point2d`
Python ⋯⋯▸ `class ColoredPoint2d(Point2d)`

- Une **sous-classe** peut ajouter de nouveaux champs et de nouvelles méthodes, donc créer des point2d *particuliers* :

```
(define colored-point2d%
  (class point2d%
    (init-field (color "black"))      ; un nouveau champ
    (define/public (getColor)         ; une nouvelle méthode
      color)
    (super-new)))
```

- L'**héritage** exprime qu'une instance de colored-point2d% pourra utiliser les méthodes de la classe-mère [voire de ses ascendants !] :

```
> (define P (new colored-point2d% (x 5) (color "red")))
> (send P pos)      ; la méthode pos est définie dans la classe-mère !
(5 0)
> (send P getColor)
"red"
> (and (is-a? P object%) (is-a? P point2d%)
       (is-a? P colored-point2d%))
#t
```

- Mais une sous-classe peut aussi modifier le comportement, en **redéfinissant** [*overriding*] certaines méthodes de sa classe-mère :

```
(define colored-point2d%
  (class point2d%
    (init-field (color "black"))
    (define/public (getColor)
      color)
    (define/override (toString)
      (string-append "colored-" (super toString) "-" color))
    (super-new)))
```

- Pour invoquer dans la sous-classe une méthode (foo a b) de la classe-mère, on demandera (super foo a b). Donc (super foo a b) équivaut à (foo a b) sauf que la recherche ascendante des méthodes se fait à partir de la classe-mère !

Java ⋯⋯▸ `super.foo(a,b)`
Python ⋯⋯▸ `super(ColoredPoint2d,self).foo(a,b)`

• Soit à rajouter dans `coloredPoint2d%` une méthode `state` retournant la liste de tous les champs :

```
> (send P state)
(5 0 "red")
```

• Le champ `color` est dans la classe mais `x` et `y` sont dans la classe mère, il faut explicitement en demander l'importation !

```
(define colored-point2d%
  (class point2d%
    (init-field (color "black"))
    (inherit-field x y)
    (define/public (getColor)
      color)
    (define/override (toString)
      (string-append "colored-" (super toString) "-" color))
    (define/public (state)
      (list x y color))      ; color ⇔ (send this getColor)
    (super-new)))
```

Comment Scheme pourrait-il deviner que x et y sont des champs et non des variables définies hors de la classe ?...

l'objet courant

• Si l'on veut exprimer que la classe `point2d%` **implémente l'interface** `forme<%>` :

Scheme ⋯⋯➤
```
(define point2d%
  (class* object% (forme<%>)
    ...))
```

Java ⋯⋯➤
```
class Point2d extends Object implements Forme {
  ...
}
```

• Une classe peut implémenter plusieurs interfaces !

```
(define A%
  (class* B% (I<%> ...)
    ...))
```

• Contrairement aux classes, on ne peut pas instancier une interface :

```
> (new forme<%>)
```
Expected argument of type class; given interface forme<%>

Pas d'héritage multiple, mais des interfaces...

• On pourrait vouloir hériter de plusieurs parents ! Ce n'est pas autorisé en Racket [*un peu en Python*] ! **Une seule classe-mère**...

• Pour y remédier, on introduit le concept d'**interface** [comme en Java]. Une **interface** est une classe *incomplète* dans laquelle les méthodes ne sont pas implémentées, seulement déclarées !

• Le nom d'une classe se termine par `%`. Le nom d'une interface se termine par `<%>`

```
(define forme<%>
  (interface () pos move))
```

aucune sur-interface

• Toute **classe implémentant l'interface** s'engage à remplir un **contrat** : concrétiser les méthodes `pos` et `move`. Par exemple les classes `point2d%`, `rectangle%`, `cercle%`, `polygone%`,... implémenteraient l'interface `forme<%>`.

EXEMPLE : la classe des **listes circulaires**

• Une liste circulaire sera une liste dotée d'une méthode `next` qui retourne l'élément courant, en avançant vers l'élément suivant. Mais à la fin de la liste, on revient automatiquement au début !

```
(define circ-list%
  (class object%
    (init-field (contenu empty))
    (define ptr contenu)
    (define/public (next)
      (let ((x (car ptr)))
        (if (null? (cdr ptr))
            (set! ptr contenu)
            (set! ptr (cdr ptr)))
        x))
    (super-new)))
```

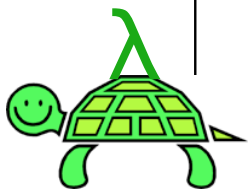
```
> (define LC (make-object circ-list% '(a b c)))

> (send LC next)
a
> (send LC next)
b
```

```
> (send LC next)
c
> (send LC next)
a
```

etc.

Les primitives (non-objet) de la Tortue (TP5)

graphisme polaire	graphisme cartésien
(forward d) (back d) (left a) (right a) (pen-up) (pen-down)	(init pos cap [reset? #f]) (heading) (set-heading a) (position) (set-position p) ; p = (x y) (xcor) (ycor) (toward p) ; p = (x y)



adt-turtle.rkt