cours8

November 3, 2020

# 1 Programmation fonctionnelle

## 1.1 Semaine 8 : Objets et labels

**Etienne Lozes - Université Nice Sophia Antipolis - 2019**

[117]:
```
let () = ()
```

---

---

## 1.2 Les labels

Un label est un nom que l'on donne à un paramètre d'une fonction.

Par exemple, je voudrais écrire un fonction `valid_date : int -> int -> bool` qui me dit si une date est valide (en supposant l'année bisextile).

Les deux arguments entiers sont le jour du mois et le numéro du mois.

Si je veux savoir si le 4 juillet est une date valide, j'écrirai sans doute `valid_date 4 7`.

Mon oncle d'Amérique, lui, écrira probablement `valid_date 7 4`.

On peut documenter la fonction pour éviter les erreurs, certes, mais avec les labels, on peut faire mieux!

[2]:
```
let valid_date ~day ~month =  (* <- notez bien le ~ devant day et month ! *)
  day > 0 &&
  month > 0 &&
  month < 13 &&
  day < [|0;31;28;31;30;31;30;31;31;30;31;30;31|].(month)
```

[2]:
```
val valid_date : day:int -> month:int -> bool = <fun>
```

Notez bien que le type de `valid_date` est maintenant `day:int -> month:int -> bool`.

Pour utiliser ma fonction, je dois rappeler les labels devant les arguments

[3]:
```
let _ = valid_date ~day:4 ~month:7
```

[3]:
```
- : bool = true
```

Comme ils sont nommés, je peux maintenant donner les arguments dans l'ordre que je veux, ou encore fixer un des deux arguments, au choix, dans une application partielle.

[4]:
```
let _ = valid_date ~month:7 ~day:4
```

[4]:
```
- : bool = true
```

[5]:
```
let check_day = valid_date ~month:1
```

[5]:
```
val check_day : day:int -> bool = <fun>
```

[6]:
```
let check_month = valid_date ~day:10
```

[6]:
```
val check_month : month:int -> bool = <fun>
```

Lorsque le nom du label (disons `bla`) est le même que le nom de l'argument (s'il s'agit d'une variable), on peut écrire simplement `~bla` au lieu de `~bla:bla`

[7]:
```
let day = 13
let _ = valid_date ~day ~month:12
```

[7]:
```
val day : int = 13
```

[7]:
```
- : bool = true
```

On peut aussi écrire des fonctions d'ordre supérieur qui prennent en argument des fonctions avec des labels

[8]:
```
let instanciate_day f day = f ~day

let foo = instanciate_day valid_date 4

let check_month month = instanciate_day valid_date 4 ~month
```

[8]:
```
val instanciate_day : (day:'a -> 'b) -> 'a -> 'b = <fun>
```

[8]:
```
val foo : month:int -> bool = <fun>
```

```
[8]: val check_month : int -> bool = <fun>
```

---

### 1.3 Valeur par défaut d'un argument labélisé

On peut fixer une valeur par défaut à un argument labélisé, et de ce fait le rendre optionnel.

- au moment de déclarer la fonction, l'argument est déclaré avec la syntaxe `?(nom_arg=valeur_defaut)`
- au moment d'appeler la fonction, si on veut changer la valeur par défaut, on écrit `~nom_arg:autre_valeur`

```
[9]: let string_of_date ?(year=2019) day month  =
       Format.sprintf "%02d/%02d/%d" day month year

     let _ = string_of_date 14 7

     let _ = string_of_date ~year:1789 14 7
```

```
[9]: val string_of_date : ?year:int -> int -> int -> string = <fun>
```

```
[9]: - : string = "14/07/2019"
```

```
[9]: - : string = "14/07/1789"
```

#### 1.3.1 Subtilité sur l'ordre des arguments

Il est recommandé de faire suivre un argument optionnel par un argument non optionnel. Cela permet d'identifier lors de l'appel de fonction si l'argument optionnel a bien été passé implicitement avec sa valeur par défaut

```
[85]: let string_of_date_bad day month ?(year=2019) = ""
```

```
File "[85]", line 1, characters 35-44:
1 | let string_of_date_bad day month ?(year=2019) = ""
                                       ^^^^^^^^^
Warning 16: this optional argument cannot be erased.
```

```
[85]: val string_of_date_bad : 'a -> 'b -> ?year:int -> string = <fun>
```

---

### 1.4 Paramètre optionnel sans valeur par défaut

On peut aussi ne pas définir de valeur par défaut pour un argument optionnel.

Dans ce cas-là, il est traité comme un argument de type `'a option` en interne.

[10]:
```
let string_of_date ?year day month =
   match year with
     | None   -> Format.sprintf "%02d/%02d" day month
     | Some y -> Format.sprintf "%02d/%02d/%d" day month y

let _ = string_of_date 14 7

let _ = string_of_date ~year:1789 14 7
```

[10]:
```
val string_of_date : ?year:int -> int -> int -> string = <fun>
```

[10]:
```
- : string = "14/07"
```

[10]:
```
- : string = "14/07/1789"
```

#### 1.4.1 Conclusion sur les labels

Les labels sont une alternative intéressante au **polymorphisme par surcharge** courant dans les langages objets.

En Java, je peux définir une méthode `valid_date(int day, int month)` et une autre méthode `valid_date(int day, int month, int year)` qui ont le même nom mais qui n'ont pas les mêmes types d'arguments, ni même nécessairement la même valeur de retour.

OCaml est plus restrictif avec les labels, mais il permet dans de nombreux cas de reproduire ce qu'on ferait en Java.

---

### 1.5 Qu'est-ce qu'un objet?

Un objet regroupe dans une même entité des **champs** qui contiennent des valeurs et des **méthodes** qui permettent de les manipuler.

Pour fixer les idées, nous allons considérer un objet `compteur` qui dispose de : - un champs `n:int` qui contient la valeur du compteur - une méthode `get_valeur:int` qui renvoie la valeur du compteur - une méthode `incr: int -> unit` qui incrémente le compteur d'une valeur donnée

[11]:
```
let compteur0 = object
  val mutable n = 0
  method  get_valeur = n
  method incr k = n <- n + k
```

```
end
```

[11]:
```
val compteur0 : < get_valeur : int; incr : int -> unit > = <obj>
```

Le type d'un objet indique quelles méthodes peuvent être appellées pour cet objet.

Pour appeler la méthode `m` d'un objet `o`, on écrit `o#m`.

[12]:
```
compteur0#get_valeur
```

[12]:
```
- : int = 0
```

[13]:
```
let () = compteur0#incr 1
```

[14]:
```
compteur0#get_valeur
```

[14]:
```
- : int = 1
```

Il n'est pas possible de lire directement la valeur d'un champs à l'extérieur de l'objet sans une méthode "accesseur" comme `get_valeur`.

[15]:
```
compteur0#n
```

```
File "[15]", line 1, characters 0-9:
1 | compteur0#n
    ^^^^^^^^^
Error: This expression has type < get_valeur : int; incr : int -> unit >
       It has no method n
```

Il n'est pas possible de modifier un champs dans une méthode s'il n'a pas été déclaré mutable

[16]:
```
let compteur0 = object
  val (* mutable *) n = 0
  method  get_valeur = n
  method incr k = n <- n +k
end
```

```
File "[16]", line 4, characters 18-27:
4 |   method incr k = n <- n +k
                      ^^^^^^^^^
Error: The instance variable n is not mutable
```

La forme `let ... = object ... end` permet de créer un **objet immédiat**. Cependant, en général, on préfère voir les objets comme des instances d'une classe…

---

### 1.6 Qu'est-ce qu'une classe?

Une classe est une "fabrique" à objets. À partir d'une classe `compteur`, je vais pouvoir créer plusieurs objets: ils auront en commun les méthodes `get_valeur` et `incr`, mais chaque objet aura son propre champs `n`.

[17]:
```
class compteur = object
   val mutable n = 0
   method get_valeur = n
   method incr k = n <- n + k
end
```

[17]:
```
class compteur :
  object
    val mutable n : int
    method get_valeur : int
    method incr : int -> unit
  end
```

Pour créer un objet `o` **instance de** de la classe `c`, j'écris `let o = new c`.

[18]:
```
let compteur1 = new compteur
let compteur2 = new compteur
```

[18]:
```
val compteur1 : compteur = <obj>
```

[18]:
```
val compteur2 : compteur = <obj>
```

[19]:
```
compteur1#incr 1
```

[19]:
```
- : unit = ()
```

[20]:
```
compteur1#get_valeur
```

[20]:
```
- : int = 1
```

[21]:
```
compteur2#get_valeur
```

```
[21]: - : int = 0
```

La classe peut prendre des arguments afin d'initialiser correctement les champs de l'objet. Par exemple, la classe compteur peut prendre en argument la valeur initiale du compteur.

```
[22]: class compteur n_init = object
         val mutable n = n_init
         method get_valeur = n
         method incr k = n <- n + k
      end
```

```
[22]: class compteur :
        int ->
        object
          val mutable n : int
          method get_valeur : int
          method incr : int -> unit
        end
```

```
[23]: let compteur3 = new compteur 42
```

```
[23]: val compteur3 : compteur = <obj>
```

```
[24]: compteur3#get_valeur
```

```
[24]: - : int = 42
```

---

### 1.7 S'appeler soi-même

Une méthode d'un objet peut en appeler une autre sur le même objet. Cet objet n'est pas connu au moment où on écrit la méthode: un nom spécial (en général `self`) est utilisé pour en parler.

```
[25]: class compteur n_init = object (self)
         val mutable n = n_init
         method get_valeur = n
         method incr k =
            n <- self#get_valeur + k  (* <- self# est nécessaire *)
      end
```

```
[25]: class compteur :
        int ->
        object
```

```
  val mutable n : int
  method get_valeur : int
  method incr : int -> unit
end
```

Note : le nom `self` n'est pas un mot-clé, on choisit comment s'appeler soi-même juste après le mot-clé `object`. Parfois on préfère s'appeler `this`.

[26]:
```
class fibo_naif = object (this)
   method fibo n =
      if n < 2
      then 1
      else this#fibo (n-1) + this#fibo (n-2)
end
```

[26]:
```
class fibo_naif : object method fibo : int -> int end
```

---

### 1.8 L'héritage

On peut définir une classe `c` en étendant une autre classe `d`.

On dit que `c` **hérite** de `d`.

[27]:
```
class compteur_identifie n_init (id_init:string) = object
   inherit compteur n_init
   val id = id_init
   method affiche = Format.sprintf "%s vaut %d" id n
end
```

[27]:
```
class compteur_identifie :
  int ->
  string ->
  object
    val id : string
    val mutable n : int
    method affiche : string
    method get_valeur : int
    method incr : int -> unit
  end
```

[28]:
```
(* definition alternative avec self *)

class compteur_identifie n_init (id_init:string) = object (self)
   inherit compteur n_init
```

```
  val id = id_init
  method affiche = Format.sprintf "%s vaut %d" id self#get_valeur
end
```

[28]:
```
class compteur_identifie :
  int ->
  string ->
  object
    val id : string
    val mutable n : int
    method affiche : string
    method get_valeur : int
    method incr : int -> unit
  end
```

[29]:
```
let compteur4 = new compteur_identifie 42 "toto"
```

[29]:
```
val compteur4 : compteur_identifie = <obj>
```

[30]:
```
compteur4#affiche
```

[30]:
```
- : string = "toto vaut 42"
```

---

### 1.9 Redéfinition de méthode

En cas d'héritage, je peux redéfinir une méthode `m` plutôt que d'en hériter sans la changer.

Ceci peut avoir un effet de bord: changer `m` peut aussi changer le comportement d'une méthode qui appelle la méthode `m`.

[31]:
```
class chenille = object (self)
  method nom = "chenille"
  method affiche = self#nom
end
class papillon = object
  inherit chenille
  method! nom = "papillon"    (* <- redéfinition *)
end
```

[31]:
```
class chenille : object method affiche : string method nom : string end
```

```
[31]: class papillon : object method affiche : string method nom : string end
```

Note: on peut utiliser `method!` au lieu de `method` en cas de redéfinition, mais ce n'est pas obligé (c'est le `@Override` de Java).

```
[32]: let papillon0 = new papillon
```

```
[32]: val papillon0 : papillon = <obj>
```

```
[33]: papillon0#nom
```

```
[33]: - : string = "papillon"
```

```
[34]: papillon0#affiche
```

```
[34]: - : string = "papillon"
```

Il faut bien comprendre ce qu'il se passe: - avec `papillon0#nom`, c'est la méthode redéfinie qui est appelée - avec `papillon0#affiche`, c'est la méthode `affiche` de la classe `chenille` qui est appelée, **mais** l'appel `self#nom` que contient `affiche` est évalué en remplaçant `self` par `papillon0`: c'est donc bien `papillon0#nom` qui est appelé.

On parle de **liaison tardive** ou de liaison dynamique.

**Et la liaison statique**? C'est la liaison "habituelle": quand une fonction en appele une autre, on sait à l'avance quelle est cette autre fonction, il ne sera pas possible de la remplacer par une autre.

```
[35]: let nom () = "chenille"
      let affiche () = Format.sprintf "%s vaut 0" (nom ())
      let nom () = "papillon"
```

```
[35]: val nom : unit -> string = <fun>
```

```
[35]: val affiche : unit -> string = <fun>
```

```
[35]: val nom : unit -> string = <fun>
```

```
[36]: let _ = affiche ()
```

```
[36]: - : string = "chenille vaut 0"
```

## 1.10 Invoquer une méthode d'une classe ancêtre

Lorsqu'on redéfinit une méthode, il est parfois utile de pouvoir appeler la version antérieure de la méthode. Il suffit de donner un nom (en général `super`) à la classe dont on hérite

[37]:
```
class papillon = object (self)
   inherit chenille as super
   method nom =
      Format.sprintf "papillon (anciennement %s)" super#nom
end
```

[37]:
```
class papillon : object method affiche : string method nom : string end
```

[38]:
```
(new papillon)#nom
```

[38]:
```
- : string = "papillon (anciennement chenille)"
```

## 1.11 Autre exemple : la mémoïsation

Revenons sur la classe `fibo` de tout à l'heure

[39]:
```
class fibo_naif = object (self)
   method fibo n =
      if n < 2
      then 1
      else self#fibo(n-1) + self#fibo(n-2)
end
```

[39]:
```
class fibo_naif : object method fibo : int -> int end
```

[40]:
```
let fibo0 = new fibo_naif
```

[40]:
```
val fibo0 : fibo_naif = <obj>
```

[41]:
```
List.init 10 (fun n -> fibo0#fibo n)
```

[41]:
```
- : int list = [1; 1; 2; 3; 5; 8; 13; 21; 34; 55]
```

```
fibo0#fibo 50
```

Ce programme ne termine pas (dans un temps raisonable). Vous vous souvenez pourquoi?

Une solution pour éviter de répéter des calculs déjà effectués consiste à les mémoriser à l'aide d'une table de hachage.

Nous allons mémoïser la méthode/fonction `fibo` de deux façons différentes:

1. sans passer par les objets, de la manière "classique": au moment de définir la fonction fibo, on prend en compte la mémoire
2. avec les objets: on écrit une première version "non mémoïsée", puis on la mémoise.

### 1.11.1 L'approche classique

1. Je crée une table de hachage pour contenir les valeurs de `fibo` déjà calculées. Au moment de créer la table de hachage, `OCaml` a besoin d'une estimation de la taille de ma table de hachage. Pour nous, ce sera 50.

[86]:
```
let memoire = Hashtbl.create 50
```

[86]:
```
val memoire : ('_weak3, '_weak4) Hashtbl.t = <abstr>
```

2. J'écris ma fonction mémoïsée `fibo`

[43]:
```
let rec fibo n =
   try Hashtbl.find memoire n (* si j'ai deja calcule fibo n *)
   with Not_found ->
     let res =                (* sinon, je le calcule *)
       if n < 2
       then 1
       else fibo(n-1) + fibo(n-2)
    in
    Hashtbl.add memoire n res;   (* je memorise fibo n *)
    res                          (* je renvoie le résultat *)
```

[43]:
```
val fibo : int -> int = <fun>
```

[44]:
```
fibo 50
```

[44]:
```
- : int = 20365011074
```

### 1.11.2 L'approche par redéfinition

Maintenant, je vais définir une classe `fibo_memo` qui contient une méthode `fibo` mémoïsée en deux étapes:

1. je définis une classe `fibo_naif` qui contient une méthode `fibo` non mémoïsée
2. je définis une classe `fibo_memo` qui hérite de `fibo_naif` et qui redéfinit la méthode `fibo` de la façon suivante:

- si `fibo n` est en mémoire, on renvoie le résultat directement
- sinon on appele la méthode `fibo` de la classe `fibo_naif`.

[45]:
```
class fibo_naif = object (self)
   method fibo n =
      if n < 2
      then 1
      else self#fibo (n-1) + self#fibo (n-2)
end


class fibo_memo =
   object (self)
   inherit fibo_naif as super

   val memoire = Hashtbl.create 50

   method fibo n =
      try Hashtbl.find memoire n
      with Not_found ->
         let res = super#fibo n in
         Hashtbl.add memoire n res;
         res
end
```

[45]:
```
class fibo_naif : object method fibo : int -> int end
```

[45]:
```
class fibo_memo :
  object val memoire : (int, int) Hashtbl.t method fibo : int -> int end
```

[46]:
```
let fibo1 = new fibo_memo
let _ = fibo1#fibo 50
```

[46]:
```
val fibo1 : fibo_memo = <obj>
```

[46]:
```
- : int = 20365011074
```

---

## 1.12 L'héritage multiple

C'est l'un des aspects sur lesquels les langages objets mainstream ont des positions différentes: *Java* n'autorise pas l'héritage multiple, mais *C++* si.

En *OCaml*, l'héritage multiple est autorisé, mais l'ordre dans lequel on liste les classes dont on hérite joue un rôle.

[47]:
```
class a = object method ma = "a" end
class b = object method mb = "b" end
class ab = object inherit a inherit b end
```

```
[47]: class a : object method ma : string end


[47]: class b : object method mb : string end


[47]: class ab : object method ma : string method mb : string end
```

[48]:
```
let o = new ab in (o#ma ^ o#mb)
```

```
[48]: - : string = "ab"
```

### 1.12.1 Le "problème du diamant"

[49]:
```
class a = object method m = "a" end
class b = object inherit a method! m = "b" end
class c = object inherit a method! m = "c" end
class d = object (self) inherit b inherit c end
```

```
[49]: class a : object method m : string end


[49]: class b : object method m : string end


[49]: class c : object method m : string end


[49]: class d : object method m : string end
```

[50]:
```
(new d)#m
```

```
[50]: - : string = "c"
```

---

### 1.13 Clonage

Il est parfois utile de faire une copie d'un objet avant de le modifier, pour ne pas perdre l'état dans lequel il était. La façon la plus simple est d'utiliser la fonction `Oo.copy`.

[76]:
```
class compteur = object
   val mutable n = 0
   method inc = n <- n + 1
   method get = n
end
```

[76]:
```
class compteur :
  object val mutable n : int method get : int method inc : unit end
```

[77]:
```
let c1 = new compteur
let () = c1#inc
let c2 = Oo.copy c1
let () = c1#inc
```

[77]:
```
val c1 : compteur = <obj>
```

[77]:
```
val c2 : compteur = <obj>
```

[78]:
```
c2#get
```

[78]:
```
- : int = 1
```

#### 1.13.1 Clonage avec {<...>}

À l'intérieur d'une méthode, on peut utiliser la syntaxe `{<...>}` pour créer une copy de `self`. Cette syntaxe permet de redéfinir des champs ou des méthodes dans la copie.

[1]:
```
class compteur = object (self)
   val mutable n = 0
   method get = n
   method increment = n <- n + 1
   method copy = {< >} (*equivaut à Oo.copy self *)
   method incremented = {< n = self#get + 1 >}
end
```

[1]:
```
class compteur :
  object ('a)
    val mutable n : int
    method copy : 'a
```

```
  method get : int
  method increment : unit
  method incremented : 'a
end
```

**Rappel** la notation `{< n = ... >}` permettant de redéfinir la valeur du champs `n` de l'objet dans la copie est proche de la notation `{ r with n = ...}` que nous avons vue dans le cours sur les enregistrements.

Vous vous souvenez de ce que fait `{r with n=...}`?

[2]:
```
let c1 = new compteur
let () = c1#increment
let c2 = c1#copy
let () = c1#increment
```

[2]:
```
val c1 : compteur = <obj>
```

[2]:
```
val c2 : compteur = <obj>
```

[3]:
```
c2#get, c1#get
```

[3]:
```
- : int * int = (1, 2)
```

[4]:
```
let c3 = c1#incremented
```

[4]:
```
val c3 : compteur = <obj>
```

[58]:
```
c3#get, c1#get
```

[58]:
```
- : int * int = (3, 2)
```

### 1.13.2 Différence entre clonage et allocation avec `new`

Il y a une différence assez subtile entre cloner un objet et en allouer un nouveau en recopiant les champs.

[59]:
```
class compteur n_init = object (self)
   val mutable n = n_init
   method get = n
   method inc = n <- n + 1
   method copy = new compteur self#get  (* differe de Oo.copy self *)
```

```
end
```

[59]:
```
class compteur :
  int ->
  object
    val mutable n : int
    method copy : compteur
    method get : int
    method inc : unit
  end
```

Cette différence se voit lorsqu'on étend la classe:

- le clonage crée une copie de la même classe que l'objet
- `new` spécifie la classe (ici `compteur`): on risque de "sous-classer" l'objet que l'on clone

Précisons sur un exemple.

[60]:
```
class entier (n_init:int) = object (self)
   val mutable n = n_init
   method get = n
   method copy = new entier self#get (* <- entier !*)
end

class compteur n = object
   inherit entier n
   method inc = n <- n + 1
end
```

[60]:
```
class entier :
  int -> object val mutable n : int method copy : entier method get : int end
```

[60]:
```
class compteur :
  int ->
  object
    val mutable n : int
    method copy : entier
    method get : int
    method inc : unit
  end
```

[61]:
```
let c1 = new compteur 1
let c2 = c1#copy
```

```
[61]: val c1 : compteur = <obj>
```

```
[61]: val c2 : entier = <obj>
```

```
[62]: c2#inc
```

```
File "[62]", line 1, characters 0-2:
1 | c2#inc
    ^^
Error: This expression has type entier
       It has no method inc
```

---

## 1.14 Exemple avancé : un peuple de compteurs réinitialisables.

Je veux définir une classe compteur comme celle vue jusqu'à présent, et je souhaite créer plusieurs compteurs par la suite.

Je voudrais cependant ajouter la possibilité de remettre à 0 tous les compteurs que j'ai déjà créés en appelant une méthode `reset_all`.

Comment faire?

Analysons le problème

1. Il va me falloir maintenir une liste du `peuple` des compteurs. Cette liste ne peut pas être un champs de compteur, elle doit être commune à tous les compteurs: c'est une **variable de classe**. Il faudra la mettre à jour à chaque création de compteur.
2. La méthode `reset_all` ne pourra modifier directement que la valeur du compteur sur lequel elle est appellée. Il faudra donc ajouter une méthode `reset` pour remettre à 0 *un* compteur, et appeler cette méthode sur chaque compteur de la liste depuis `reset_all`.

### 1.14.1 Typage des objets

Le typage des objets est délicat en OCaml. On ne fera qu'effleurer le sujet…

La première chose à comprendre, c'est que deux classes différentes définissent deux types différents ***incompatibles***, même si l'une hérite de l'autre.

Ceci a une conséquence directe sur la possibilité de ranger dans une même liste des objets de classes différentes.

```
[79]: class class1 = object method m1 = "m1" end
      class class2 = object inherit class1 method m2 ="m2" end
      let c1 = new class1
      let c2 = new class2
```

```
[79]: class class1 : object method m1 : string end

[79]: class class2 : object method m1 : string method m2 : string end

[79]: val c1 : class1 = <obj>

[79]: val c2 : class2 = <obj>
```

```
[80]: c1::c2::[]
```

```
File "[80]", line 1, characters 4-6:
1 | c1::c2::[]
        ^^
Error: This expression has type class2 but an expression was expected of type
         class1
       The second object type has no method m2
```

La solution consiste à "sous-classer" l'objet `c2` pour pouvoir le mettre dans la même liste que `c1`. Pour cela, on utilise une **coercion** de la forme `(e :> sous_classe)` (ou parfois `(e:classe:>sous_classe)`)

```
[81]: let c2_1 = (c2:>class1)

[81]: val c2_1 : class1 = <obj>
```

```
[82]: c2_1#m1

[82]: - : string = "m1"
```

```
[83]: c2_1#m2
```

```
File "[83]", line 1, characters 0-4:
1 | c2_1#m2
    ^^^^
Error: This expression has type class1
       It has no method m2
```

La solution pour mettre `c1` et `c2` ensemble dans une même liste consiste donc à sous-classer `c2` à l'aide d'une coercion.

[1]:
```
class class1 = object method m = "m de c1" end
class class2 = object inherit class1 method! m = "m de c2" method n = "n" end
let c1, c2 = new class1, new class2
```

[1]:
```
class class1 : object method m : string end


class class2 : object method m : string method n : string end


val c1 : class1 = <obj>
val c2 : class2 = <obj>
```

[2]:
```
let l = c1 :: (c2 :> class1) :: []
```

[2]:
```
val l : class1 list = [<obj>; <obj>]
```

[3]:
```
List.map (fun c-> c#m) l
```

[3]:
```
- : string list = ["m de c1"; "m de c2"]
```

**Remarque** une coercion s'apparente à un *cast* en C. Mais contrairement au cast, une coercion ne casse pas le système de typage.

Revenons à notre problème: un peuple de compteurs réinitialisables.

[102]:
```
module Compteur_Non_Signe = struct
   class type resetable = object method reset : unit end

  let peuple : resetable list ref = ref []
  let naissance c = peuple := (c :> resetable) :: !peuple

  class compteur = object (self)
     initializer naissance self  (* <- initialiseur *)
     val mutable n = 0
     method get = n
     method inc = n <- n + 1
     method reset = n <- 0
     method reset_all = List.iter (fun c-> c#reset) !peuple
     method copy = let res = {<>} in naissance res; res
  end
end
```

[102]:
```
module Compteur_Non_Signe :
  sig
    class type resetable = object method reset : unit end
    val peuple : resetable list ref
    val naissance : #resetable -> unit
    class compteur :
      object ('a)
        val mutable n : int
        method copy : 'a
        method get : int
        method inc : unit
        method reset : unit
        method reset_all : unit
      end
  end
```

**Remarque** L'initialiseur est appelé à chaque création d'objet par `new`. Il n'est pas appelé par clonage, c'est pourquoi on privilégie une méthode `copy` qui répète l'initialiseur.

Pour finir, je peux signer mon module afin de rendre privé l'interface (class type) `resetable` et la variable `peuple`.

[108]:
```
module type COMPTEUR = sig
  class compteur : object ('a)
      method get : int
      method inc : unit
      method reset : unit
      method reset_all : unit
      method copy : 'a
   end
end
```

[108]:
```
module type COMPTEUR =
  sig
    class compteur :
      object ('a)
        method copy : 'a
        method get : int
        method inc : unit
        method reset : unit
        method reset_all : unit
      end
  end
```

[109]:
```
module Compteur : COMPTEUR = Compteur_Non_Signe
```

```
[109]: module Compteur : COMPTEUR
```

```
[114]: let c1,c2 = Compteur.(new compteur, new compteur)
       let c3 = c1#copy
       let () = c1#inc; c2#inc; c2#inc; c3#inc;c3#inc; c3#inc
       let _ = c1#get, c2#get, c3#get
```

```
[114]: val c1 : Compteur.compteur = <obj>
       val c2 : Compteur.compteur = <obj>
```

```
[114]: val c3 : Compteur.compteur = <obj>
```

```
[114]: - : int * int * int = (1, 2, 3)
```

```
[115]: let () = c1#reset_all
       let _ = c1#get, c2#get, c3#get
```

```
[115]: - : int * int * int = (0, 0, 0)
```

```
[111]: Compteur.peuple
```

```
File "[111]", line 1, characters 0-15:
1 | Compteur.peuple
    ^^^^^^^^^^^^^^^
Error: Unbound value Compteur.peuple
```

### 1.15 Pour conclure

Nous avons vu beaucoup de choses en une seule séance, et pourtant nous n'avons fait qu'effleurer le sujet… Résumons.

- une classe qui hérite d'une autre classe peut y ajouter des méthodes ou en redéfinir (`method!`)
- la liaison tardive permet de changer le comportement d'une méthode `m` héritée sans la redéfinir, mais en redéfinissant d'autres méthodes que la méthode `m` appelle
- cloner un objet avec `Oo.copy` ou `{<>}` n'est pas la même chose qu'en allouer un autre avec `new`
- le typage des objets nécessite parfois de faire des coercions `(o :> sous_classe)`

Rendez-vous au cours de POO de L3 pour revoir ces notions et pour en découvrir bien d'autres (classes virtuelles, méthodes privées, etc)