Classes et Objets en Ocaml.

Didier Rémy
2001 - 2002

http://cristal.inria.fr/~remy/mot/2/
http://www.enseignement.polytechnique.fr/profs/informatique/Didier.Remy/mot/2/

Slide 1

Cours	**Exercices**
1. Objets 2. Classes 3. Héritage 4. Typage 5. Abréviations 6. Sous-typage 7. Classes paramétrées 8. Méthodes virtuelles 9. Objets fonctionnels 10. Méthodes binaires	1. Classes 2. Les piles en style objet 3. Les chaînes en style objet 4. Variations sur la sauvegarde 5. Renommage

Avertissement

Ces notes introduisent informellement la couche objet du langage Caml. Le but n'est pas ici d'expliquer ce que sont les objets ou les classes, mais d'apprendre à les utiliser d'abord de façon intuitive dans des situations simples.

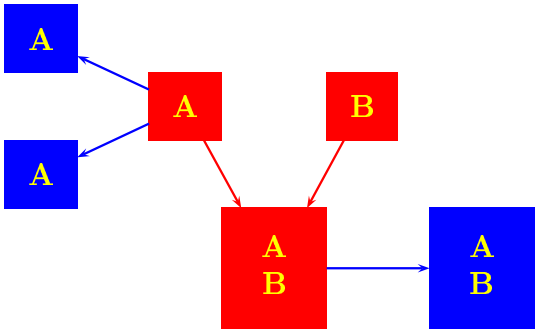
Slide 2

Nous présenterons également des exemples plus avancées d'utilisation les objets et des classes et nous expliquerons plus formellement le concept d'objet dans les cours suivants.

Objets et classes

Les classes (en rouge) sont des modèles d'objets, extensibles (les flèchent rouge représente l'héritage).

Slide 3



Les objets (en bleu) sont des instances (flèches bleu) des classes, rigides. Ce sont des valeurs qui peuvent être arguments ou résultats.

Les objets

Un objet est un enregistrement qui regroupe deux sortes de champs :
- Les **variables d'instance**, éventuellement mutables, forment l'état interne de l'objet ; ce sont des valeurs qui ne sont accessibles que par les méthodes.
- Les **méthodes** sont des fonctions qui ont accès en plus de leurs arguments aux variables d'instance et à l'objet lui-même.

Slide 4

Les objets sont construits à partir des classes.

La seule opération possible sur les objets depuis l'extérieur (*i.e.* en dehors de la classe qui le définit) est l'appel de méthode aussi appelé envoi de message.

objet#méthode [arguments]

Exemple : un compte en banque

État interne
- Solde du compte,
- Opérations récentes,
- Découvert autorisé,

Méthodes
- Consulter le solde,
- Déposer de l'argent,
- Retirer de l'argent,
- Imprimer un relevé.

Slide 5

Exemple de messages envoyés à un objet compte `c` :

```
c # dépôt (100.0)
c # retrait (50.0)
c # relevé
```

Les classes

Un mini langage pour abstraire les classes par rapport à des arguments (valeurs du langage de base ou objets) ou bien les appliquer à des arguments.

Les définitions de classes sont restreintes au toplevel.

Slide 6

Une classe totalement appliquée est un modèle d'objet (une sorte de moule) défini par :
- un ensemble de variables d'instances
- un ensemble de méthodes.

Au besoin, une classe est abstraite, explicitement, par rapport à l'objet (souvent mais pas obligatoirement appelé self) qui sera dans le moule au moment de l'exécution des méthodes.

On crée une instance du modèle par `new` *classe*

Exemples

La classe vide

```
class vide = object end;;
let objet_vide = new vide;;
let un_autre_objet_vide = new vide;;
```

Égalité sur les objets

Slide 7

Différents objets d'une même classe sont toujours différents (chaque objet port un champ invisible, son identité, différent des autres)

```
un_autre_objet_vide = objet_vide;;
```

```
- : bool = false
```

```
objet_vide = objet_vide;;
```

```
- : bool = true
```

```
class élève son_nom = object
  val nom : string = son_nom
```

```
  val mutable moyenne = 0.
  val mutable nombre_de_notes = 0
  method note x =
    let n = float nombre_de_notes in
    moyenne <- (x +. moyenne *. n) /. (n +. 1.);
    nombre_de_notes <- nombre_de_notes + 1; moyenne
  method bon_élève = moyenne > 14.
end;;
```

Slide 8

Exercice

Exercice 1 (Exemple (*)) *Transformer un petit programme simple, monomorphe, sans classe en un programme avec une seule classe la plus englobante possible. Y a-t-il un intérêt à cette transformation ?* Réponse

Slide 9

Essayer de transformer une librairie de fonctions polymorphes, par exemple sur les listes. Quel est le problème rencontré ? Réponse □

Syntaxe des classes (simplifiée)

Slide 10

Définition de classe

`class` [`virtual`] [`[` Type-var* `]`] Nom [Variable*] `=` Classe

Expression de classe

Nom
`fun` Variable* `->` Classe
Classe Expression*
`object` [`(` Variable [`:` Type] `)`] Corps `end`

Corps de classe

[Type*] `inherit` Classe [`as` Variable]
`val` [`mutable`] Variable `=` Expression
`method` [`virtual`] [`private`] Variable `=` Expression
`initializer` Expression

Nouvelles expressions du langage de base

Variable `<-` Expression
Expression `#` Variable
`new` Classe

Héritage : compte bancaire

Slide 11

```
class compte =
  object
    val mutable solde = 0.0
    method solde = solde
    method dépôt x = solde <- solde +. x
    method retrait x =
      if x <= solde then (solde <- solde -. x; x) else 0.0
  end;;
let c = new compte in c # dépôt 100.0; c # retrait 50.0;;
```

Envoi de message à soi-même.

```
class compte_avec_intérêts =
  object (self)
    inherit compte
    method intérêts = self # dépôt (0.03 *. self # solde)
  end;;
```

Méthode privée

Slide 12

Il peut être utile de ne pas donner la méthode `intérêts` à l'utilisateur, mais seulement de la rendre visible dans les sous-classes en vue d'une utilisation ultérieure.

On en fait une méthode privée :

```
class compte_avec_intérêts =
  object (self)
    inherit compte
    method private intérêts =
      self # dépôt (0.03 *. self # solde)
  end;;
```

Les méthodes privées ne sont pas accessibles de l'extérieur

```
(new compte_avec_intérêts) # intérêts;;
This expression has type compte_avec_intérêts
It has no method intérêts
```

Liaison tardive

Slide 13

Un appel récursif à une méthode prend en compte sa redéfinition ultérieure ; c'est le mécanisme de la liaison tardive.

Par exemple si on corrige le comportement de la méthode dépôt pour éviter une usage abusif :

```
class bon_compte =
  object
    inherit compte_avec_intérêts
    method dépôt x = if x > 0.0 then solde <- solde +. x
  end;;
```

La méthode `intérêts` appellera la nouvelle définition de la méthode `dépôt`.

Raffinement d'une méthode

Slide 14

Lors de l'héritage, on peut lier la vue de l'objet dans la classe parente pour appeler les anciennes méthodes.

Ainsi, on aurait pu définir :

```
class bon_compte =
  object
    inherit compte_avec_intérêts as super
    method dépôt x =
      if x > 0.0 then super # dépôt x
      else raise (Invalid_argument "dépôt")
  end;;
```

Ce qui n'oblige pas à connaître l'implémentation actuelle de la méthode `dépôt`

Compte avec relevé

Slide 15

Mélange des aspects précédents dans un compte avec relevé

```
type opération = Dépôt of float | Retrait of float

class compte_avec_relevé =
  object (self)
    inherit bon_compte as super
    val mutable registre = []
    method private trace x = registre <- x::registre
    method dépôt x =
      self#trace (Dépôt x); super # dépôt x
    method retrait x =
      self#trace (Retrait x); super # retrait x
    method relevé = List.rev registre
  end;;
```

Initialisation des objets

Slide 16

On peut abstraire une classe par rapport à des valeurs initiales.

Le mieux aurait été de le prévoir dès le départ. On peut aussi utiliser la clause `initializer` pour rattraper le coup.

Paramétrisation a priori

```
class compte x =
 object
   val mutable solde = x
   method ...
   ...
 end
```

Paramétrisation a posteriori

```
class promotion x =
 object (self)
   inherit compte_avec_relevé
   initializer solde <- x
 end;;
```

Les clauses `initializer` sont exécutées immédiatement après la création de l'objet, dans l'ordre de définition. Elles ont accès aux variables d'instance et à self.

Plus souvent utilisées pour assurer des invariants.

Clause d'initialisation

Slide 17

On peut aussi effectuer un dépôt automatique après la création de l'objet (ce qui correspond à l'idée de promotion).

```
class promotion x =
  object (self)
    inherit compte_avec_relevé
    initializer self # dépôt x
  end;;
```

Cette version effectue l'initialisation comme un dépôt, ce qui empêche automatiquement l'initialisation avec une valeur négative.

```
let ccp = new promotion 100.0 in
ignore (ccp # retrait 50.0); ccp # relevé;;
```

− : opération list = [Dépôt 100; Retrait 50]

Types des objets

Slide 18

Les types des objets sont de petits enregistrements de types :

Le type d'un compte bancaire :

```
type compte_bancaire =
  < solde : float;
    dépôt : float -> unit;
    retrait : float -> float;
    relevé : opération list >;;
(new promotion 50.0 : compte_bancaire);;
(new compte : compte_bancaire);;
Only the second object type has a method relevé
```

Le type `compte_bancaire` est dit fermé (entièrement spécifié)

Types des objets ouverts

Slide 19

Le type d'un objet peut être partiellement spécifié.

```
let fermeture c = c # retrait (c # solde);;
```

val fermeture : < retrait : 'a -> 'b; solde : 'a; .. > -> 'b = <fun>

Cette fonction, polymorphe, peut s'appliquer à toutes sortes de comptes.
Les `..` se comporte comme une variable de type anonyme.

```
fermeture (new compte);;
fermeture (new promotion 100.0);;
```

Le partage est exprimé et conservé par la construction `as`

```
let dépôt c x = if x > 100.0 then c # dépôt x; c;;
```

val dépôt : (< dépôt : float -> unit; .. > as 'a) -> float -> 'a = <fun>

```
(dépôt (new compte), dépôt (new compte_avec_relevé));;
```

*– : (float -> compte) * (float -> compte_avec_relevé) = <fun>, <fun>*

Types des classes

Les types des classes sont des petites spécifications qui décrivent la structure des classes.

Dans les classes, le type de `self` est toujours ouvert (partiellement spécifié). Il est compatible avec l'ensemble des méthodes couramment définies.

Slide 20

Syntaxe des types des classes

Type-de-définition-de-classe

`class` [`virtual`] [`[` Type-var* `]`] Nom [Type*] `:` T

Définition-de-type-de-classe

`class type` [`virtual`] [`[` Type-var* `]`] Nom [Type*] `=` T

T (Type de classe)

Slide 21

Nom
Type `->` T
`object` [`(` Type `)`] Type-du-corps `end`

Type-du-corps

`val` [`mutable`] Variable `:` Type
`method` [`virtual`] [`private`] Variable `:` Type

Nouvelles expressions de type

`<` variable `:` Type [`;` variable `:` Type] * [`;` `..`] `>`
`(` Type `as` Type-var `)`

Exemple

Le type (inféré) des classes avec relevé :

```
class compte_avec_relevé :
  object
    val mutable registre : opération list
    val mutable solde : float
    method dépôt : float -> unit
    method private intérêts : unit
    method relevé : opération list
    method retrait : float -> float
    method solde : float
    method private trace : opération -> unit
  end
```

Slide 22

Abréviations automatiques

Les définitions de classes (et de types de classes) créent automatiquement des abréviations de types

```
type compte_avec_relevé =
   < dépôt : float -> unit; relevé : opération list;
     retrait : float -> float; solde : float >
```

Slide 23

```
type #compte_avec_relevé =
   < dépôt : float -> unit; relevé : opération list;
     retrait : float -> float; solde : float; .. >
```

Le premier est une instance du second. En fait tout objet d'une sous-classe de `compte_avec_relevé` est une instance du second.

```
fun x -> ( (x : compte_avec_relevé) : #compte);;
```

Sous-typage sur les objets

Les types `compte_avec_relevé` et `compte` sont incompatibles.

```
fun x -> ( (x : compte_avec_relevé) : compte);;
```

Mais on peut explicitement coercer l'un vers l'autre :

```
fun x -> (x : compte_avec_relevé :> compte);;
```

Il existe un raccourci, moins puissant mais souvent suffisant *(utiliser la version longue lorsque le raccourci ne s'applique pas)*

Slide 24

```
fun x -> (x :> compte);;
```

Par exemple, pour mettre les comptes avec ou sans relevé dans une même collection :

```
[ (new compte :> compte);
  (new compte_avec_relevé :> compte) ];;
```

```
- : compte list = [<obj>; <obj>]
```

Règle de sous-typage (approximation)

Un type d'objet A est un sous-type d'un type d'objet B si A et B sont égaux ou bien tous deux fermés et tels que :

- chaque méthode de A est une méthode de B ;
- son type dans A est un sous-type de son type dans B (sous-typage en profondeur) ;

Le type de $D_A \to I_A$ d'une fonction est sous-type de $D_B \to I_B$ si

Slide 25

- I_A est sous-type de I_B *on peut agrandir l'Image*
- D_B est sous-type de D_A *on peut rétrécir son Domaine*

En particulier, si A et B contiennent une même *méthode binaire*, *e.g.* :

```
A = (< m : 'a -> unit; ... > as 'a)
B = (< m : 'b -> unit; ... > as 'b)
```

alors A est sous-type de B seulement s'il est égal à B. En effet, pour que A soit sous-type de B la contavariance sur la méthode m ne impose que B soit aussi sous-type de A, et la seule solution est que A soit égal à B. (On vérifie facilement la relation de sous-typage est antisymmétrique.)

Que peut-on cacher dans une classe ?

Les variables d'instance et les méthodes privées peuvent être cachées en donnant à la classe une contrainte de type.

```
class foo : object method m : int end =
  object (self)
    val foo = 2
    method private bar = foo * foo
    method m = self # bar * self # bar
  end;;
```

Slide 26

Le même effet peut être produit par une contrainte de signature

```
module Truc : sig class inutile : object end end =
  struct
    class inutile = object val foo = 2 end
  end;;
```

Classes paramétrées

Une mappe est une table d'association.

```
class type ['a, 'b] mappe =
  object
    method trouve : 'a -> 'b
    method ajoute : 'a -> 'b -> unit
  end;;
```

Slide 27

Une implémentation simple et raisonnable pour de petites mappes avec des listes :

```
class ['a, 'b] petite_mappe : ['a, 'b] mappe =
  object
    val mutable table = []
    method trouve clé = List.assoc clé table
    method ajoute clé valeur = table <- (clé, valeur) :: table
  end;;
```

Classes paramétrées

Une implémentation plus efficace pour les mappes de grande taille avec une table de hache. Pour résoudre les conflits, les éléments de la table de hache sont eux mêmes des petites mappes.

Slide 28

```
class ['a, 'b] grande_mappe taille : ['a, 'b] mappe =
  object (self)
    val table =
      Array.init taille (fun i -> new petite_mappe)
    method private hash clé =
      (Hashtbl.hash clé) mod (Array.length table)
    method trouve clé = table.(self#hash clé) # trouve clé
    method ajoute clé = table.(self#hash clé) # ajoute clé
  end;;
```

Une version plus efficace pourrait automatiquement retailler la table de hache lorsque les petites mappes deviennent trop grandes (à condition de leur ajouter une méthode taille).

Exercice

Exercice 2 (Piles (*)) *Écrire une classe des piles (d'entiers) Généraliser en une classe paramétrique des piles. Notez les différences avec les piles comme module.* Réponse □

Slide 29

Classes et méthodes virtuelles

Slide 30

Une méthode est virtuelle lorsqu'elle est utilisée mais pas définie. La classe est alors elle même virtuelle : on peut en dériver des sous-classes, mais pas en créer des instances.

Les méthodes virtuelles peuvent être utilisées pour définir un comportement commun aux sous-classes utilisant des caractéristiques pas encore définies des sous-classes.

Une classe virtuelle doit être indiquée comme telle. (ce qui évite qu'une méthode mal orthographiée deviennent accidentellement virtuelle...)

Exemple Une classe définit des règles générales, un protocole, etc. Une ou plusieurs sous classes implémentent une stratégie, un comportement plus précis.

Exemple : jeu et stratégies

Slide 31

Les règles du jeu sont partagées

```
class virtual joueur nom =
  object (self : 'a)
    method virtual coup : int
    method vérifie n = 1 + (n-1) mod 2
    method joue_avec (partenaire : 'a) jeu =
      if  (jeu = 0) then (nom : string) else
      let suivant = jeu - (self # vérifie (self # coup)) in
      partenaire # joue_avec self suivant
  end;;
```

Exemple : jeu et stratégies

Slide 32

Deux stratégies indépendantes

```
class petit nom =
  object
    inherit joueur nom
    method coup = 1
  end;;
```

```
class hasard nom =
  object
    inherit joueur nom
    method coup =
      1 + Random.int 2
  end;;
```

La partie

```
(new hasard "Pierre") # joue_avec
  (new petit "Jacques") 26;;
```

Les modificateurs de champs

Slide 33

Les modificateurs de champs sont visibles et hérités dans les sous-classes, à moins qu'ils ne soient masqués par le typage.

val mutable

Ce modificateur peut être ajouté tardivement et masqué par le typage.

method private

Annotation à placer dès l'origine, héritée. La méthode, invisible dans l'objet, peut être cachée dans la classe par le typage.

class virtual, method virtual

Annotation obligatoire pour une méthode utilisée mais non définie et pour une classe comprenant une telle méthode. Propriété héritée qui disparaît après définition de la méthode.

Objets dans le style fonctionnel

On peut retourner une copie superficielle d'un objet quelconque en utilisant la fonction de librairie :

```
Oo.copy : (< .. > as 'a) -> 'a
```

Dans une définition de classe, on peut aussi retourner une copie de self avec certains champs éventuellement modifiés par la construction :

Slide 34

$$\{\langle \ell_1 = e_1; \ldots \ell_k = e_k \rangle\}$$

où les champs ℓ_1 à ℓ_k sont un sous-ensemble des variables d'instances.

Exemple : les mappes fonctionnelles

La méthode `ajoute` doit retourner une nouvelle mappe, sans modifier l'ancienne.

```
class type ['a, 'b] mappe_fonctionnelle =
  object ('mytype)
    method trouve : 'a -> 'b
    method ajoute : 'a -> 'b -> 'mytype
  end;;
```

Slide 35

Des petites mappes peuvent être représentées par des listes :

```
class ['a, 'b] petite_mappe : ['a, 'b] mappe_fonctionnelle =
  object
    val mutable table = []
    method trouve clé = List.assoc clé table
    method ajoute clé valeur =
      {< table = (clé, valeur) :: table >}
  end;;
```

Héritage multiple

Une classe en librairie...

```
class sauvegarde =
  object (self : 'mytype)
    val mutable copie = None
    method sauve = copie <- Some {< >}
    method récupère =
      match copie with (Some x) -> x | _ -> raise Not_found
  end;;
```

Slide 36

Note : ici on peut remplacer `{< >}` par `(Oo.copy self)`.

Les fonctionnalités de la classe `sauvegarde` peuvent être ajoutées *a posteriori* à (presque) n'importe quelle autre classe, *e.g.* :

```
class compte_protégé =
  object inherit compte inherit sauvegarde end;;
```

Exercices

Exercice 3 (Sauvegarde) *Les fonctionnalités de la classe sauvegarde peuvent être ajoutées à* presque *n'importe quelle autre classe. Pourquoi* presque *?* Réponse

Slide 37

La classe sauvergarde permet chaîne en fait les versions entre elles. On peut donc récupérer une version de deuxième génération entre récupérant la sauvegarde de la sauvegarde.

Écrire une variante de la classe sauvegarde qui ne conserve qu'une seule copie. Quel est l'intérêt de cette modification (si une seule copie est nécessaire) ? Réponse

Écrire une variante fonctionnelle ***`fsauvegarde`*** *de la classe sauvegarde : la méthode récupère retourne un objet qui est une copie de* ***`self`*** *dans laquelle le champ* ***`original`*** *pointer vers la version originale. Il faut continuer avec la copie pour pouvoir plus tard récupérer la version originale.* Réponse

On reprend la version qui conserve les sauvegardes intermédiaires. Rafiner la version impérative de la sauvergarde en profondeur en ajoutant une méthode `balai` *qui conserve les sauvegardes de façon logarithmique (*i.e. *ne conserve que les versions d'âge* 2^0, 2^1, ... 2^n*).* Réponse □

Slide 38

Méthodes binaires

Ce sont des méthodes qui prennent parmi leurs arguments au moins un objet du même type que self.

Slide 39

Les vraies

```
class binaire = object (self)
  method choisir x = if Random.int 1 = 0 then x else self
end;;
class binaire : object ('a) method choisir : 'a -> 'a end
```

Le type d'un objet avec une méthode binaire *visible* n'a que lui-même pour sous-type. En corollaire, pour cacher certaines méthodes par sous-typage, il faut aussi cacher toutes ses méthodes binaires.

Les fausses La contrainte de type à l'origine de la méthode binaire peut être relâchée, en générale en rendant la classe paramétrique en le type de cet argument.

Méthodes binaires et classes paramétrées

Slide 40

```
class étoile =
  object (self : 'a)
    val mutable pos = 0
    method pos = pos
    method eq (x : 'a) =
      (x#pos = self#pos)
  end;;
```

```
class étoile : object ('a)
  val mutable pos : int
  method eq : 'a -> bool
  method pos : int
end
```

La méthode `eq` est-elle binaire ?

OUI

```
class ['a] étoile =
  object (self)
    val mutable pos = 0
    method pos = pos
    method eq (x : 'a) =
      (x#pos = self#pos)
  end;;
```

```
class ['a] étoile :  object
  constraint 'a = < pos:int; .. >
  val mutable pos : int
  method eq : 'a -> bool
  method pos : int
end
```

NON

Exercices

Exercice 4 (Méthodes binaires (**)) *Créer une classe* `string` *avec les fonctionnalités essentielles des chaînes.*

Ajouter une méthode `concat` *aux chaînes de caractères.*

Quel est le problème éventuel ? Réponse □

Slide 41

Exercices (suite)

Exercice 5 (Renommage (***)) *En considérant que la monnaie par défaut est le franc, étendre la classe* `compte` *pour fournir des dépôts et des retraits en euros, les conversions étant explicites. (Ici, on ne demande pas de cacher la représentation de la monnaie par l'utilisation de type abstrait ; on pourra donc utiliser des float à la fois pour représenter les dollars et les francs. Voir l'exercice sur les taux de changes.)*

Slide 42

Reprendre le même exercice, mais en définissant d'abord une classe `euro_compte` *puis une ensuite classe* `compte_mixte` *par héritage multiple.*

Quel est le problème éventuel ? Réponse □

1 Solutions des exercices

Exercice 1, page 9

Il n'y a pas d'intérêt a priori, sauf situation très particulière. Cette transformation vient surtout des langages tout objets qui n'ont pas de fonctions mais seuleument des méthodes. En Ocaml, les fonctions sont faciles à définir et à manipiler. Lorsque cette approche suffit, elle est plus légère que le style à objets et donc préférable.

Exercice 1 (continued)

Lorsque des fonctions polymorphes sont mises dans une classes ont voudrait en faire des méthodes polymorphes. Mais ce n'est pas possibles car les objets ne sont jamais polymorphes (comme des références, ils contiennent des champs mutables), et leurs méthodes non plus.

Au mieux on pourrait définir une classe paramétrique (par exemple des opérations sur les listes), mais ils faudra spécialiser celles-ci complétement à la création d'un objet. C'est une limitation forte à l'usage des classes pour regrouper des fonctions de librairie.

Exercice 2, page 29

On ne donne que la solution pour des classes paramétriques représentées par des listes.

```
exception Vide;;
class ['a] pile =
  object
    val mutable p : 'a list = []
    method ajouter x = p <- x :: p
```

```
    method retirer =
      match p with
      | [] -> raise Vide
      | x::t -> p <- t; x
  end;;
```

Exercice 3, page 37

Les noms de méthodes cette autre classe ne doivent pas être en conflit avec ceux de la classe sauvegarde.

Exercice 3 (continued)

Il suffit lorsqu'on prend une copie de mettre le champ copie à `None` :

```
class une_sauvegarde =
    object (self : 'mytype)
      inherit sauvegarde
      method sauve = copie <- Some {< copie = None >}
    end;;
```

Si une seule copie est nécessaire, la version ci-dessus libère les anciennces versions qui peuvent être récupérées par le GC. Dans la version précédentes toutes les versions intermédiaires restaient vivantes aussi longtemps que la version de travail.

Exercice 3 (continued)

```
class fsauvegarde =
  object (self)
    val original = None
    method copie = {< original = Some self >}
    method récupère =
      match original with None -> self | Some x -> x
  end;;
```

Exercice 3 (continued)

La méthode `balai` efface récursivement et physiquement certaines copie en les faisant pointer vers la copie précedente.

```
class sauvegarde_autogérée =
  object (self : 'mytype)
    inherit sauvegarde
    method private coup_de_balai p q =
      try
        let q' = if p = q then 2 * q else q in
        let la_sauvegarde = self # récupère # coup_de_balai (succ p) q' in
        copie <- la_sauvegarde;
        if p = q then Some self else la_sauvegarde
      with Not_found -> None
    method balai = ignore (self#coup_de_balai 1 1)
  end;;
```

Remarquer l'importance de faire une méthode auxilliaire de `coup_de_balai`. Une fonction ne conviendrait pas, car `coup_de_balai` utilise la variable d'instance copie.

Exercice 4, page 41

La méthode concat pose problème parce que c'est une méthode binaire. En fait, c'est une pseudo méthode binaire. En effet, si on naturellement :

```
class ostring s = object (self)
  val s = s
  method repr = s
  method concat t = {< s = s ^ t # repr >}
end;;
```
alors l'argument `t` n'est pas nécessairement du type de self, mais doit seulement posséder une méthode repr du bon type. Cela rend la class paramétrique (d'où le message d'erreur). Une solution est donc :
```
class ['a] ostring s = object (self)
  val s = s
  method repr = s
  method concat (t:'a) = {< s = s ^ t # repr >}
end;;
```
Une autre solution, sans doute meilleurs en pratique, et de contraindre le type 'a de t a être le même que le type de self (la classe n'est plus paramétrique)
```
class ostring s = object (self : 'a)
  val s = s
  method repr = s
  method concat (t:'a) = {< s = s ^ t # repr >}
end;;
```
Attention ! on pourrait aussi contraindre `t` à être du type de la classe `string`
```
class ostring s = object (self : 'a)
  val s = s
  method repr = s
  method concat (t : ostring) = {< s = s ^ t # repr >}
end;;
```
Cette solution n'est pas bonne car dans une sous-classe enrichie `ostring_plus`, la méthode `concat` ne se comportera pas comme attendu : elle prendra et un argument du type `ostring` et non pas du type `ostring_plus`.

Exercice 5, page 42

Le problème éventuel est le conflit de nom, si par exemple, la classe `euro_compte` utiliser les mêmes méthodes `dépôt` et `retrait` que la classe `compte`.

Malheureusement, il n'est pas possible (pour l'instant, en Ocaml) de renommer les méthodes.

Le choix des noms est important en style objet, car la programmation par "envoi de messages" attache un sémantique implicite à chaque message et les noms sont bien plus que des conventions syntaxiques. Par exemple, un object imprimable aura naturellement une méthode "print", sinon il ne pourra pas être imprimé par d'autres classes définies en librairie.

La solution la plus simple consiste à prévoir ce conflit et choisir des noms différents pour chacune des méthodes.
```
class euro_compte = object
  val sole_en_euro = ...
  method dépôt_en_euro = ...
end
```
Une autre solution, plus modulaire, sinon plus élégante, permet toutefois de partager le compte en euro et en francs. Mais il faut pour cela prévoir le conflit, et utiliser un motif en peigne (cf introduction) avec une classe génératrice, où les noms sont privés, et des classes dérivées où les noms sont figés. (On choisi une version simplifié, mais on imaginera une version raffinée de la classe ou le corps de la méthode `dépôt` est suffisamment gros pour justifier le partage).
```
class compte = object
  val mutable solde = 0.
  method private dépôt x = solde <- solde +. x; solde
end;;
```

Pour la classe en franc, on choisit des nom précis

```
class type compte_français = object
  method dépôt_en_franc : float -> float
end

class compte_en_franc : compte_français = object (self)
  inherit compte
  method dépôt_en_franc = self#dépôt
end;;
```

Pour de façon similaire en évitant les conflits de noms :

```
class type compte_européen = object
  method dépôt_en_euro : float -> float
end

class compte_en_euro : compte_européen = object (self)
  inherit compte
  method dépôt_en_euro = self#dépôt
end;;
```

On peut alors mélanger les deux comptes :

```
class compte_mixte = object
  inherit compte_en_franc
  inherit compte_en_euro
end;;
```

On vérifie que la class `compte_mixte` comporte bien deux variables d'instances soldes cachées distinctes : il n'y a pas d'override entre les champs (variables ou méthodes) cachés. (Dans cette version simplifié du compte, on effectue un dépôt vide pour lire la valeur du compte) :

```
let cm = new compte_mixte;;
let soldes() = (cm # dépôt_en_franc 0., cm # dépôt_en_euro 0.);;
soldes();;
```

```
- : float * float = 3.000000, 4.000000
```

```
cm # dépôt_en_franc 1.;;
```

```
- : float = 4.000000
```

```
soldes();;
```

```
- : float * float = 4.000000, 4.000000
```

```
cm # dépôt_en_euro 2.;;
```

```
- : float = 6.000000
```

```
soldes();;
```

```
- : float * float = 4.000000, 6.000000
```

NOTE : On a ainsi résolu le problème de façon modulaire (partage du code), mais au pris d'un motif en peigne, qu'il a fallu concevoir a priori.