# Plan du chapitre *Objets*

1. Introduction - l'approche orientée objet
2. Un très bref historique des langages orientés objet
3. Les objets en OCaml
4. Héritage et classes virtuelles
5. Typage des classes en OCaml
6. Classes paramétriques
7. Objets et sous-typage

# 7 Introduction - l’approche orienté objet

## 7.1 Structuration par données . . .

. . . au lieu de structuration par fonctions (découpage vertical au lieu de découpage horizontal). Voir par exemple le TD sur le découpage de l’interpréteur en modules.

Un *objet* est une entité autonome qui « sait » répondre à une requête.

Au lieu d’appliquer des fonctions à des données, on envoie maintenant à des objets des *messages* demandant à l’objet l’exécution d’une *méthode*.

**Avantages**

- une modélisation par objets correspond souvent mieux au monde réel. Par exemple : dans une interface graphique, on peut modéliser les fenêtres différentes par des objets, et envoyer aux fenêtres des messages demandant d'afficher qqc ou de changer leur configuration.
- dans un projet à longue durée et évolutif, la structuration en objets est souvent plus stable que les spécifications des fonctions. Donc, il semble pertinent de prendre la structuration en donnée comme base, au lieu de la structuration en fonctionnalités qui risque de changer plus rapidement.

## 7.2 Des objets différents comprennent le même message

. . . mais y réagissent par l'exécution de leur implantation privée de la méthode demandée.

Exemple : Des objets différents qui implantent tous une méthode *afficher*. L'implantation de cette méthode, et le format d'affichage, dépendent de la nature de l'objet qui exécute cette méthode.

La réalisation d'une fonctionnalité est localisée dans l'objet qui l'exécute et pas centralisée dans une définition de fonction.

Conséquence sur la structure du code :

Avant on avait une structure comme

```
type t = Constr1 of t1 | Constr2 of t2
let affiche = function
    Constr1(x) -> ......
  | Constr2(x) -> ......
```

Avec une approche orientée objet la structure est comme suit :

```
classe Constr1(t1)
   m\’ethode affiche -> ....
end

classe Constr(t2)
   m\’ethode affiche -> ...
end
```

Puis, on peut créer des *objets* des classes définies et leur envoyer le message « affiche-toi »

```
x := new Constr1 (argument1)
x.affiche
x := new Constr2 (argument2)
```

```
x.affiche
```

qui déclenche dans les deux cas l'exécution d'une méthode différente. C'est au moment d'exécution que la méthode effectivement exécutée est déterminée (principe de la « liaison retardée » (angl. : late binding)

(Questions laissé ouvertes pour l'instant : quel est le type de la variable `x` ? et ou est le type `t` du programme du départ ?)

Dans la première approche : Quand on veut ajouter un nouveau constructeur `Constr3 of t3` dans la définition du type `t` il faut changer le code de toutes les fonctions qui font un filtrage sur un argument du type `t`. Ces fonctions peuvent être localisés dans des modules différents qu'il faut donc tous ré-compiler.

Par contre dans l'approche orientée objet, on définit simplement une

nouvelle classe

```
classe Constr3(type3)
   m\'ethode affiche -> ...
end
```

et il n’est pas nécessaire de toucher le code des anciennes classe, ni de les ré-compiler. Donc, l’extension de la définition des types et plus facile.

Par contre, le problème revient partiellement quand on essaye d’ajouter une nouvelle fonction :

– dans la première approche : facile
– dans l’approche orientée objet : définir une nouvelle méthode dans toutes les classes (il y a dans certains cas une meilleure solution à l’aide de l’héritage, voir au-dessous)

## 7.3 L’héritage

On peut écrire des classes et puis les spécialiser par des sous-classes.

Les sous-classes *héritent* les méthodes de leur super-classe, peuvent définir des nouvelles méthodes et même ré-définir des méthodes héritées.

```
classe T
  m\’ethode  f -> ...
end
classe Constr1 (type1)
  sous-classe de T
  m\’ethode afficher -> ...
end
classe Constr2 (type 2)
  sous-classe de T
```

```
  m\'ethode afficher -> ...
end
```

Donc solution partielle au problème d'ajout des nouvelles fonctions : quand la fonction n'est pas spécifique au classes `Constr1`, `Constr2`, alors on peut mettre la définition de la nouvelle fonction dans la classe `T`. Par contre, il n'y a pas de bonne solution quand l'implantation de la méthode dépend de la sous-classe.

Réponses aux questions laissé ouvertes avant ?

– le type `t` est devenu la super-classe `T`
– le type de la variable `x` (qui prend comme valeurs des objets des classes `Constr1` et aussi `Constr2` *peut* être `T` (dépend du langage de programmation, en particulier attention à la relation entre héritage et sous-typage !).

## 7.4 Les objets peuvent avoir un « état » . . .

. . . réalisé par des *variables d'instance* : variables qui sont locales à l'objet, et dont l'accès est restreint (par exemple : seulement visibles pour les méthodes de la classe). L'état est donc encapsulé par l'objet.

(voir l'exercice à la fin du chapitre *Programmation modulaire*.)

Ce n'est pas un trait essentiel des objets, on peut aussi avoir des objets purement fonctionnels.

## 7.5 Récapitulatif des notions principales

**Classe** une classe est un ensemble agrégé de champs de données (appelés des variables d'instance) et de traitements (appelés méthodes).

**Objet** un objet est un élément (ou instance) d'une classe. Un objet possède les comportements de la classe à laquelle il appartient. L'objet est le composant effectif des programmes (c'est lui qui calcule) alors que la classe est plutôt une définition ou une spécification pour l'ensemble des instances à venir.

**Méthode** une méthode est une action qu'un objet est à même d'effectuer.

**Message** un envoi de message à un objet est la demande faite à ce dernier d'exécuter une de ses méthodes. On pourra également dire que l'on invoque une méthode.

# 8 Un très bref historique des langages à objets

## 8.1 SIMULA

- 1967 (Ole-Johan Dahl et Kristen Nygaard)
- basé sur ALGOL (un grand classique de la programmation impérative et des langages structurés)
- Domaine d'application : simulations des systèmes complexes
- tout sur l'histoire de SIMULA : `http://java.sun.com/people/jag/SimulaHistory.html`

## 8.2 Smalltalk

– 1976 (Adele Goldberg de chez XEROX PARC)
– langage non-typé, un peu dans la tradition de LISP (un grand classique de la programmation fonctionnelle)
– *tout* est un objet, même les nombres et les classes
– environnement graphique très innovateur (fenêtres, icônes, pointeur, qui sont modélisés comme des objets SMALLTALK)

## 8.3 À partir des années 80

– EIFFEL (Bertrand Meyer)
– C++ (Bjarne Stroustrup)
– Des extensions objets pour tous les langages : Ada 95, Clos (pour LISP), Turbo PASCAL, même pour COBOL

– et bien sûr JAVA (de chez SUN) mais ce langage a certainement aussi profité du succès de l’internet. Autre aspect important : mobilité de code (« applets »)

# 9 Les objets on Objective CAML

## 9.1 Définition d'une classe

```
class nom p1 ... pn = object
   ...
   variables d'instance
   ...
   m\'ethodes
end
```

où $p_1, \ldots, p_n$ sont les paramètres que prendra le constructeur de cette classe. Une classe peut n'avoir aucun paramètre.

Les « variables d’instance » se déclarent dans une des deux formes

```
val nom = expr
val mutable nom = expr
```

Toutes les variables d’instances sont visibles pour toutes les méthodes de la classe, mais elles ne sont pas visibles vers l’extérieur. L’accès de l’extérieur aux variables d’instances ne se fait que par les méthodes de la classe.

Dans le deuxième cas la variable est aussi modifiable par toutes les méthodes de cette classe, à l’aide de la construction (comme pour les champs mutables des enregistrements).

```
nom <- expr
```

Une méthode est déclarée par la construction

```
method nom p1 ...pn = expr
```

où $p_1, \ldots, p_n$ sont les paramètres que prendra une invocation de cette méthode. Une méthode peut n'avoir aucun paramètre (à distinguer du cas d'un paramètre `()`).

## 9.2 Création d'un objet

Un objet d'une classe $nom$ avec paramètres de types $t_1, \ldots, t_n$ est créé par

```
new nom exp1 ... expn
```

où les $exp_i$ sont des expresions de type $t_i$.

## Exemple (voir la démo `objets1.ml`)

```
(* Une classe très simple : les compteurs. On peut la simule
   dans CAML sans les objets.
*)

class counter (initial) = object
  val mutable c = initial
  method increment () = c <- c+1
  method show () = c
end

(* Maintenant on peut créer deux compteurs indépendants *)
let x = new counter 0;;
x#increment ();;
x#increment ();;
x#show ();;
```

```
let y = new counter 42;;
y#increment ();;
y#show ();;
x#show ();;

(* les variables d'instaces ne sont pas visibles *)
x#c;; (* erreur *)


(* Simulation en CAML sans objets. La variable d'instance x e
   ici simulée par un identificateur local qui n'est visible
   clôtures des fonctions increment et show.
*)

type counter = {
  increment: unit -> unit;
  show: unit -> int
}
```

```
let create_counter (initial) =
  let c = ref initial
  in
    {
      increment = (function () -> c := !c+1);
      show = (function () -> !c)
    }

let x = create_counter 0;;
x.increment ();;
x.increment ();;
x.show ();;
let y = create_counter 42;;
y.increment ();;
y.show ();;
x.show ();;

(* l'identificateur c n'est pas un champ des enregistrements
```

```
x.c;; (* erreur *)
```

(voir le schema comment traduire les classes simples en OCaml sans objets)

## 9.3 Type d'objet $\neq$ classe

Le *type d'un objet* est constitué des types des méthodes qu'il peut exécuter.
Conséquences :

– Les types des variables d'instances ne figurent pas dans le type d'un objet.
– Le nom de la classe est simplement une *abbréviation* pour l'ensemble des types de ses méthodes.
– Deux objets de classe différente peuvent avoir le même type.

Relation d'égalité entre objets : Un objet est seulement égal à lui même (identité physique). Deux objets créés indépendamment sont toujours

différents, même s'ils ont les mêmes implantations des méthodes et les mêmes variables d'instances avec les mêmes valeurs. (voir la démo `objet-types.ml`)

```
(* Deux classes du même type *)

class counter (initial) = object
  val mutable c = initial
  method increment () = c <- c+1
  method show () = string_of_int c
end

class counter2 (initial1, initial2) = object
  val mutable c1 = initial1
  val mutable c2 = initial2
  method increment () = c1 <- c1 + 1 ; c2 <- c2 * c2
  method show () = "[" ^ (string_of_int c1) ^ " , " ^ (strin
end
```

```
let x = new counter 0;;
x#increment ();;
x#show ();;
let y = new counter2 (1,2);;
y#increment ();;
y#show ();;
x=y;; (* x et y sont du même type ! *)

(* les noms des classes sont simplement des abbréviations pou
(x:counter) = (y:counter2);;

(* Définition d'une classe de type différent *)

class counter3 (initial) = object
  val mutable c = initial
  method incrementer () = c <- c+1
  method montrer () = string_of_int c
end
```

```
let z = new counter3 0;;

x = z ;; (* les types de x et z sont différents *)

(* Quand est-ce que deux objets sont égaux *)

let x1 = new counter 0;;
let x2 = new counter 0;;

x1 = x2 ;;
x1 = x1;;

(* chaque objet à sa propre identité *)



(* Simulation partielle en CAML sans objets. *)
```

```
type counter_type = {
  increment: unit -> unit;
  show: unit -> string
}

let create_counter (initial) =
  let c = ref initial
  in
    {
      increment = (function () -> c := !c+1);
      show = (function () -> string_of_int !c)
    };;

let create_counter2 (initial1, initial2) =
  let c1 = ref initial1
  and c2 = ref initial2
  in
    {
```

```
      increment = (function () -> c1 := !c1+1; c2 := !c2*(!c
      show = (function () ->
"[" ^ (string_of_int !c1) ^ " , " ^ (string_of_int !c2) ^ "]
    };;

(* les deux fonctions de création renvoient le même type *)
create_counter;;
create_counter2;;

let x = create_counter 0;;
x.increment ();;
x.show ();;
let y = create_counter2 (1,2);;
y.increment ();;
y.show ();;

x = x;;
```

```
(* cela ne marche pas parceque OCaml ne peut pas comparer les
x = y;;

(* L’identité des valeurs de type « enregistrement de fonctio
   différent de l’identité d’objets.
*)
```

## 9.4 Méthodes récursives et `self`

Dans la déclaration d'une classe on peut donner un nom qu'un objet peut utiliser pour envoyer un message à lui-même :

```
class nom p1 ...pn = object (self)
   ...
end
```

Le nom peut être choisi librement (dans autres langages orienté objets : mot clef pour designer l'objet lui-même `self`, ou `this` en C++ ).

Cela permet par exemple de réaliser des méthodes récursives : voir la démo `rec.ml`.

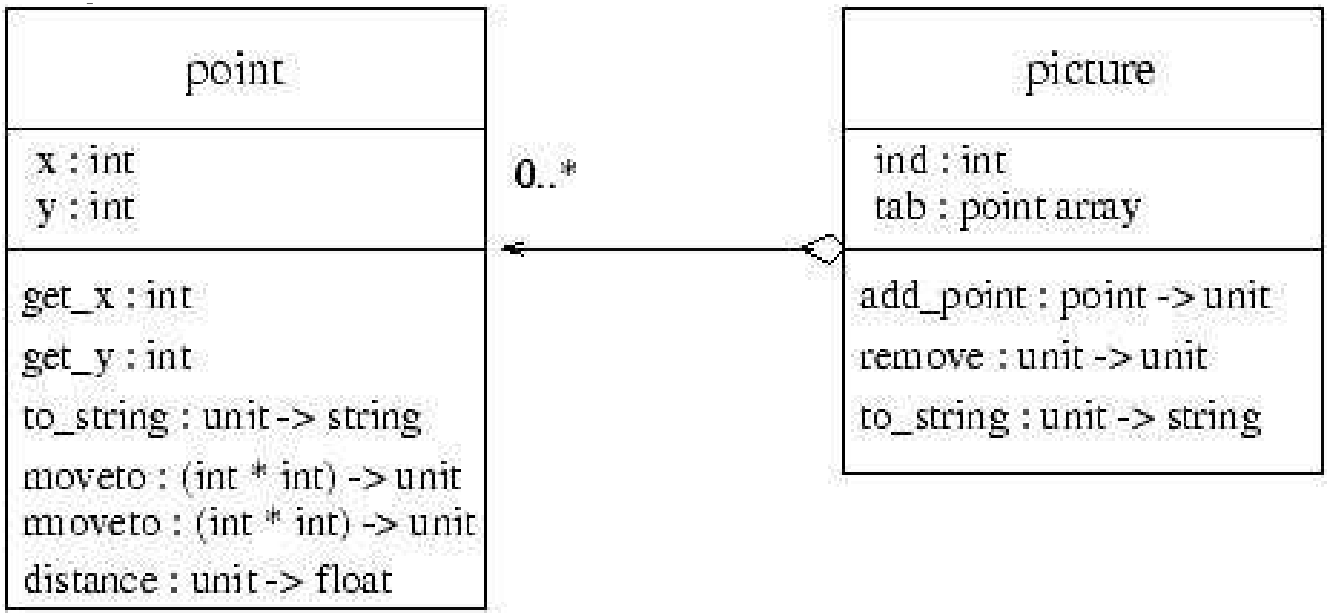
Dans cet exemple, nous avons vu comment généraliser le schema de traduction aux objets qui peuvent envoyer des messages à eux-mêmes.

Il s'agit cependant d'une première solution *naïve*, parce-que elle ne resistera pas à l'*héritage*, qui va exhiber la nature de recursion ouverte des messages envoyés à `self`.

## 9.5 Agrégation d’objets

Première relation importante entre classes : *agrégation* (l’autre rélation importante est l’*héritage*, voir plus tard) : Une classe peut définir des variables d’instances d’un type objet (listes d’objets etc.).

Notion graphique de UML (*Unified Modelling Language*) pour les classes et la rélation d’



Exemple : (voir la démo `points-pictures.ml`)

# 10 Héritage et classes virtuelles

## 10.1 Syntaxe de l'héritage

Dans le corps d'une classe on peut déclarer qu'on souhaite hériter les variables d'instances et les méthodes d'une autre classe :

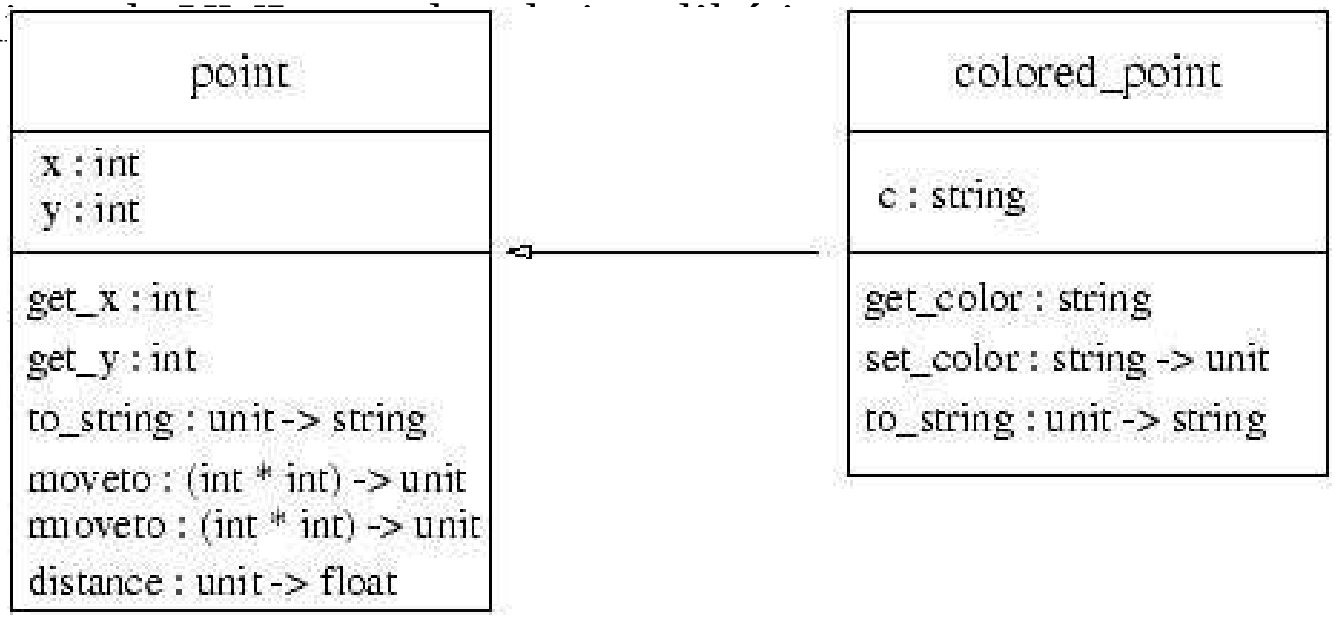
```
inherit nom1 p1 ...pn  [ as nom2 ]
```

où $nom_1$ est le nom de la classe de laquelle on veut hériter, $p_1, \ldots, p_n$ sont les paramètres nécessaires pour la création d'une instance de la classe $nom_1$, et $nom_2$ est un nom qu'on peut utiliser pour faire directement référence à des méthodes héritées, ce qui correspond au mot clef `super` dans d'autres langages orientés objet (voir plus tard).

Puis, une classe peut ré-définir des variables d'instances et des méthodes héritées.

1. la ré-définition d'une méthode doit conserver le type de la méthode
2. l'ancienne définition d'une méthode ré-définie est toujours accessible à l'aide du préfixe `nom2#`

Notation graphi




Exemple (démo `heritage.ml`)

## 10.2 Liaison retardée

Envoi de message $\neq$ appel d'une fonction !

Voir la démo `latebinding`.

**Exercice:** Expliquer comment traduire des définitions de classes en Caml sans objets et avec une discipline de typage relâchée, tel que le principe de la liaison retardé est conservé.

**Exercice:** Quelles sont les réponses aux deux invocations de méthodes dans le programme suivant (fichier `super.ml`) :

```
class c1 = object
  method m1 = "a"
end

class c2 = object (self)
  inherit c1 as super
  method m1 = "b"
  method m2 = super#m1
  method m3 = self#m1
end

class c3 = object
  inherit c2 as super
  method m1 = "c"
end
```

```
class c3’ = object (self)
  inherit c1
  method  m4=self#m1
  inherit c2
end

let x = new c3;;
x#m2;; (* quel est le résultat ? *)
x#m3;; (* quel est le résultat ? *)
```

## 10.3 Héritage multiple

Il peut être utile de hériter de plusieurs classes à la fois. Par exemple, en aurait pu définir la classe des point colorés par héritage des deux classes *point* et *couleur*.

Voir la démo `multiple`.

Problème : quoi faire quand une méthode ou variable est définie dans plusieurs classes ancêtres ?

Classe A : défini une variable $x$

Classe B : défini une variable $x$

Classe C : hérite à la fois de A et de B

La variable $x$ de A est différente de la variable $x$ de B, il faut trouver une solution pour résoudre le conflit, par exemple :

- c'est la dernière définition qui cache les définitions antérieures
- construction pour renommer les variables et méthodes au moment de l'héritage

La solution est moins évidente quand les deux variables $x$ viennent à l'origine de la même définition, par exemple :

Classe A : défini une variable $x$

Classe B : hérite de A, contient donc une variable $x$

Classe C : hérite de B, contient donc une variable $x$

Classe D : hérite à la fois de B et de C. On peut argumenter que D doit contenir les deux copies de $x$ ou une seule.

**Exemple 1**

Classe A : classe des *engins à moteur*, qui contient comme champs les caractéristiques du moteur (puissance, consommation, etc.)

Classe B : classe des *automobiles*, obtenue par héritage de A

Classe C : classe des *grues*, obtenue par héritage de A

Classe D : classe des *grues automotrices*, obtenue par double héritage de B et de C. On voudrait que le moteur « automobile » soit distinct du moteur « grue »

**Exemple 2**

Classe A : classe des objets mobiles, avec une variable *vitesse*

Classe B : classe des bateaux, obtenue par héritage de A

Classe C : classe de objets propulsés par le vent, obtenue par héritage de A

Classe D : classe des bateaux à voile, obtenue par double héritage de B et de C. On voudrait qu'il y ait une seule variable *vitesse*.

**La bonne solution ?**

La solution la plus propre est de retenir une seule instance de la variable obtenue par plusieurs « chemins » de héritage.

Dans l'exemple 1 : c'est plutôt un abus de l'héritage. Une meilleure modélisation est de hériter d'une seule classe (par ex. *automobile*), et de définir dans la classe des grues automotrices une variable d'instance qui contient le moteur de la grue.

Mais : problème de complexité de déterminer si deux définitions de la même variable ont le même origine.

**Exercice:** Proposer un expérience qui permet de déterminer quelle est la solution choisie en OCaml.