Objets

Objets

- des objets pour structurer le code

```
class point =    (* grosse tarte à la crème *)
object
   val mutable coord = 0
   method pos = coord
   method show = (print_string "position:  "; print_int coord)
   method move dx = coord <- coord + dx
end
```

- des *valeurs* (variables d'instance) et des *méthodes* :
  - l'état et de quoi agir dessus
  - l'intérieur et l'extérieur (d'où la méthode `pos`)
- à première vue, pas beaucoup plus qu'un enregistrement

Manipuler les objets

```
class point =
object
   val mutable coord = 0
   method pos = coord
   method show = (print_string "position:  ";
        print_int coord)
   method move dx = coord <- coord + dx
end

let p1 = new point in
let p2 = new point in
p1#move 3;
p2#move 9;
let milieu = new point in
milieu#move ((p1#pos + p2#pos)/2);
milieu#show;;
position:  6
```

- `new`: engendre un objet à partir d'une classe
  - `let o = new c` : l'objet `o` est une *instance* de la classe `c`
  - il y a du `malloc` là-dessous
    + *initialisation*: les valeurs des variables sont affectées
  - on espère que le code des méthodes est partagé entre objets instances de la même classe (les valeurs pas, bien entendu)
- ⟨objet⟩#⟨méthode⟩ : appel de méthode (ou 'passage de message')
  - l'appel se fait depuis l'*extérieur* de l'objet

Soi

self: l'objet dans lequel réside la méthode que l'on appelle

```
class point = object (moi)
  val mutable coord = 0
  method pos = coord
  method show = (print_string "position:  "; print_int coord)
  method bla = (print_string "je suis un point; ma "; moi#show)
  method move dx = coord <- coord + dx
end
```

- ici `moi#pos` aurait pu être `coord`
- typage: OCaml répond

```
class point :  object
   val mutable coord :  int
   method bla :  unit
   method move :  int -> unit
   method pos :  int
   method show :  unit        end
```

Héritage

- réutilisation, partage du code

```
class colpoint = object
 inherit point       (* "extends" en drei *)
 val mutable color = "rouge"
 method paint c = color <- c
 method show =
    Format.printf "couleur:  %s, position:  %d\n" color pos
end
```

  - on ajoute `color` et `paint`
  - on modifie `show` (obligation: *en préservant le même type*)

- spécialisation du code: *liaison tardive* (ou retardée)

```
class c = object (self)
   method even n = n=0 || self#odd (n-1)
   method odd n = n=1 || self#even (n-1)   end

class d = object
    inherit c
    method even n = (n mod 2) = 0   end
let o = new d in o#odd 1 000 001

let rec even n = n=0 || odd (n-1) and odd n = n=1 || even (n-1)
let even n = (n mod 2) = 0 in    odd 1 000 001
```

- spécialisation du code: *liaison tardive* (ou retardée)

Classe virtuelle

- on peut décrire *partiellement* une classe

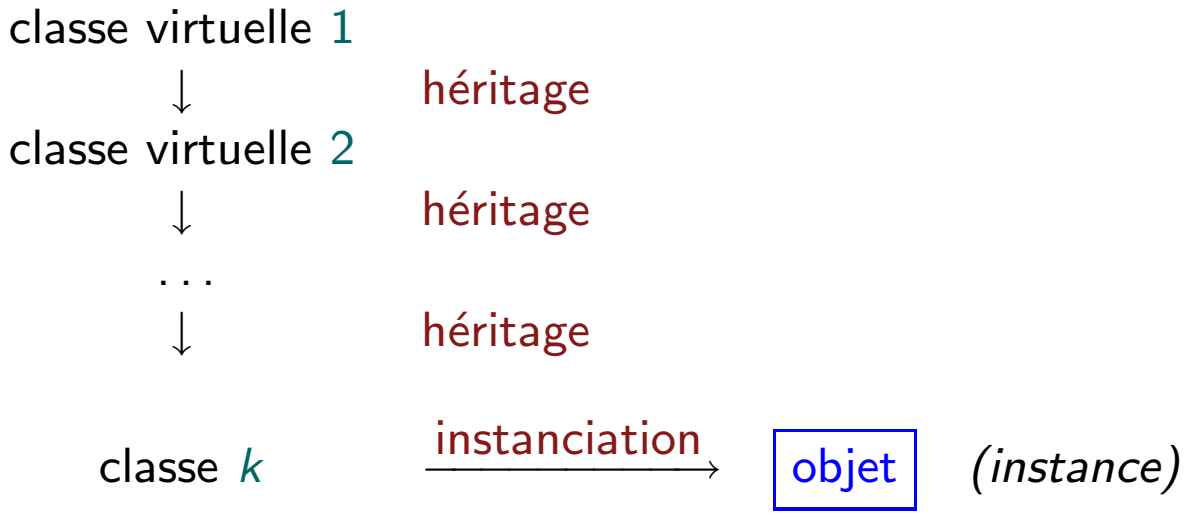
```
class virtual cv =                          classe virtuelle
object (self)
  method virtual m :  int                   méthode virtuelle
  method n = 11 + self#m
end
```

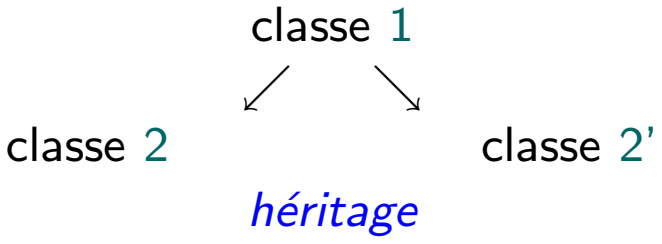
  - le `virtual` est obligatoire
  - pas de `new` sur un `virtual`

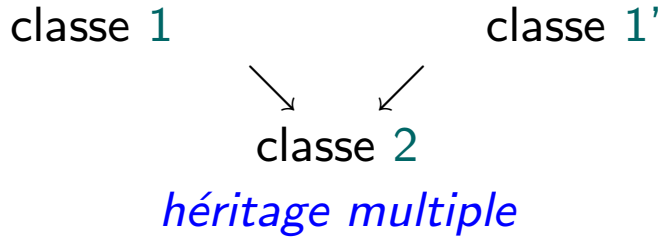
- on précise les choses lors de l'héritage

- une manière de *spécifier* le code (un peu le rôle d'un `.mli`)

Classes, classes virtuelles, instances







Objets et méthodes – typage

- les objets peuvent être vus comme des *enregistrements extensibles*

```
# let f o = o#m;;
val f : < m : 'a; .. > -> 'a = <fun>
```

`< m : t; m' : t'; .. >` *l'ellipse*

```
# let f o = o#a#b + o#c;;
val f : < a : < b : int; .. >; c : int; .. > -> int = <fun>
```

- les objets sont *dans le langage* *(pas les classes)*

  mais pas de `let f o m = o#m` : les noms de méthodes ne sont pas au même niveau

- l'ellipse introduit une forme de polymorphisme
  - pour les fonctions qui s'appliquent à des objets
  - polymorphisme de rangée

Sous-typage explicite: coercions

- à l'image des modules, il y a du sous-typage dans les objets: oubli de composants
  - sous-typage $\not\Leftrightarrow$ héritage
- en OCaml, le sous-typage entre types d'objets est *explicite*: l'utilisateur doit employer des **coercions**
  - $\neq$ *utilisation de l'ellipse*
  - syntaxe: `o :> t`
  - DÉMO `coercions.ml`
- l'importance du typage/sous-typage est variable selon les langages
  - rien (Smalltalk)
  - typage peu flexible et coercions (Java: vérif. à l'exécution, C++, Object Pascal: rien)
  - coercion vers un sous-type: Eiffel

Types récursifs

- types récursifs: option `-rectypes` de Caml

  DÉMO `rectypes.ml`

- on a besoin de récursion pour typer les objets

```
# let o = object (self) method f = self end;;
val o : < f : 'a > as 'a = <obj>
```

- la fonction `Oo.copy` permet de créer un nouvel objet, qui est une copie de l'objet passé en argument
  quel est son type? `(< .. > as 'a) -> 'a`

Méthodes polymorphes

```
# (fun x -> fun y -> x,y);;
- :  'a -> 'b -> 'a * 'b = <fun>
# (fun x -> fun y -> x,y) 3;;
- :  '_a -> int * '_a = <fun>

# class iteration = object
method itere f acc = List.fold_left f acc [23;15;25] end;;
(...)  Some type variables are unbound in this type:
class iteration :  object method itere : ('a -> int -> 'a) -> 'a -> 'a end
The method itere has type ('a -> int -> 'a) -> 'a -> 'a where 'a is unbound

# class ['a] iteration = object
method itere f (acc:'a) = List.fold_left f acc [23;15;25] end;;
class ['a] iteration :
object method itere :  ('a -> int -> 'a) -> 'a -> 'a end

# let o = new iteration;;
val o :  '_a iteration = <obj>


# class iteration = object
method itere :  'a.  'a -> ('a -> int -> 'a) -> 'a
= fun acc -> fun f -> List.fold_left f acc [23;15;25] end;;
class iteration :  object method itere : 'a -> ('a -> int -> 'a) -> 'a end
```

Considérations diverses

les objets en Caml ressemblent à

- des modules
  - il manque **les types** (dans les objets) et l'encapsulation
    mais
    - quand on appelle une méthode, on ne sait pas quel est le code qui va être exécuté; l'utilisateur n'a pas à se préoccuper de quel code va tourner
      pas de TypeCase en OCaml (pas d'information de type à l'exécution)
    - méthodes privées, protégées, etc.
  - plus que ressembler, c'est conçu pour des objectifs semblables

- des enregistrements, avec en plus
  - distinction valeur-méthode, appel de méthode
    - l'ajout des objets modifie de manière "intime" le langage: un nouveau mécanisme d'évaluation (#)
  - héritage (via les classes)

Vers une compréhension “essentielle”

- une classe (constructeur d'objets) peut être codée comme une (sorte de) fonction qui
  - prend `self` (et des valeurs pour les variables d'instance) en argument
  - renvoie un enregistrement
- héritage: modification du corps de la fonction
- faire `new`, c'est prendre un *point fixe* (fermer la boucle)
  - comme quand on avait évoqué la récursion ouverte
  - un objet est un enregistrement récursif
- faire un système de types raisonnablement expressif prenant en compte les traits objets est une gageure

OO: un champ vaste

- on n'a mentionné que les traits essentiels des (de certains) langages de programmation orientée objets
- foisonnement
  - origines: Smalltalk (et quelques antécédents)
  - nombreuses propositions (class-based, object-based)
  - langages à grande diffusion sont class-based: C++, Java, C# (mais aussi Simula, Smalltalk, Modula-3)
- tour de Babel au sein de la tour de Babel:
  - réponses différentes aux questions typiques du génie logiciel: réutilisation, encapsulation, visibilité, etc.
  - un langage OO = souvent une combinaison "bien pratique" de mécanismes, plus il y en a plus c'est classe
  - des tentatives pour dégager un corpus de notions élémentaires permettant d'expliquer le plus grand fragment possible de la programmation orientée objet

merci Martin Abadi

Why Objects

Who needs object-oriented languages, anyway?

- Systems may be modeled by other paradigms.
- Data abstraction can be achieved with plain abstract data types.
- Reuse can be achieved by parameterization and modularization.

Anyway, the object-oriented approach has been uniquely successful:

- Some of its features are not easy to explain as the union of well-understood concepts.
- It seems to integrate good design and implementation techniques in an intuitive framework.

Objects are just data structures with an attitude.

Alex Morrow