Slide 1

Plan du cours 4

- Types et objets
  - sous-typage
  - style fonctionnel
  - interface
- Comparaison modules-objets

Slide 2

Programmation objet

Encapsulation:

Classes et Instances:

Relations : agrégat et héritage:

Redéfinition et Liaison Retardée:

Polymorphisme et Sous-Typage:

Slide 3

Terminologie Objet

- classe = description des données et des procédures qui les manipulent ⇒ définit des comportements
- objet = instance d'une classe (possède tous les comportements de la classe)
- méthode = action (procédure, fonction) que l'on peut effectuer sur un objet
- envoi de message = demande d'exécution d'une méthode

Slide 4

Extension objet en O'Caml

- extension objet ≠ langage objet
- langage à classes
- sans surcharge
- avec héritage multiple
- et classes paramétrées
- sous-typage ≠ sous-classes

Seul langage avec extension objet, statiquement typé avec inférence de types!!!

Attention: partie du langage en mutation.

Slide 5

Classes

Déclaration d'une classe:

class [**virtual**] nom [p_1 p_2 ... p_n] =

object [(p)]

inherit autre_classe [p_i p_j]

constraint typeexpr = typeexpr

val [**mutable**] ident = expr

initializer expr

method [**private**] [**virtual**] nom_methode = expr

end

Slide 6

Classe `Point`

```
class point (x_init,y_init) =
object
  val mutable x = x_init
  val mutable y = y_init
  method get_x = x
  method get_y = y
  method moveto (a,b) =  begin x <- a; y <- b end
  method rmoveto (dx,dy) =
    begin x <- x + dx; y <- y + dy end
  method to_string () =  "( "^(string_of_int  x)^
                         ", "^(string_of_int y)^")"
  method distance () = sqrt(float(x*x + y*y))
end;;
```

Slide 7

Qu'infère O'Caml?

2 choses:

- 1 abbréviation d'un type `object`
- 1 fonction de construction à utiliser avec `new`

```
class point :  int * int ->
  object
    val mutable x : int
    val mutable y : int
    method distance : unit -> float
    method get_x : int
    method get_y : int
    method moveto : int * int -> unit ...
    method to_string : unit -> string   end
```

Slide 8

Objets et égalité

Un objet est une valeur d'une classe, appelée instance de cette classe.

Cette instance est créée par le constructeur d'objets **new** à qui on indique la classe et les valeurs d'initialisation.

```
# let p1 = new point (0,0);;
val p1 : point = <obj>
# let p2 = new point (3,4);;
val p2 : point = <obj>
# let p3 = new point (0,0);;
val p3 : point = <obj>
# p1 == p3;;
- : bool = false
# p1 = p3;;
- : bool = false
```

Slide 9

Envoi de messages

Un objet sait répondre à un envoi de message du nom d'une méthode de sa classe suivi des paramètres du bon type.

On utilise la notation `#` :

```
# p1#get_x;;              - : int = 0
# p2#get_y;;              - : int = 4
# p1#to_string();;        - : string = "( 0, 0)"
# p2#to_string();;        - : string = "( 3, 4)"
# if (p1#distance()) = (p2#distance())
    then print_string ("c'est le hasard\n")
    else print_string ("on pouvait parier\n");;
on pouvait parier
```

Slide 10

Type des instances

Le type inféré pour les instances `p1` et `p2` est le type objet (`<obj> point`). C'est une abréviation du type objet long suivant :

```
point =
  < distance : unit -> float; get_x : int; get_y : int;
    moveto : int * int -> unit; rmoveto : int * int -> unit;
    to_string : unit -> unit; >
```

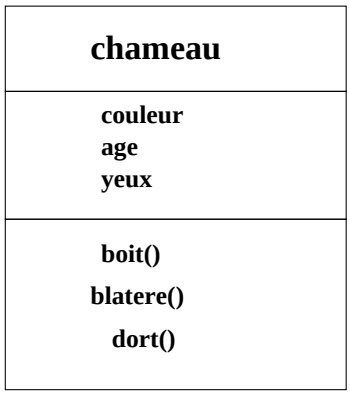
correspondant aux types de ses méthodes.

Typage statique: garantie que les requêtes (envois de message) pourront être traitées.

Slide 11

Notation graphique des classes

Les classes se notent par un rectangle constitué de trois parties. Une partie portant le nom de la classe. Une autre où figure les attributs d'une instance de la classe. Enfin une dernière où sont inscrites les méthodes d'une instance de la classe.



Slide 12

Relations entre objets

- relation d'agrégation : **Has-a**

  ex1: C_1 a un champs de C_2

  ex2: C_1 a de 0 à n champs de C_2,

- relation d'héritage : **Is-a**

  ex3: SC est sous-classe de C

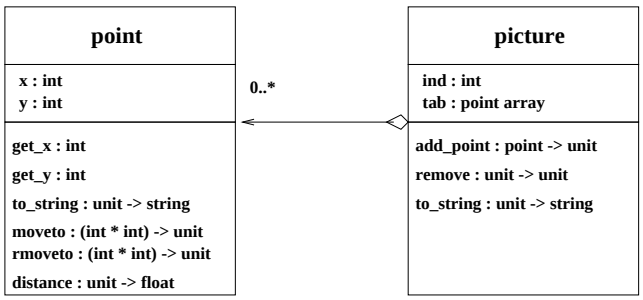
  C'est l'avantage majeur de la programmation objet que de pouvoir étendre le comportement d'une classe existante tout en continuant à utiliser le code écrit par la classe originale. Quand on étend une classe, la nouvelle classe hérite de tous les champs, de données et de méthodes, de la classe qu'elle étend.

Slide 13

Exemple d'agrégat

classe `picture`**:**

- possède entre 0 à n instances de `point`



Slide 14

Code de la classe `picture`

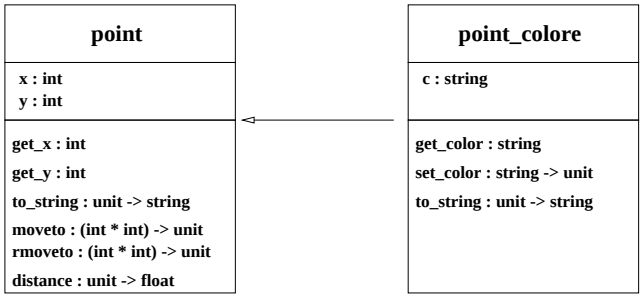
```
class picture n =
object
  val mutable ind = 0
  val tab = Array.create n (new point(0,0))
  method add p =  if (ind < n -1) then  begin
       tab.(ind)<-p; ind <- ind + 1 end
    else failwith ("picture.add:ind ="^(string_of_int ind))
  method remove () = if (ind > 0) then ind <-ind-1
  method to_string () = let s = ref "" in
      for i=0 to ind do s:= !s^" "^tab.(i)#to_string() done;
      !s
end;;
```

Slide 15

Exemple d'héritage

classe `point_colore+`**:**

- hérite de la classe `point`



Slide 16

Code de la classe `point_colore`

```
class point_colore p c =
object
  inherit point p
  val mutable c = c
  method get_color = c
  method set_color nc = c <- nc
  method to_string () =
      "( " ^ (string_of_int x) ^
      ", " ^ (string_of_int y) ^ ")"^
      " de couleur " ^ c
end;;
```

Slide 17

Référencement : self et super

```
class point_colore p c  =
object(self)
  inherit point p as super
  val c = c
  method get_color = c
  method to_string () =
  begin
      super#to_string() ^" de couleur "^ self#get_color
  end
end;;
```

Les noms sont libres, mais on utilisera `this` ou `self` pour soi-même et `super` pour la classe ancêtre.

Slide 18

Liaison retardée

- liaison "retardée" (ou liaisons "dynamique") :

  détermination à l'exécution de la méthode à utiliser lors de l'envoi d'un message

- liaison "précoce" (ou liaison "statique") :

  effectue cette résolution à la compilation

O'Caml implante la liaison retardée!!!

Slide 19

Recherche d'une méthode

Le typage statique garantit que l'envoi d'un message correspondra bien à l'appel d'une méthode de même nom et de même signature.

Par contre le choix de la méthode à exécuter s'effectue à l'exécution et s'explique de la manière suivante :

- recherche dans la table des méthodes de la classe de l'instance
  - si elle apparait, l'exécuter, avec les paramètres du message
  - sinon, effectuer cette recherche au niveau des méthodes définies dans la classe ancêtre

Slide 20

Exemple de liaison retardée

On modifie la méthode `distance` de la classe `point` :

```
method distance () = sqrt(float(self#get_x*self#get_x) +.
                          float(self#get_y*self#get_y) +. )
```

et on redéfinit la méthode `get_x` de `point_colore` :

```
  method get_x = x * 2
```

Alors

```
(new point_colore (2,3) "bleu")#distance()
```

retourne la valeur $5.0 \neq \sqrt{4+9}$

$\Rightarrow$ permettant de modifier le comportement de méthodes héritées.

Slide 21

Initialisation

initialiseur: méthode particulière déclenchée immédiatement après la construction de l'objet.

```
class point (x_init,y_init) =
object  ...
initializer print_string "Creation d'un point";
              print_newline(); flush stdout
end;;
class point_colore p c =
object
  inherit point p ...
  initializer print_string "Creation d'un point colore";
              print_newline(); flush stdout
end;;
```

Slide 22

Trace d'initialiseurs

L'exécution suivante permet de suivre l'ordre de déclenchement de la construction des objets et de leur initialisateur :

```
# let p = new point;;
val p : int * int -> point = <fun>
# let p = new point (3,4);;
Creation d'un point
val p : point = <obj>
# let pc = new point_colore (3,4) "blanc";;
Creation d'un point
Creation d'un point colore
val pc : point_colore = <obj>
```

Slide 23

Visibilité

Une méthode peut être déclarée `private` :

- n'apparait pas dans l'interface de la classe (donc dans le type d'un objet de cette classe)
- s'hérite et donc utilisable dans la sous-hiérarchie

```
class point (x_init,y_init) =
...
  method private rmoveto (dx,dy) =
  begin x <- x + dx;
        y <- self#get_y + dy
  end
  method step1 = self#rmoveto(1,1)
...
end;;
```

Slide 24

Exemple de méthodes privées

L'interface ne contient donc pas la méthode rmoveto comme le montre l'exemple suivant :

```
# let p = new point (2,3);;
Creation d'un point
val p : point = <obj>
# p#to_string();;
- : string = "( 2, 3)"
# p#step1;;
- : unit = ()
# p#to_string();;
- : string = "(3, 4)"
# p#rmoveto(1,1);;
This expression has type point. It has no method rmoveto
```

Slide 25

Classes et méthodes abstraites

classe abstraite: classe dont certaines méthodes ne possèdent pas de corps.

- ces méthodes sont dites *abstraites*;
- utilisation du mot clé `virtual`.

Si une sous-classe, d'une classe abstraite, redéfinit toutes les méthodes abstraite de l'ancêtre, alors elle devient concrète, sinon elle reste abstraite.

Slide 26

Exemple d'une classe abstraite

```
class virtual graphical_object () =
object(self)
  method virtual to_string : unit -> string
  method display () = print_string (self#to_string())
end;;
```

L'interface calculée est la suivante :

```
class virtual graphical_object :
  unit ->
  object
    method display : unit -> unit
    method virtual to_string : unit -> string
  end
```

Slide 27

Classe rectangle

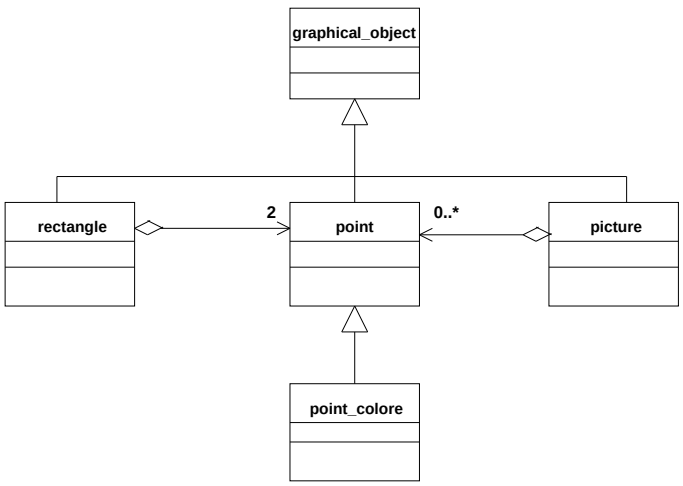
```
class rectangle (p1,p2) =
object
  inherit graphical_object ()
  val mutable llc = (p1 : point)
  val mutable ruc = (p2 : point)
  method to_string()="("^p1#to_string()^","^
                      p2#to_string()^")"        end;;
```

d'interface

```
class rectangle :
  point * point ->  object
    val mutable llc : point
    val mutable ruc : point
    method to_string : unit -> string  end
```

Slide 28

Diagramme des relations de classes



Slide 29

Objets et types

Le type d'un objet est le type de ses méthodes. Par exemple le type point est une abréviation du type :

```
point =<distance: unit -> float; get_x: int; get_y: int;
        moveto: int * int -> unit; rmoveto: int*int-> unit;
        to_string: unit -> string >
```

Lors d'un envoi de message l'inférence de types peut construire un type objet ouvert :

```
# let f x = x#get_x;;
val f : < get_x : 'a; .. > -> 'a = <fun>
# let p = new point(2,3);;
val p : point = <obj>
# f p;;                          - : int = 2
```

Slide 30

Types ouverts

type ouvert: est représenté par la notation `< ..>`,

pour passer d'un type objet fermé un type objet ouvert, on utilisera alors la notation `#type_obj` comme dans l'exemple suivant :

```
# let g (x : #point) = x#amess;;
val g :
<amess: 'a; distance: unit -> float; get_x: int; get_y: int;
 moveto: int * int -> unit; to_string: unit -> string;
 rmoveto : int * int -> unit; .. > ->  'a = <fun>
```

où la coercion de type avec `#point` force `x` à avoir au moins toutes les méthodes de `point`, et l'envoi du message `amess` ajoute une méthode au type du paramètre `x`.

Héritage multiple

Slide 31

L'héritage multiple permet d'hériter des champs de données et des méthodes de plusieurs classes.

En cas de noms de champs ou de méthodes identiques, seulement la dernière déclaration, dans l'ordre de la déclaration de l'héritage, sera conservée.

Les différentes classes héritées n'ont pas forcément de liens d'héritage entre elles.

Intérêt: augmenter la réutilisabilité des classes.

Exemple d'héritage multiple

Slide 32

On définit la classe abstraite `geometric_object` qui déclare deux méthodes `compute_area` et `compute_circ` pour le calcul de la surface et du périmètre.

```
class virtual geometric_object () =
object
  method virtual compute_area : unit -> float
  method virtual compute_circ : unit -> float
end;;
```

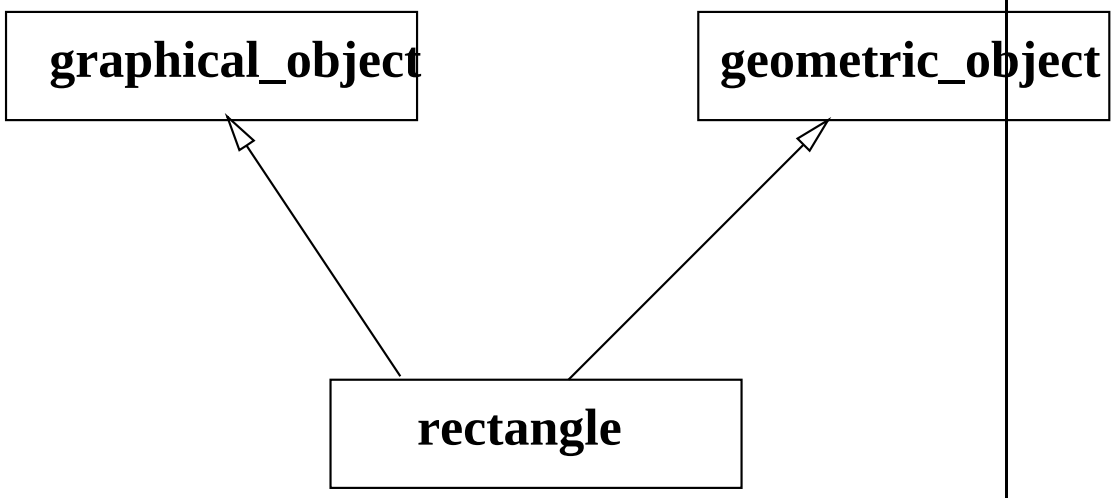
Slide 33

Nouvelle classe rectangle

```
class rectangle (p1,p2) =
object
  inherit graphical_object ()
  inherit geometric_object ()
  val mutable llc = (p1 : point)
  val mutable ruc = (p2 : point)
  method to_string()="("^p1#to_string()^","^
                    p2#to_string()^")"
  method compute_area() = abs(ruc#get_x - llc#get_x) *
                          abs(ruc#get_t - llc#get_y)
  method compute_circ() = (abs(ruc#get_x - llc#get_x) +
                          abs(ruc#get_t - llc#get_y)) * 2
end;;
```

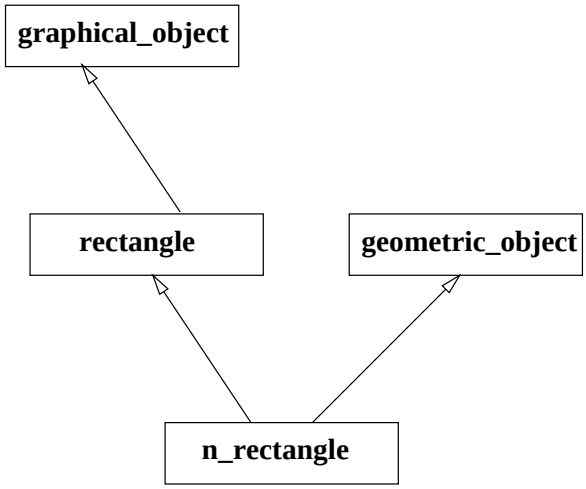
Slide 34

Modélisation de l'héritage multiple



Autres modélisations:

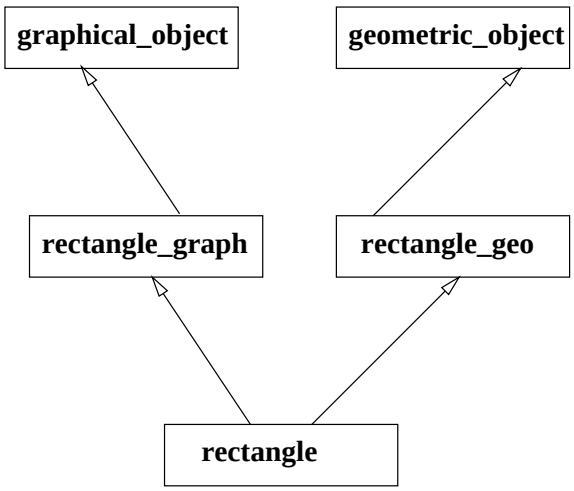
Slide 35



Dans ce cas là, seules les méthodes abstraites de la classe abstraite `geometric_object` auraient du être définies dans `n_rectangle`.
Toujours dans la même veine, les développement de la hiérarchie `graphical_object` et `geometric_object` auraient pu être séparés

Slide 36

jusqu'au moment il devenait utile d'avoir une classe possèdant les deux comportements :



Slide 37

Classes paramétrées

Utilisation: du polymorphisme paramétrique dans les classes

Intérêt: augmente la généricité du code

```
class ['a,'b] pair (a:'a) (b:'b) =
  object
    val x = a
    val y = b
    method fst = x
    method snd = y
  end;;
# let v = new pair 3 true;;
val v : (int, bool) pair = <obj>
```

Slide 38

Classe paramétrée `Pile`

```
class ['a] pile ((x:'a),n) =
object(self)
 val mutable ind = 0
 val tab = Array.create n x
 method is_empty () = if ind = 0 then true else false
 method private is_full () =
   if ind = n+1 then true else false
 method pop() = if self#is_empty() then failwith "pile vide"
                else ind <- ind -1 ; tab.(ind)
 method push y =
   if self#is_full() then failwith "pile pleine"
   else tab.(ind) <- y; ind <- ind + 1
end;;
```

Slide 39

Utilisation de la classe `pile`

```
# let pi = new pile (0.0,10);;
val pi : float pile = <obj>
# pi#push(3.14);;
- : unit = ()
# let ps = new pile ("hello", 20);;
val ps : string pile = <obj>
# ps#push("hello");;
- : unit = ()
# let pp = new pile (new point(0,0),10);;
val pp : point pile = <obj>
# pp#push(new point(4,5));;
- : unit = ()
```

Slide 40

Contraintes de typage

Selon l'usage de la valeur du type paramétré, des contraintes de typage peuvent apparaître dans l'interface inférée. On cherche à construire des listes paramétrées sans utiliser de vecteurs. Pour cela on définit une classe abstraite `['a] liste` ainsi que deux sous-classes : `['a] cons` et `['a] nil` de la manière suivante :

```
class virtual ['a] liste () =
object
  method virtual empty : unit -> bool
  method virtual cons  : 'a -> unit
  method virtual head :  'a
  method virtual tail : 'a liste
  method virtual display : unit -> unit
end;;
```

Slide 41

Sous-classes de `liste`

```
class  ['a] cons (v,l)  =
object (self)
  inherit ['a] liste ()
  val mutable car = (v:'a)
  val mutable cdr = (l:'a liste)
  method empty () = false
  method cons x = cdr<-new cons(car,cdr); car<-x
  method head = car
  method tail = cdr
  method display () =
     car#print(); print_string "  ::";
     self#tail#display()
end;;
```

Slide 42

Contrainte de type implicite

contrainte inférée

```
class ['a] cons :
  'a * 'a liste ->
  object
    constraint 'a = < print : unit -> 'b; .. >
    val mutable car : 'a
    val mutable cdr : 'a liste
    method cons : 'a -> unit
    method display : unit -> unit
    method empty : unit -> bool
    method head : 'a
    method tail : 'a liste
  end
```

Slide 43

```
exception ListeVide;;
class  ['a] nil () =
object (self)
  inherit ['a] liste ()
  val nil = ()
  method empty () = true
  method cons (x:'a) = failwith "bad argument"
  method head = raise ListeVide
  method tail = raise ListeVide
  method display () = print_string "[]"
end;;
```

Slide 44

Constrainte de type explicite

```
class virtual printable ()  =
object
  method virtual print : unit -> unit
end;;


class  ['a] cons (v,l)  =
object (self)
  inherit ['a] liste ()
  constraint 'a = #printable
  val mutable car = (v:'a)
```

Slide 45

Utilisation des listes

On définit une classe `integer` possèdant une méthode `print`

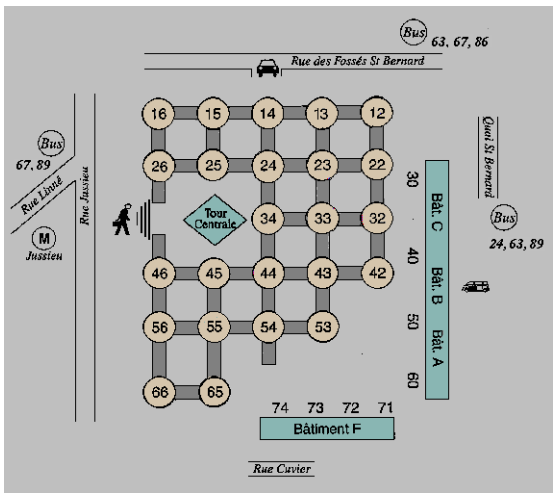
```
class integer i = object  val v = i
  method get = v   method print () = print_int v end;;
```

La construction de liste est la suivante :

```
# let i1 = new integer 1;;
val i1 : integer = <obj>
# let i2 = new integer 2;;
val i2 : integer = <obj>
# let n = new nil ();;
val n : '_a nil = <obj>
# let l = new cons (i1,n);;
val l : integer liste = <obj>
# l#display();;              1  ::[]- : unit = ()
```

Slide 46

Pause



Slide 47

Sous-typage

sous-typage: est la possibilité pour un objet d'un certain type d'être considéré et utilisé comme un objet d'un autre type.

Notation: La relation "est un sous-type de" se note `:>` . On note que `point_colore` est un sous-type de `point` de la manière suivante :

```
point_colore :> point
```

Si le membre gauche de la relation est omis, alors c'est le type de la valeur qui sera considéré comme membre gauche.

La relation de sous-typage, combinée avec la liaison tardive, introduit une nouvelle forme de polymorphisme :
le polymorphisme d'**inclusion**.

Slide 48

Exemple de sous-typage

Soient les déclarations suivantes :

```
# let p = new point (4,5);;
val p : point = <obj>
# let pc = new point_colore (4,5) "blanc";;
val pc : point_colore = <obj>
# let np = (pc :> point);;
val np : point = <obj>
# let np2 = (pc : point_colore :> point);;
val np2 : point = <obj>
```

Slide 49

Exemple de polymorphisme d'inclusion

Invocation: de la méthode `to_string`

```
# p#to_string();;
- : string = "( 4, 5)"
# pc#to_string();;
- : string = "( 4, 5) de couleur blanc"
# np#to_string();;
- : string = "( 4, 5) de couleur blanc"
```

où l'envoi d'un message `to_string` sur `np`, valeur considérée de type `point` déclenche la méthode `to_string` de la classe `point_colore`.

Slide 50

Exemple

Construction: d'une liste de points

```
# let l = [p; np];;
val l : point list = [<obj>; <obj>]
# List.map (fun x -> x#to_string()) l;;
- : string list = ["( 4, 5)";
            "( 4, 5) de couleur blanc"]
```

Cela vient de la liaison tardive (choix de la méthode à utiliser à l'exécution).

Slide 51

Sous-typage ≠ héritage

2 arguments:

- on peut être sous-type sans héritage

  il est possible de forcer un type classe dans un autre type classe sans que le premier corresponde à un descendant du deuxième

- on peut hériter sans être sous-type

  cela arrive quand une des méthodes de la classe ancêtre a un paramètre du type de la classe

Slide 52

Sous-typage entre objets

Soient $t =< m_1 : \tau_1; \ldots; m_n : \tau_n >$ et $t' =< m_1 : \sigma_1; \ldots; m_n : \sigma_n; \tau' >$ où τ' est une suite de méthodes,
on dit que t' est un sous-type de t dans C (contexte de typage), noté $t' \leq t$
si $\sigma_i \leq \tau_i$ pour $i \in \{1, \ldots, n\}$.

appel de fonctions: Si $f : \sigma \to \tau$ dans C, $a : \sigma'$ dans C et $\sigma' \leq \sigma$ dans C
alors (fa) est bien typé dans C et a le type τ.

Une fonction f qui attend un argument de type σ peut recevoir sans danger un argument d'un sous-type de σ.

Slide 53

sous-typage de types fonctionnels (1)

Si on définit les classes suivantes :

```
class a =
 ...
  method f : t1 -> t2
 ...
end;;
class b =
 ...
  method f : t3 -> t4
 ...
end;;
```

Si on veut montrer que $b \leq a$ alors il faut vérifier
$(t_3 \rightarrow t_4) \leq (t_1 \rightarrow t_2)$.

Slide 54

sous-typage de types fonctionnels (2)

Pour distinguer les deux méthodes f on les nomme : f_a et f_b.

Soient $t_1 \rightarrow t_2$ et $t_3 \rightarrow t_4$ deux types fonctionnels, ils sont en relation de sous-typage:

$(t_3 \rightarrow t_4) \leq (t_1 \rightarrow t_2)$
si et seulement si :

- $t_4 \leq t_2$ (co − variance)
- $t_1 \leq t_3$ (contra − variance)

Slide 55

Justification

Soient les 2 fonctions suivantes bien typées :

```
let g (p : t2) = ...
let h ((o:a),(x:t1)) = g(o#f(x));;
```

avec

```
g : t2 -> nt
h : ( a * t1) -> nt
```

1. co-variance : la fonction g attend un argument de type t_2 ou d'un de ses sous-types. Comme cet argument est dans le corps de h résultat de l'envoi du message $f(x)$, il peut être résultat de l'appel de f_b, donc :

$$type\_res(f_b) \leq type\_res(f_a) \Rightarrow t_4 \leq t_2$$

Slide 56

2. contra-variance : En appliquant f à une instance de b (notée o_b on obtient :

$$h(o_b, x) \Rightarrow \mathrm{g}(o_b \# f_b(x))$$

Le type de x est t_1 (type des arguments de f_a, mais il doit pouvoir être passé comme argument de f_b (de type t_3, donc

$$(type\_arg(f_a) = type(x) = t_1) \leq type\_arg(f_b) \Rightarrow t_1 \leq t_3$$

La relation $t_3 \leq t_1$ est impossible car alors f_b ne pourrait recevoir un argument de type t_1 et l'appel $h(o_b, r)$ avec r de type t_1 serait alors incorrect.

Slide 57

Exemples

En reprenant l'exemple sur les `point` et `point_colore` précédent, on obtient :

$$eq_{point} : point \to bool \quad eq_{point\_colore} : point\_colore \to bool$$

et on s'aperçoit alors que pour que

$$point\_colore \leq point$$

il faudrait que

$$(point\_colore \to bool) \leq (point \to bool)$$

c'est à dire, avec la relation de contra-variance des types fonctionnels

$$point \leq point\_colore$$

Slide 58

Style fonctionnel

Le style de la programmation objet est le plus souvent impératif. Un message est envoyé à un objet qui modifie physiquement son état interne (ses champs de données). Néanmoins il est aussi possible d'aborder la programmation objet par le style fonctionnel. L'envoi d'un message à un objet retourne un nouvel objet.

Copie d'objets: On utilise pour cela l'annotation `{< ... >}` qui retourne une copie de l'objet (self) dans lequel les valeurs de certains champs de données sont changées.

La fonction `Oo.copy` retourne une copie d'un objet. Son type est le suivant :

```
(< .. > as 'a) -> 'a
```

Slide 59

Exemple en style fonctionnel

```
class point (x_init,y_init) = object
  val x = x_init
  val y = y_init
  method moveto (a,b) = {<x=a; y=b>}
  method rmoveto (dx,dy) = {<x=x+dx;y=y+dy>}
  method to_string () =  "( "^(string_of_int  x)^
                         ", "^(string_of_int y)^")" end;;
```

```
# let p = new point (2,3);;
val p : point = <obj>
# (p#rmoveto(10,10))#to_string();;
- : string = "( 12, 13)"
# p#to_string();;
- : string = "( 2, 3)"
```

Slide 60

Interfaces (1)

L'interface `interf_point` est déclarée de la manière suivante :

```
class type interf_point  =
object
  method get_x : int
  method get_y : int
  method moveto : (int * int ) -> unit
  method rmoveto : (int * int )  -> unit
  method print : unit -> unit
  method distance : unit -> float
end;;
```

Slide 61

Interfaces (2)

Celle-ci peut dont être utilisée pour coercer le type d'une définition de classe.

```
# let f (x:interf_point) = x;;
val f : interf_point -> interf_point = <fun>
```

Une interface ne masque que les variables d'instance et les méthodes privés.

Intérêt: construire des interfaces de classes sans avoir à définir la classe pour les modules

Slide 62

Retour sur les classes

Déclarations de classes

class nom = function p1 -> ... -> function pn -> object end;;

avec possibilité de déclarations locales :

```
class titi =
  let x =  ref 0 in fun y ->
    object(self)
      val mutable z = 0
      val nom = (y : string)
      method gensym () = z<-z+1;
        y^"_"^(string_of_int !x)^"_"^(string_of_int z)
      initializer incr x
    end;;
```

Slide 63

Exemple d'exécution

```
# let a = new titi "R";;
val a : titi = <obj>
#  a#gensym();;
- : string = "R_1_1"
#  a#gensym();;
- : string = "R_1_2"
# let b = new titi "T";;
val b : titi = <obj>
#  b#gensym();;
- : string = "T_2_1"
#  b#gensym();;
- : string = "T_2_2"
# a#gensym();;
- : string = "R_2_3"
```

Slide 64

Pause