OCaml: les objets

David Delahaye

Faculté des Sciences
David.Delahaye@lirmm.fr

Licence L3 2018-2019

Plan du cours

3 semaines de cours

1. Noyau fonctionnel ;
2. ***Objets simples (héritage simple et multiple, sous-typage) ;***
3. Objets avancés (types ouverts, contraintes, « self-types »).

Nous allons apprendre OCaml !

Histoire d'OCaml

- 1978 : langage ML (Milner) ;
- 1980 : projet Inria Formel (Huet) ;
- 1985 : « Categorical Abstract Machine » (Cousineau, Curien, Mauny) ;
- 1987 : première release de Caml (Suarez) ;
- 1988-1992 : Caml prend de l'ampleur (Mauny, Weis) ;
- 1990-1991 : machine Zinc, Caml Light (Leroy, Doligez) ;
- 1995 : ajout des modules, Caml Special Light (Leroy) ;
- 1996 : ajout des objets, Objective Caml (Vouillon, Rémy) ;
- 2000 : merge avec la branche Objective Label (Guarrigue) ;
- 2011 : le nom devient définitivement OCaml.

Premières classes et premiers objets

Dans la boucle interactive

```
# class cell =
object
  val content = 0
  method get = content
end;;
class cell :
  object val content : int method get : int end
# let o = new cell;;
val o : cell = <obj>
# o#get;;
- : int = 0
# o#content;;
Error: This expression has type cell
       It has no method content
```

Premières classes et premiers objets

Dans la boucle interactive

```
# class cell =
object
  val content = 0
  method get = content
end;;
class cell :
  object val content : int method get : int end
# let o = new cell;;
val o : cell = <obj>
# o#get;;
- : int = 0
# o#content;;
Error: This expression has type cell
       It has no method content
```

Premières classes et premiers objets

Dans la boucle interactive

```
# class cell =
object
  val content = 0
  method get = content
end;;
class cell :
  object val content : int method get : int end
# let o = new cell;;
val o : cell = <obj>
# o#get;;
- : int = 0
# o#content;;
Error: This expression has type cell
       It has no method content
```

Premières classes et premiers objets

Dans la boucle interactive

```
# class cell =
object
  val content = 0
  method get = content
end;;
class cell :
  object val content : int method get : int end
# let o = new cell;;
val o : cell = <obj>
# o#get;;
- : int = 0
# o#content;;
Error: This expression has type cell
       It has no method content
```

Premières classes et premiers objets

Dans la boucle interactive

```
# class cell =
object
  val content = 0
  method get = content
end;;
class cell :
  object val content : int method get : int end
# let o = new cell;;
val o : cell = <obj>
# o#get;;
- : int = 0
# o#content;;
Error: This expression has type cell
       It has no method content
```

Premières classes et premiers objets

Dans la boucle interactive

```
# class cell =
object
  val content = 0
  method get = content
end;;
class cell :
  object val content : int method get : int end
# let o = new cell;;
val o : cell = <obj>
# o#get;;
- : int = 0
# o#content;;
Error: This expression has type cell
       It has no method content
```

Premières classes et premiers objets

Dans la boucle interactive

```
# class cell =
object
  val content = 0
  method get = content
end;;
class cell :
  object val content : int method get : int end
# let o = new cell;;
val o : cell = <obj>
# o#get;;
- : int = 0
# o#content;;
Error: This expression has type cell
       It has no method content
```

Premières classes et premiers objets

Dans la boucle interactive

```
# class cell =
object
  val content = 0
  method get = content
end;;
class cell :
  object val content : int method get : int end
# let o = new cell;;
val o : cell = <obj>
# o#get;;
- : int = 0
# o#content;;
Error: This expression has type cell
       It has no method content
```

Premières classes et premiers objets

Classes polymorphes

```
# class ['a] cell (n : 'a) =
object
  val content = n
  method get = content
end;;
class ['a] cell :
  'a → object val content : 'a method get : 'a end
# let o = new cell 1;;
val o : int cell = <obj>
# o#get;;
- : int = 1
# let o = new cell true;;
val o : bool cell = <obj>
# o#get;;
- : bool = true
```

Premières classes et premiers objets

Classes polymorphes

```
# class ['a] cell (n : 'a) =
object
  val content = n
  method get = content
end;;
class ['a] cell :
  'a → object val content : 'a method get : 'a end
# let o = new cell 1;;
val o : int cell = <obj>
# o#get;;
- : int = 1
# let o = new cell true;;
val o : bool cell = <obj>
# o#get;;
- : bool = true
```

Premières classes et premiers objets

Classes polymorphes

```
# class ['a] cell (n : 'a) =
object
  val content = n
  method get = content
end;;
class ['a] cell :
  'a → object val content : 'a method get : 'a end
# let o = new cell 1;;
val o : int cell = <obj>
# o#get;;
- : int = 1
# let o = new cell true;;
val o : bool cell = <obj>
# o#get;;
- : bool = true
```

Premières classes et premiers objets

Classes polymorphes

```
# class ['a] cell (n : 'a) =
object
  val content = n
  method get = content
end;;
class ['a] cell :
  'a → object val content : 'a method get : 'a end
# let o = new cell 1;;
val o : int cell = <obj>
# o#get;;
- : int = 1
# let o = new cell true;;
val o : bool cell = <obj>
# o#get;;
- : bool = true
```

Premières classes et premiers objets

Classes polymorphes

```
# class ['a] cell (n : 'a) =
object
  val content = n
  method get = content
end;;
class ['a] cell :
  'a → object val content : 'a method get : 'a end
# let o = new cell 1;;
val o : int cell = <obj>
# o#get;;
- : int = 1
# let o = new cell true;;
val o : bool cell = <obj>
# o#get;;
- : bool = true
```

Premières classes et premiers objets

Classes polymorphes

```
# class ['a] cell (n : 'a) =
object
  val content = n
  method get = content
end;;
class ['a] cell :
  'a → object val content : 'a method get : 'a end
# let o = new cell 1;;
val o : int cell = <obj>
# o#get;;
- : int = 1
# let o = new cell true;;
val o : bool cell = <obj>
# o#get;;
- : bool = true
```

Premières classes et premiers objets

Classes polymorphes

```
# class ['a] cell (n : 'a) =
object
  val content = n
  method get = content
end;;
class ['a] cell :
  'a → object val content : 'a method get : 'a end
# let o = new cell 1;;
val o : int cell = <obj>
# o#get;;
- : int = 1
# let o = new cell true;;
val o : bool cell = <obj>
# o#get;;
- : bool = true
```

Premières classes et premiers objets

Classes polymorphes

```
# class ['a] cell (n : 'a) =
object
  val content = n
  method get = content
end;;
class ['a] cell :
  'a → object val content : 'a method get : 'a end
# let o = new cell 1;;
val o : int cell = <obj>
# o#get;;
- : int = 1
# let o = new cell true;;
val o : bool cell = <obj>
# o#get;;
- : bool = true
```

Premières classes et premiers objets

Classes polymorphes

```
# class ['a] cell (n : 'a) =
object
  val content = n
  method get = content
end;;
class ['a] cell :
  'a → object val content : 'a method get : 'a end
# let o = new cell 1;;
val o : int cell = <obj>
# o#get;;
- : int = 1
# let o = new cell true;;
val o : bool cell = <obj>
# o#get;;
- : bool = true
```

Premières classes et premiers objets

Classes polymorphes

```
# class ['a] cell (n : 'a) =
object
  val content = n
  method get = content
end;;
class ['a] cell :
  'a → object val content : 'a method get : 'a end
# let o = new cell 1;;
val o : int cell = <obj>
# o#get;;
- : int = 1
# let o = new cell true;;
val o : bool cell = <obj>
# o#get;;
- : bool = true
```

Premières classes et premiers objets

Constructeurs fonctionnels, application partielle

```
# let f = new cell;;
val f : 'a → 'a cell = <fun>
# let o = f 1;;
val o : int cell = <obj>
# o#get;;
- : int = 1
# let o = f true;;
val o : bool cell = <obj>
# o#get;;
- : bool = true
```

Premières classes et premiers objets

Constructeurs fonctionnels, application partielle

```
# let f = new cell;;
val f : 'a → 'a cell = <fun>
# let o = f 1;;
val o : int cell = <obj>
# o#get;;
- : int = 1
# let o = f true;;
val o : bool cell = <obj>
# o#get;;
- : bool = true
```

Premières classes et premiers objets

Constructeurs fonctionnels, application partielle

```
# let f = new cell;;
val f : 'a → 'a cell = <fun>
# let o = f 1;;
val o : int cell = <obj>
# o#get;;
- : int = 1
# let o = f true;;
val o : bool cell = <obj>
# o#get;;
- : bool = true
```

Premières classes et premiers objets

Constructeurs fonctionnels, application partielle

```
# let f = new cell;;
val f : 'a → 'a cell = <fun>
# let o = f 1;;
val o : int cell = <obj>
# o#get;;
- : int = 1
# let o = f true;;
val o : bool cell = <obj>
# o#get;;
- : bool = true
```

Premières classes et premiers objets

Constructeurs fonctionnels, application partielle

```
# let f = new cell;;
val f : 'a → 'a cell = <fun>
# let o = f 1;;
val o : int cell = <obj>
# o#get;;
- : int = 1
# let o = f true;;
val o : bool cell = <obj>
# o#get;;
- : bool = true
```

Premières classes et premiers objets

Constructeurs fonctionnels, application partielle

```
# let f = new cell;;
val f : 'a → 'a cell = <fun>
# let o = f 1;;
val o : int cell = <obj>
# o#get;;
- : int = 1
# let o = f true;;
val o : bool cell = <obj>
# o#get;;
- : bool = true
```

Premières classes et premiers objets

Constructeurs fonctionnels, application partielle

```
# let f = new cell;;
val f : 'a → 'a cell = <fun>
# let o = f 1;;
val o : int cell = <obj>
# o#get;;
- : int = 1
# let o = f true;;
val o : bool cell = <obj>
# o#get;;
- : bool = true
```

Premières classes et premiers objets

Constructeurs fonctionnels, application partielle

```
# let f = new cell;;
val f : 'a → 'a cell = <fun>
# let o = f 1;;
val o : int cell = <obj>
# o#get;;
- : int = 1
# let o = f true;;
val o : bool cell = <obj>
# o#get;;
- : bool = true
```

Premières classes et premiers objets

Constructeurs fonctionnels, application partielle

```
# let f = new cell;;
val f : 'a → 'a cell = <fun>
# let o = f 1;;
val o : int cell = <obj>
# o#get;;
- : int = 1
# let o = f true;;
val o : bool cell = <obj>
# o#get;;
- : bool = true
```

Premières classes et premiers objets

Constructeurs fonctionnels, application partielle

```
# let f = new cell;;
val f : 'a → 'a cell = <fun>
# let o = f 1;;
val o : int cell = <obj>
# o#get;;
- : int = 1
# let o = f true;;
val o : bool cell = <obj>
# o#get;;
- : bool = true
```

Premières classes et premiers objets

Types de classes

```
# class type ['a] cell_type =
object
  method get : 'a
end;;
class type ['a] cell_type = object method get : 'a end
# class ['a] cell n : ['a] cell_type =
object
  val content = n
  method get = content
end;;
class ['a] cell : 'a -> ['a] cell_type
```

Premières classes et premiers objets

Types de classes

```
# class type ['a] cell_type =
object
  method get : 'a
end;;
class type ['a] cell_type = object method get : 'a end
# class ['a] cell n : ['a] cell_type =
object
  val content = n
  method get = content
end;;
class ['a] cell : 'a → ['a] cell_type
```

Premières classes et premiers objets

Types de classes

```
# class type ['a] cell_type =
object
  method get : 'a
end;;
class type ['a] cell_type = object method get : 'a end
# class ['a] cell n : ['a] cell_type =
object
  val content = n
  method get = content
end;;
class ['a] cell : 'a → ['a] cell_type
```

Premières classes et premiers objets

Types de classes

```
# class type ['a] cell_type =
object
  method get : 'a
end;;
class type ['a] cell_type = object method get : 'a end
# class ['a] cell n : ['a] cell_type =
object
  val content = n
  method get = content
end;;
class ['a] cell : 'a → ['a] cell_type
```

Premières classes et premiers objets

Valeurs mutables

```
# class ['a] cell (n : 'a) =
object
  val mutable content = n
  method get = content
  method set n = content <- n
end;;
class ['a] cell :
  'a →
  object val mutable content : 'a
    method get : 'a method set : 'a → unit end
```

Premières classes et premiers objets

Valeurs mutables

```
# class ['a] cell (n : 'a) =
object
  val mutable content = n
  method get = content
  method set n = content <- n
end;;
class ['a] cell :
  'a →
  object val mutable content : 'a
    method get : 'a method set : 'a → unit end
```

Premières classes et premiers objets

Valeurs mutables

```
# let o = new cell 0;;
val o : int cell = <obj>
# o#get;;
- : int = 0
# o#set 1;;
- : unit = ()
# o#get;;
- : int = 1
```

Premières classes et premiers objets

Valeurs mutables

```
# let o = new cell 0;;
val o : int cell = <obj>
# o#get;;
- : int = 0
# o#set 1;;
- : unit = ()
# o#get;;
- : int = 1
```

Premières classes et premiers objets

Valeurs mutables

```
# let o = new cell 0;;
val o : int cell = <obj>
# o#get;;
- : int = 0
# o#set 1;;
- : unit = ()
# o#get;;
- : int = 1
```

Premières classes et premiers objets

Valeurs mutables

```
# let o = new cell 0;;
val o : int cell = <obj>
# o#get;;
- : int = 0
# o#set 1;;
- : unit = ()
# o#get;;
- : int = 1
```

Premières classes et premiers objets

Valeurs mutables

```
# let o = new cell 0;;
val o : int cell = <obj>
# o#get;;
- : int = 0
# o#set 1;;
- : unit = ()
# o#get;;
- : int = 1
```

Premières classes et premiers objets

Valeurs mutables

```
# let o = new cell 0;;
val o : int cell = <obj>
# o#get;;
- : int = 0
# o#set 1;;
- : unit = ()
# o#get;;
- : int = 1
```

Premières classes et premiers objets

Valeurs mutables

```
# let o = new cell 0;;
val o : int cell = <obj>
# o#get;;
- : int = 0
# o#set 1;;
- : unit = ()
# o#get;;
- : int = 1
```

Premières classes et premiers objets

Valeurs mutables

```
# let o = new cell 0;;
val o : int cell = <obj>
# o#get;;
- : int = 0
# o#set 1;;
- : unit = ()
# o#get;;
- : int = 1
```

Types des objets et égalité entre types d'objet

Principe

- Type d'un objet = type de toutes les méthodes de l'objet ;
- Les variables d'instance ne sont pas considérées ;
- Égalité entre types d'objet structurel ;
- Deux types d'objets sont égaux si et seulement si :
  - Les deux objets ont les mêmes méthodes avec les mêmes noms et les mêmes types.
- Note : contrairement au type d'une fonction, un type d'objet ne contient plus aucune variable de type.

Types des objets et égalité entre types d'objet

Égalité structurelle

```
# class ['a] cell (n : 'a) =
object
  val mutable content = n
  method get = content
  method set n = content <- n
end;;
class ['a] cell :
  'a →
  object val mutable content : 'a
    method get : 'a method set : 'a → unit end
```

Types des objets et égalité entre types d'objet

Égalité structurelle

```
# class ['a] cell (n : 'a) =
object
  val mutable content = n
  method get = content
  method set n = content <- n
end;;
class ['a] cell :
  'a →
  object val mutable content : 'a
    method get : 'a method set : 'a → unit end
```

Types des objets et égalité entre types d'objet

Égalité structurelle

```
# class ['a] box l (n : 'a) =
object
  val name = "Name: " ^ l
  val mutable content = n
  method get = content
  method set n = content <- n
end;;
```

Types des objets et égalité entre types d'objet

Égalité structurelle

```
class ['a] box :
  string →
  'a →
  object
    val mutable content : 'a
    val name : string
    method get : 'a
    method set : 'a → unit
  end
```

Types des objets et égalité entre types d'objet

Égalité structurelle

```
# let c = new cell 1;;
val c : int cell = <obj>
# let b = new box "Integer" 2;;
val b : int box = <obj>
# let l = [c; b];;
val l : int cell list = [<obj>; <obj>]
# List.map (fun o → o#get) l;;
- : int list = [1; 2]
```

Dans cet exemple

- `int cell` = < get : int ; set : int → unit > = int box ;
- OCaml ne fait aucune différence entre les deux types ;
- La liste `l` a le type `int cell` ou `int box` indifféremment (OCaml choisit `int cell` car il type `c` en premier).

Types des objets et égalité entre types d'objet

Égalité structurelle

```
# let c = new cell 1;;
val c : int cell = <obj>
# let b = new box "Integer" 2;;
val b : int box = <obj>
# let l = [c; b];;
val l : int cell list = [<obj>; <obj>]
# List.map (fun o → o#get) l;;
- : int list = [1; 2]
```

Dans cet exemple

- `int cell` = < get : int ; set : int → unit > = int box ;
- OCaml ne fait aucune différence entre les deux types ;
- La liste `l` a le type `int cell` ou `int box` indifféremment (OCaml choisit `int cell` car il type `c` en premier).

Types des objets et égalité entre types d'objet

Égalité structurelle

```
# let c = new cell 1;;
val c : int cell = <obj>
# let b = new box "Integer" 2;;
val b : int box = <obj>
# let l = [c; b];;
val l : int cell list = [<obj>; <obj>]
# List.map (fun o → o#get) l;;
- : int list = [1; 2]
```

Dans cet exemple

- `int cell` = < get : int ; set : int → unit > = int box ;
- OCaml ne fait aucune différence entre les deux types ;
- La liste `l` a le type `int cell` ou `int box` indifféremment (OCaml choisit `int cell` car il type `c` en premier).

Types des objets et égalité entre types d'objet

Égalité structurelle

```
# let c = new cell 1;;
val c : int cell = <obj>
# let b = new box "Integer" 2;;
val b : int box = <obj>
# let l = [c; b];;
val l : int cell list = [<obj>; <obj>]
# List.map (fun o → o#get) l;;
- : int list = [1; 2]
```

Dans cet exemple

- `int cell` = < get : int ; set : int → unit > = int box ;
- OCaml ne fait aucune différence entre les deux types ;
- La liste `l` a le type `int cell` ou `int box` indifféremment (OCaml choisit `int cell` car il type `c` en premier).

Types des objets et égalité entre types d'objet

Égalité structurelle

```
# let c = new cell 1;;
val c : int cell = <obj>
# let b = new box "Integer" 2;;
val b : int box = <obj>
# let l = [c; b];;
val l : int cell list = [<obj>; <obj>]
# List.map (fun o → o#get) l;;
- : int list = [1; 2]
```

Dans cet exemple

- `int cell` = < get : int ; set : int → unit > = int box ;
- OCaml ne fait aucune différence entre les deux types ;
- La liste `l` a le type `int cell` ou `int box` indifféremment (OCaml choisit `int cell` car il type `c` en premier).

Types des objets et égalité entre types d'objet

Égalité structurelle

```
# let c = new cell 1;;
val c : int cell = <obj>
# let b = new box "Integer" 2;;
val b : int box = <obj>
# let l = [c; b];;
val l : int cell list = [<obj>; <obj>]
# List.map (fun o -> o#get) l;;
- : int list = [1; 2]
```

Dans cet exemple

- `int cell` = < get : int ; set : int → unit > = int box ;
- OCaml ne fait aucune différence entre les deux types ;
- La liste `l` a le type `int cell` ou `int box` indifféremment (OCaml choisit `int cell` car il type `c` en premier).

Types des objets et égalité entre types d'objet

Égalité structurelle

```
# let c = new cell 1;;
val c : int cell = <obj>
# let b = new box "Integer" 2;;
val b : int box = <obj>
# let l = [c; b];;
val l : int cell list = [<obj>; <obj>]
# List.map (fun o → o#get) l;;
- : int list = [1; 2]
```

Dans cet exemple

- `int cell` = < get : int ; set : int → unit > = int box ;
- OCaml ne fait aucune différence entre les deux types ;
- La liste `l` a le type `int cell` ou `int box` indifféremment (OCaml choisit `int cell` car il type `c` en premier).

Types des objets et égalité entre types d'objet

Égalité structurelle

```
# let c = new cell 1;;
val c : int cell = <obj>
# let b = new box "Integer" 2;;
val b : int box = <obj>
# let l = [c; b];;
val l : int cell list = [<obj>; <obj>]
# List.map (fun o → o#get) l;;
- : int list = [1; 2]
```

Dans cet exemple

- `int cell` = < get : int ; set : int → unit > = int box ;
- OCaml ne fait aucune différence entre les deux types ;
- La liste `l` a le type `int cell` ou `int box` indifféremment (OCaml choisit `int cell` car il type `c` en premier).

Types des objets et égalité entre types d'objet

Égalité structurelle

```
# let c = new cell 1;;
val c : int cell = <obj>
# let b = new box "Integer" 2;;
val b : int box = <obj>
# let l = [c; b];;
val l : int cell list = [<obj>; <obj>]
# List.map (fun o → o#get) l;;
- : int list = [1; 2]
```

Dans cet exemple

- `int cell` = < get : int ; set : int → unit > = int box ;
- OCaml ne fait aucune différence entre les deux types ;
- La liste `l` a le type `int cell` ou `int box` indifféremment (OCaml choisit `int cell` car il type `c` en premier).

Héritage (simple)

Clauses d'initialisation

```
# class account b =
object (self)
  val mutable balance = 0.0
  method get = balance
  method deposit a = balance <- balance +. a
  method withdraw a = balance <- balance -. a
  method print = print_float balance; print_newline ()
  initializer self#deposit b
  end;;
```

Héritage (simple)

Clauses d'initialisation

```
class account :
  float →
  object
    val mutable balance : float
    method deposit : float → unit
    method get : float
    method print : unit
    method withdraw : float → unit
  end
```

Héritage (simple)

Clauses d'initialisation

```
  # let o = new account 100.;;
val o : account = <obj>
# o#print;;
100.
- : unit = ()
# o#deposit 50.;;
- : unit = ()
# o#get;;
- : float = 150.
```

Héritage (simple)

Clauses d'initialisation

```
  # let o = new account 100.;;
val o : account = <obj>
# o#print;;
100.
- : unit = ()
# o#deposit 50.;;
- : unit = ()
# o#get;;
- : float = 150.
```

Héritage (simple)

Clauses d'initialisation

```
  # let o = new account 100.;;
val o : account = <obj>
# o#print;;
100.
- : unit = ()
# o#deposit 50.;;
- : unit = ()
# o#get;;
- : float = 150.
```

Héritage (simple)

Clauses d'initialisation

```
  # let o = new account 100.;;
val o : account = <obj>
# o#print;;
100.
- : unit = ()
# o#deposit 50.;;
- : unit = ()
# o#get;;
- : float = 150.
```

Héritage (simple)

Clauses d'initialisation

```
  # let o = new account 100.;;
val o : account = <obj>
# o#print;;
100.
- : unit = ()
# o#deposit 50.;;
- : unit = ()
# o#get;;
- : float = 150.
```

Héritage (simple)

Clauses d'initialisation

```
  # let o = new account 100.;;
val o : account = <obj>
# o#print;;
100.
- : unit = ()
# o#deposit 50.;;
- : unit = ()
# o#get;;
- : float = 150.
```

Héritage (simple)

Clauses d'initialisation

```
  # let o = new account 100.;;
val o : account = <obj>
# o#print;;
100.
- : unit = ()
# o#deposit 50.;;
- : unit = ()
# o#get;;
- : float = 150.
```

Héritage (simple)

Clauses d'initialisation

```
  # let o = new account 100.;;
val o : account = <obj>
# o#print;;
100.
- : unit = ()
# o#deposit 50.;;
- : unit = ()
# o#get;;
- : float = 150.
```

Héritage (simple)

Ajout de méthodes

```
# class interest_account b =
object
  inherit account b
  method interest =
    balance <- balance +. 5. *. balance /. 100.
end;;
```

Héritage (simple)

Ajout de méthodes

```
class interest_account :
  float →
  object
    val mutable balance : float
    method deposit : float → unit
    method get : float
    method interest : unit
    method print : unit
    method withdraw : float → unit
  end
```

Héritage (simple)

Ajout de méthodes

```
# let o = new interest_account 100.;;
val o : interest_account = <obj>
# o#get;;
- : float = 100.
# o#interest;;
- : unit = ()
# o#get;;
- : float = 105.
```

Héritage (simple)

Ajout de méthodes

```
# let o = new interest_account 100.;;
val o : interest_account = <obj>
# o#get;;
- : float = 100.
# o#interest;;
- : unit = ()
# o#get;;
- : float = 105.
```

Héritage (simple)

Ajout de méthodes

```
# let o = new interest_account 100.;;
val o : interest_account = <obj>
# o#get;;
- : float = 100.
# o#interest;;
- : unit = ()
# o#get;;
- : float = 105.
```

Héritage (simple)

Ajout de méthodes

```
# let o = new interest_account 100.;;
val o : interest_account = <obj>
# o#get;;
- : float = 100.
# o#interest;;
- : unit = ()
# o#get;;
- : float = 105.
```

Héritage (simple)

Ajout de méthodes

```
# let o = new interest_account 100.;;
val o : interest_account = <obj>
# o#get;;
- : float = 100.
# o#interest;;
- : unit = ()
# o#get;;
- : float = 105.
```

Héritage (simple)

Ajout de méthodes

```
# let o = new interest_account 100.;;
val o : interest_account = <obj>
# o#get;;
- : float = 100.
# o#interest;;
- : unit = ()
# o#get;;
- : float = 105.
```

Héritage (simple)

Ajout de méthodes

```
# let o = new interest_account 100.;;
val o : interest_account = <obj>
# o#get;;
- : float = 100.
# o#interest;;
- : unit = ()
# o#get;;
- : float = 105.
```

Héritage (simple)

Ajout de méthodes

```
# let o = new interest_account 100.;;
val o : interest_account = <obj>
# o#get;;
- : float = 100.
# o#interest;;
- : unit = ()
# o#get;;
- : float = 105.
```

Héritage (simple)

Redéfinition de méthodes

```
# class secure_account b =
object
  inherit account b as super
  method withdraw a =
    if (balance -. a) >= 0. then super#withdraw a
    else failwith "Not enough money!"
end;;
```

Héritage (simple)

Redéfinition de méthodes

```
class secure_account :
  float →
  object
    val mutable balance : float
    method deposit : float → unit
    method get : float
    method print : unit
    method withdraw : float → unit
  end
```

Héritage (simple)

Redéfinition de méthodes

```
# let o = new secure_account 100.;;
val o : secure_account = <obj>
# o#withdraw 150.;;
Exception: Failure "Not enough money!".
```

Héritage (simple)

Redéfinition de méthodes

```
# let o = new secure_account 100.;;
val o : secure_account = <obj>
# o#withdraw 150.;;
Exception: Failure "Not enough money!".
```

Héritage (simple)

Redéfinition de méthodes

```
# let o = new secure_account 100.;;
val o : secure_account = <obj>
# o#withdraw 150.;;
Exception: Failure "Not enough money!".
```

Héritage (simple)

Redéfinition de méthodes

```
# let o = new secure_account 100.;;
val o : secure_account = <obj>
# o#withdraw 150.;;
Exception: Failure "Not enough money!".
```

Sous-typage

Mettre des comptes hétérogènes dans une liste

```
# let a = new account 100.;;
val a : account = <obj>
# let s = new secure_account 100.;;
val s : secure_account = <obj>
# [a; s];;
- : account list = [<obj>; <obj>]
```

- Aucun problème car les types `account` et `secure_account` sont égaux !

Sous-typage

Mettre des comptes hétérogènes dans une liste

```
# let a = new account 100.;;
val a : account = <obj>
# let s = new secure_account 100.;;
val s : secure_account = <obj>
# [a; s];;
- : account list = [<obj>; <obj>]
```

- Aucun problème car les types `account` et `secure_account` sont égaux !

Sous-typage

Mettre des comptes hétérogènes dans une liste

```
# let a = new account 100.;;
val a : account = <obj>
# let s = new secure_account 100.;;
val s : secure_account = <obj>
# [a; s];;
- : account list = [<obj>; <obj>]
```

- Aucun problème car les types `account` et `secure_account` sont égaux !

Sous-typage

Mettre des comptes hétérogènes dans une liste

```
# let a = new account 100.;;
val a : account = <obj>
# let s = new secure_account 100.;;
val s : secure_account = <obj>
# [a; s];;
- : account list = [<obj>; <obj>]
```

- Aucun problème car les types `account` et `secure_account` sont égaux !

Sous-typage

Mettre des comptes hétérogènes dans une liste

```
# let a = new account 100.;;
val a : account = <obj>
# let s = new secure_account 100.;;
val s : secure_account = <obj>
# [a; s];;
- : account list = [<obj>; <obj>]
```

- Aucun problème car les types `account` et `secure_account` sont égaux !

Sous-typage

Mettre des comptes hétérogènes dans une liste

```
# let a = new account 100.;;
val a : account = <obj>
# let s = new secure_account 100.;;
val s : secure_account = <obj>
# [a; s];;
- : account list = [<obj>; <obj>]
```

- Aucun problème car les types `account` et `secure_account` sont égaux !

Sous-typage

Mettre des comptes hétérogènes dans une liste

```
# let a = new account 100.;;
val a : account = <obj>
# let s = new secure_account 100.;;
val s : secure_account = <obj>
# [a; s];;
- : account list = [<obj>; <obj>]
```

- Aucun problème car les types `account` et `secure_account` sont égaux !

Sous-typage

Mettre des comptes hétérogènes dans une liste

```
# let a = new account 100.;;
val a : account = <obj>
# let i = new interest_account 100.;;
val i : interest_account = <obj>
# [a; i];;
Error: This expression has type interest_account
       but an expression was expected of type account
       The second object type has no method interest
```

- Les types `account` et `interest_account` ne sont pas égaux ;
- Le type `interest_account` possède la méthode `interest` en plus.

Sous-typage

Mettre des comptes hétérogènes dans une liste

```
# let a = new account 100.;;
val a : account = <obj>
# let i = new interest_account 100.;;
val i : interest_account = <obj>
# [a; i];;
Error: This expression has type interest_account
       but an expression was expected of type account
       The second object type has no method interest
```

- Les types `account` et `interest_account` ne sont pas égaux ;
- Le type `interest_account` possède la méthode `interest` en plus.

Sous-typage

Mettre des comptes hétérogènes dans une liste

```
# let a = new account 100.;;
val a : account = <obj>
# let i = new interest_account 100.;;
val i : interest_account = <obj>
# [a; i];;
Error: This expression has type interest_account
       but an expression was expected of type account
       The second object type has no method interest
```

- Les types `account` et `interest_account` ne sont pas égaux ;
- Le type `interest_account` possède la méthode `interest` en plus.

Sous-typage

Mettre des comptes hétérogènes dans une liste

```
# let a = new account 100.;;
val a : account = <obj>
# let i = new interest_account 100.;;
val i : interest_account = <obj>
# [a; i];;
Error: This expression has type interest_account
       but an expression was expected of type account
       The second object type has no method interest
```

- Les types `account` et `interest_account` ne sont pas égaux ;
- Le type `interest_account` possède la méthode `interest` en plus.

Sous-typage

Mettre des comptes hétérogènes dans une liste

```
# let a = new account 100.;;
val a : account = <obj>
# let i = new interest_account 100.;;
val i : interest_account = <obj>
# [a; i];;
Error: This expression has type interest_account
       but an expression was expected of type account
       The second object type has no method interest
```

- Les types `account` et `interest_account` ne sont pas égaux ;
- Le type `interest_account` possède la méthode `interest` en plus.

Sous-typage

Mettre des comptes hétérogènes dans une liste

```
# let a = new account 100.;;
val a : account = <obj>
# let i = new interest_account 100.;;
val i : interest_account = <obj>
# [a; i];;
Error: This expression has type interest_account
       but an expression was expected of type account
       The second object type has no method interest
```

- Les types `account` et `interest_account` ne sont pas égaux ;
- Le type `interest_account` possède la méthode `interest` en plus.

Sous-typage

Mettre des comptes hétérogènes dans une liste

```
# let a = new account 100.;;
val a : account = <obj>
# let i = new interest_account 100.;;
val i : interest_account = <obj>
# [a; i];;
Error: This expression has type interest_account
       but an expression was expected of type account
       The second object type has no method interest
```

- Les types `account` et `interest_account` ne sont pas égaux ;
- Le type `interest_account` possède la méthode `interest` en plus.

Sous-typage

Utiliser le sous-typage explicite

```
# [a; (i:interest_account:>account)];;
- : account list = [<obj>; <obj>]
# [a; (i:>account)];;
- : account list = [<obj>; <obj>]
```

- Possible si `interest_account` est un sous-type de `account` ;
- Dans la liste, l'objet `i` possède toujours la méthode `interest` mais elle ne peut plus être utilisée ;
- Du point de vue de la sûreté d'exécution, c'est incorrect de vouloir l'utiliser et ça n'est donc pas une contrainte !

Sous-typage

Utiliser le sous-typage explicite

```
# [a; (i:interest_account:>account)];;
- : account list = [<obj>; <obj>]
# [a; (i:>account)];;
- : account list = [<obj>; <obj>]
```

- Possible si `interest_account` est un sous-type de `account` ;
- Dans la liste, l'objet `i` possède toujours la méthode `interest` mais elle ne peut plus être utilisée ;
- Du point de vue de la sûreté d'exécution, c'est incorrect de vouloir l'utiliser et ça n'est donc pas une contrainte !

Sous-typage

Utiliser le sous-typage explicite

```
# [a; (i:interest_account:>account)];;
- : account list = [<obj>; <obj>]
# [a; (i:>account)];;
- : account list = [<obj>; <obj>]
```

- Possible si `interest_account` est un sous-type de `account` ;
- Dans la liste, l'objet `i` possède toujours la méthode `interest` mais elle ne peut plus être utilisée ;
- Du point de vue de la sûreté d'exécution, c'est incorrect de vouloir l'utiliser et ça n'est donc pas une contrainte !

Sous-typage

Utiliser le sous-typage explicite

```
# [a; (i:interest_account:>account)];;
- : account list = [<obj>; <obj>]
# [a; (i:>account)];;
- : account list = [<obj>; <obj>]
```

- Possible si `interest_account` est un sous-type de `account` ;
- Dans la liste, l'objet `i` possède toujours la méthode `interest` mais elle ne peut plus être utilisée ;
- Du point de vue de la sûreté d'exécution, c'est incorrect de vouloir l'utiliser et ça n'est donc pas une contrainte !

Sous-typage

Utiliser le sous-typage explicite

```
# [a; (i:interest_account:>account)];;
- : account list = [<obj>; <obj>]
# [a; (i:>account)];;
- : account list = [<obj>; <obj>]
```

- Possible si `interest_account` est un sous-type de `account` ;
- Dans la liste, l'objet `i` possède toujours la méthode `interest` mais elle ne peut plus être utilisée ;
- Du point de vue de la sûreté d'exécution, c'est incorrect de vouloir l'utiliser et ça n'est donc pas une contrainte !

Sous-typage

Sous-typage structurel (approximation)

Un type d'objet A est un sous-type d'un type d'objet B :

- Si A et B sont égaux ;
- Ou si chaque méthode de A est une méthode de B avec le même type.

Notez bien

- On ne regarde pas le nom des classes ;
- Conséquence immédiate : sous-type $\not\Rightarrow$ sous-classe.

Dans l'exemple

D'après cette définition :

- `interest_account` est bien un sous-type de `account`.

Sous-typage

Sous-typage en profondeur

Un type d'objet A est un sous-type d'un type d'objet B :

- Si A et B sont égaux ;
- Ou si chaque méthode de A de type τ_A est une méthode de B avec le type τ_B tel que τ_A est un sous-type de τ_B.

Sous-typage entre types fonctionnels (Reynolds, Cardelli)

Le type $D_A \to I_A$ est un sous-type de de $D_B \to I_B$ si :

- I_A est un sous-type de I_B : on peut agrandir l'image (covariance de l'image) ;
- D_B est un sous-type de D_A : on peut rétrécir le domaine (contravariance du domaine).

Sous-typage

Sous-typage en profondeur et fonctionnel

```
# class operations1 =
object
  method op (a : account) = new interest_account a#get
end;;
class operations1 : object
  method op : account → interest_account end
# class operations2 =
object
  method op (a : interest_account) = new account a#get
end;;
class operations2 : object
  method op : interest_account → account end
```

Sous-typage

Sous-typage en profondeur et fonctionnel

```
# class operations1 =
object
  method op (a : account) = new interest_account a#get
end;;
class operations1 : object
  method op : account → interest_account end
# class operations2 =
object
  method op (a : interest_account) = new account a#get
end;;
class operations2 : object
  method op : interest_account → account end
```

Sous-typage

Sous-typage en profondeur et fonctionnel

```
# class operations1 =
object
  method op (a : account) = new interest_account a#get
end;;
class operations1 : object
  method op : account → interest_account end
# class operations2 =
object
  method op (a : interest_account) = new account a#get
end;;
class operations2 : object
  method op : interest_account → account end
```

Sous-typage

Sous-typage en profondeur et fonctionnel

```
# class operations1 =
object
  method op (a : account) = new interest_account a#get
end;;
class operations1 : object
  method op : account → interest_account end
# class operations2 =
object
  method op (a : interest_account) = new account a#get
end;;
class operations2 : object
  method op : interest_account → account end
```

Sous-typage

Sous-typage en profondeur et fonctionnel

```
# let o1 = new operations1;;
val o1 : operations1 = <obj>
# let o2 = new operations2;;
val o2 : operations2 = <obj>
# [(o1:>operations2); o2];;
- : operations2 list = [<obj>; <obj>]
```

Sous-typage

Sous-typage en profondeur et fonctionnel

```
# let o1 = new operations1;;
val o1 : operations1 = <obj>
# let o2 = new operations2;;
val o2 : operations2 = <obj>
# [(o1:>operations2); o2];;
- : operations2 list = [<obj>; <obj>]
```

Sous-typage

Sous-typage en profondeur et fonctionnel

```
# let o1 = new operations1;;
val o1 : operations1 = <obj>
# let o2 = new operations2;;
val o2 : operations2 = <obj>
# [(o1:>operations2); o2];;
- : operations2 list = [<obj>; <obj>]
```

Sous-typage

Sous-typage en profondeur et fonctionnel

```
# let o1 = new operations1;;
val o1 : operations1 = <obj>
# let o2 = new operations2;;
val o2 : operations2 = <obj>
# [(o1:>operations2); o2];;
- : operations2 list = [<obj>; <obj>]
```

Sous-typage

Sous-typage en profondeur et fonctionnel

```
# let o1 = new operations1;;
val o1 : operations1 = <obj>
# let o2 = new operations2;;
val o2 : operations2 = <obj>
# [(o1:>operations2); o2];;
- : operations2 list = [<obj>; <obj>]
```

Sous-typage

Sous-typage en profondeur et fonctionnel

```
# let o1 = new operations1;;
val o1 : operations1 = <obj>
# let o2 = new operations2;;
val o2 : operations2 = <obj>
# [(o1:>operations2); o2];;
- : operations2 list = [<obj>; <obj>]
```

Héritage multiple

Points colorés

```
# class point ((xi, yi) : int * int) =
object
  val x = xi
  val y = yi
  method get_x = x
  method get_y = y
end;;
class point :
  int * int →
  object val x : int val y : int method get_x : int
    method get_y : int end
```

Héritage multiple

Points colorés

```
# class point ((xi, yi) : int * int) =
object
  val x = xi
  val y = yi
  method get_x = x
  method get_y = y
end;;
class point :
  int * int →
  object val x : int val y : int method get_x : int
    method get_y : int end
```

Héritage multiple

Points colorés

```
# class color (c : string) =
object
  val color = c
  method get_color = c
end;;
class color :
  string →
  object val color : string method get_color : string end
```

Héritage multiple

Points colorés

```
# class color (c : string) =
object
  val color = c
  method get_color = c
end;;
class color :
  string →
  object val color : string method get_color : string end
```

Héritage multiple

Points colorés

```
# class colored_point (xi, yi) c =
object (self)
  inherit point (xi, yi)
  inherit color c
  method get = (self#get_x, self#get_y, self#get_color)
end;;
```

Héritage multiple

Points colorés

```
class colored_point :
  int * int →
  string →
  object
    val color : string
    val x : int
    val y : int
    method get : int * int * string
    method get_color : string
    method get_x : int
    method get_y : int
  end
```

Héritage multiple

Points colorés

```
# let cp = new colored_point (1, 2) "blue";;
val cp : colored_point = <obj>
# cp#get;;
- : int * int * string = (1, 2, "blue")
```

Héritage multiple

Points colorés

```
# let cp = new colored_point (1, 2) "blue";;
val cp : colored_point = <obj>
# cp#get;;
- : int * int * string = (1, 2, "blue")
```

Héritage multiple

Points colorés

```
# let cp = new colored_point (1, 2) "blue";;
val cp : colored_point = <obj>
# cp#get;;
- : int * int * string = (1, 2, "blue")
```

Héritage multiple

Points colorés

```
# let cp = new colored_point (1, 2) "blue";;
val cp : colored_point = <obj>
# cp#get;;
- : int * int * string = (1, 2, "blue")
```

Héritage multiple

Avec deux méthodes de même nom dans les super-classes

```
# class point ((xi, yi) : int * int) =
object
  val x = xi
  val y = yi
  method get_x = x
  method get_y = y
  method print =
    print_string "(";
    print_int x;
    print_string ", ";
    print_int y;
    print_endline ")"
end;;
```

Héritage multiple

Avec deux méthodes de même nom dans les super-classes

```
class point :
  int * int →
  object
    val x : int
    val y : int
    method get_x : int
    method get_y : int
    method print : unit
  end
```

Héritage multiple

Avec deux méthodes de même nom dans les super-classes

```
# class color (c : string) =
object
  val color = c
  method get_color = c
  method print = print_endline ("Color: " ^ c)
end;;
class color :
  string →
  object val color : string method get_color : string
    method print : unit end
```

Héritage multiple

Avec deux méthodes de même nom dans les super-classes

```
# class color (c : string) =
object
  val color = c
  method get_color = c
  method print = print_endline ("Color: " ^ c)
end;;
class color :
  string →
  object val color : string method get_color : string
    method print : unit end
```

Héritage multiple

Avec deux méthodes de même nom dans les super-classes

```
# class colored_point (xi, yi) c =
object (self)
  inherit point (xi, yi)
  inherit color c
  method get = (self#get_x, self#get_y, self#get_color)
end;;
```

Héritage multiple

Avec deux méthodes de même nom dans les super-classes

```
class colored_point :
  int * int →
  string →
  object
    val color : string
    val x : int
    val y : int
    method get : int * int * string
    method get_color : string
    method get_x : int
    method get_y : int
    method print : unit
  end
```

Héritage multiple

Avec deux méthodes de même nom dans les super-classes

```
# let cp = new colored_point (1, 2) "blue";;
val cp : colored_point = <obj>
# cp#get;;
- : int * int * string = (1, 2, "blue")
# cp#print;;
Color: blue
- : unit = ()
```

Héritage multiple

Avec deux méthodes de même nom dans les super-classes

```
# let cp = new colored_point (1, 2) "blue";;
val cp : colored_point = <obj>
# cp#get;;
- : int * int * string = (1, 2, "blue")
# cp#print;;
Color: blue
- : unit = ()
```

Héritage multiple

Avec deux méthodes de même nom dans les super-classes

```
# let cp = new colored_point (1, 2) "blue";;
val cp : colored_point = <obj>
# cp#get;;
- : int * int * string = (1, 2, "blue")
# cp#print;;
Color: blue
- : unit = ()
```

Héritage multiple

Avec deux méthodes de même nom dans les super-classes

```
# let cp = new colored_point (1, 2) "blue";;
val cp : colored_point = <obj>
# cp#get;;
- : int * int * string = (1, 2, "blue")
# cp#print;;
Color: blue
- : unit = ()
```

Héritage multiple

Avec deux méthodes de même nom dans les super-classes

```
# let cp = new colored_point (1, 2) "blue";;
val cp : colored_point = <obj>
# cp#get;;
- : int * int * string = (1, 2, "blue")
# cp#print;;
Color: blue
- : unit = ()
```

Héritage multiple

Avec deux méthodes de même nom dans les super-classes

```
# let cp = new colored_point (1, 2) "blue";;
val cp : colored_point = <obj>
# cp#get;;
- : int * int * string = (1, 2, "blue")
# cp#print;;
Color: blue
- : unit = ()
```

Héritage multiple

Avec deux méthodes de même nom dans les super-classes

```
# class colored_point (xi, yi) c =
object (self)
  inherit point (xi, yi) as point_super
  inherit color c as color_super
  method get = (self#get_x, self#get_y, self#get_color)
  method print = point_super#print; color_super#print
end;;
```

Héritage multiple

Avec deux méthodes de même nom dans les super-classes

```
class colored_point :
  int * int →
  string →
  object
    val color : string
    val x : int
    val y : int
    method get : int * int * string
    method get_color : string
    method get_x : int
    method get_y : int
    method print : unit
  end
```

Héritage multiple

Avec deux méthodes de même nom dans les super-classes

```
# let cp = new colored_point (1, 2) "blue";;
val cp : colored_point = <obj>
# cp#print;;
(1, 2)
Color: blue
- : unit = ()
```

Héritage multiple

Avec deux méthodes de même nom dans les super-classes

```
# let cp = new colored_point (1, 2) "blue";;
val cp : colored_point = <obj>
# cp#print;;
(1, 2)
Color: blue
- : unit = ()
```

Héritage multiple

Avec deux méthodes de même nom dans les super-classes

```
# let cp = new colored_point (1, 2) "blue";;
val cp : colored_point = <obj>
# cp#print;;
(1, 2)
Color: blue
- : unit = ()
```

Héritage multiple

Avec deux méthodes de même nom dans les super-classes

```
# let cp = new colored_point (1, 2) "blue";;
val cp : colored_point = <obj>
# cp#print;;
(1, 2)
Color: blue
- : unit = ()
```