Programmation fonctionnelle avec OCaml

6ème et dernière séance, 30 avril 2014

Programmation orientée objet en OCaml

(*Objective* Caml est devenu *OCaml* en 2011)

samuel.hornus@inria.fr

`http://www.loria.fr/~shornus/ocaml/`

Programmation fonctionnelle avec OCaml

6ème et dernière séance, 30 avril 2014

Programmation orientée objet en OCaml

(*Objective* Caml est devenu *OCaml* en 2011)

samuel.hornus@inria.fr

`http://www.loria.fr/~shornus/ocaml/`

Parenthèse

```
# Array.init 10 (fun i -> i mod 3);;
- : int array = [|0; 1; 2; 0; 1; 2; 0; 1; 2; 0|]
```

Créer et utiliser un objet

```
let c = object
  val mutable v = 0
  method valeur = v
  method ajoute d = v <- v + d
  method incremente = v <- v + 1
end
val c : < ajoute : int -> unit; incremente : unit;
          valeur : int > = <obj>
```

Créer et utiliser un objet

```
let c = object
  val mutable v = 0
  method valeur = v
  method ajoute d = v <- v + d
  method incremente = v <- v + 1
end
val c : < ajoute : int -> unit; incremente : unit;
          valeur : int > = <obj>

# c#valeur;;
- : int = 0
# c#incremente; c#valeur;;
- : int = 1
# c#ajoute 11; c#valeur;;
- : int = 12
```

Créer et utiliser un objet

```
let c = object
  val mutable v = 0
  method valeur = v
  method ajoute d = v <- v + d
  method incremente = v <- v + 1
end
val c : < ajoute : int -> unit; incremente : unit;
          valeur : int > = <obj>


# c#valeur;;
- : int = 0
# c#incremente; c#valeur;;
- : int = 1
# c#ajoute 11; c#valeur;;
- : int = 12
```

Encapsulation : Les *variables membre* ne sont accessibles *que* dans la définition de l'objet.

Créer et utiliser un objet

```
let c = object
  val mutable v = 0
  method valeur = v
  method ajoute d = v <- v + d
  method incremente = v <- v + 1
end
val c : < ajoute : int -> unit; incremente : unit;
          valeur : int > = <obj>

# c#valeur;;
- : int = 0
# c#incremente; c#valeur;;
- : int = 1
# c#ajoute 11; c#valeur;;
- : int = 12
```

invocation dynamique
de méthode avec #

Créer et utiliser un objet

```
let c = object
  val mutable v = 0
  method valeur = v
  method ajoute d = v <- v + d
  method incremente = v <- v + 1
end
val c : < ajoute : int -> unit; incremente : unit;
          valeur : int > = <obj>
```

type = <méthodes (avec leur type)>

```
# c#valeur;;
- : int = 0
# c#incremente; c#valeur;;
- : int = 1
# c#ajoute 11; c#valeur;;
- : int = 12
```

Auto référence

```
let c = object (moimeme)
  val mutable v = 0
  method valeur = v
  method ajoute d = v <- v + d
  method incremente = moimeme#ajoute 1
end
val c : < ajoute : int -> unit; incremente : unit;
          valeur : int > = <obj>
```

Auto référence

```
let c = object (toto)
  val mutable v = 0
  method valeur = v
  method ajoute d = v <- v + d
  method incremente = toto#ajoute 1
end
val c : < ajoute : int -> unit; incremente : unit;
          valeur : int > = <obj>
```

Auto référence

```
let c = object (this)
  val mutable v = 0
  method valeur = v
  method ajoute d = v <- v + d
  method incremente = this#ajoute 1
end
val c : < ajoute : int -> unit; incremente : unit;
          valeur : int > = <obj>
```

Auto référence

```
let c = object (self)
  val mutable v = 0
  method valeur = v
  method ajoute d = v <- v + d
  method incremente = self#ajoute 1
end
val c : < ajoute : int -> unit; incremente : unit;
          valeur : int > = <obj>
```

Le nom *self* (ou autre) est lié tardivement, au moment de l'invocation de méthode de l'objet.

Les classes

— Une **classe** fournit un constructeur d'objets :

Utilisation : `let p = new` *`nom_de_classe [paramètre(s)]`*

— Le **type** d'un objet est la liste de ses méthodes (avec leur type). Il est souvent très long !

— Une classe permet l'**abbréviation** du type d'un objet

— Une classe permet d'implémenter l'**héritage**

Les classes

```
class compteur i = object (s)
  val mutable v = i
  method valeur = v
  method ajoute d = v <- v + d
  method incremente = s#ajoute 1
end
class compteur :
  int →
  object
    val mutable v : int
    method ajoute : int → unit
    method incremente : unit
    method valeur : int
  end
```

Les classes

```
class compteur i = object (s)
  val mutable v = i
  method valeur = v
  method ajoute d = v <- v + d
  method incremente = s#ajoute 1
end
# let c = new compteur 3;;
val c : compteur = <obj>
# c#ajoute 10; c#valeur;;
- : int = 13
```

Les classes

```
class compteur i = object (s)
  val mutable v = i
  method valeur = v
  method ajoute d = v <- v + d
  method incremente = s#ajoute 1
end
# let c = new compteur 3;;
val c : compteur = <obj>
# c#ajoute 10; c#valeur;;
- : int = 13
```

Instantiation d'un objet de type (abrégé) `compteur`

Les classes

```
class compteur i = object (s)
  val mutable v = i
  method valeur = v
  method ajoute d = v <- v + d
  method incremente = s#ajoute 1
end
# let c = new compteur 3;;
val c : compteur = <obj>
# c#ajoute 10; c#valeur;;
- : int = 13
```

Instantiation d'un objet de type (abrégé) `compteur`

À RETENIR : une classe n'est pas un type, mais, c'est

- une recette de cuisine pour créer un objet,
- ET une abbréviation du type de l'objet créé

Héritage

```
class compteurPair i = object (self)
  inherit compteur (i / 2) as super
  method decremente = self#ajoute (−1)
  method valeur = 2 * super#valeur
  method valeur = 2 * v
end
```

Le mot-clé `inherit` importe

les • variables membres

et les • méthodes de la classe héritée.

Dans notre exemple :

- la variable `v` est liée correctement (à quoi ?)
- **surcharge** de la méthode `valeur`

Héritage

```
class compteurPair i = object (self)
  inherit compteur (i / 2) as super
  method decremente = self#ajoute (−1)
  method valeur = 2 * super#valeur
  method valeur = 2 * v
end
```

Le mot-clé `inherit` importe

les • variables membres

et les • méthodes de la classe héritée.

Dans notre exemple :

- la variable `v` est liée correctement (à quoi ?)
- **surcharge** de la méthode `valeur`

Héritage

```
class compteurPair i = object (self)
  inherit compteur (i / 2) as super
  method decremente = self#ajoute (−1)
  method valeur = 2 * super#valeur
  method valeur = 2 * v
end
```

Le mot-clé `inherit` importe
les • variables membres
et les • méthodes de la classe héritée.

Dans notre exemple :

- la variable `v` est liée correctement (à quoi ?)
- **surcharge** de la méthode `valeur`

En vrac : héritage multiple

```
class toto a b c = object
  inherit compteur a
  inherit couleur b c
end
```

Si deux classes *parentes* définissent un même nom (variable membre ou méthode), alors

- si même type $\Rightarrow$ le $2^{\text{ème}}$ surcharge le 1^{er}
- sinon $\Rightarrow$ la compilation signale une erreur.

En vrac : initializer

Il est possible de définir une expression qui sera évaluée juste après que l'objet ait été constuit.

```
class toto x = object (me)
  val x = x
  method say_hi = print_string "Hi!"; print_int x
  initializer me#say_hi
end
```

Les *initializers* ne peuvent être surchargés : lors de l'instanciation d'un objet de classe `donald` dérivée de la classe `canard`, l'initializer de `canard` est évaluée d'abord ; l'initializer de `donald` est évaluée ensuite ;

En vrac : méthode privée

Une méthode privée n'est utilisable que par les méthodes (ou l'*initializer*) de la **même** instance.

```
# let po = object (this)
  val mutable x = 0
  method private ajoute d = x <- x + d
  method incr = this#ajoute 1
end;;
val po : < incr : unit > = <obj>
```

En vrac : méthode privée

Une méthode privée n'est utilisable que par les méthodes (ou l'*initializer*) de la **même** instance.

```
# let po = object (this)
  val mutable x = 0
  method private ajoute d = x <- x + d
  method incr = this#ajoute 1
end;;
val po : < incr : unit > = <obj>

# po # incr;;
- : unit = ()
# po # ajoute 3;;
Error: This expression has type < incr : unit >
       It has no method ajoute
```

En vrac : méthode virtuelle

On peut déclarer un méthode comme étant **virtuelle**, pour indiquer que sa définition sera donnée dans les classes dérivées. Alors, on n'indique **que le type** de la méthode :

```
# class virtual abstractCompteur i = object (self)
  val mutable virtual x : int
  method valeur = x
  method incremente = self#ajoute 1
  method virtual ajoute : int -> unit
end;;
class virtual abstractCompteur :
'a ->
object
  val mutable virtual x : int
  method virtual ajoute : int -> unit
  method incremente : unit
  method valeur : int
end
```

En vrac : méthode virtuelle

On peut déclarer un méthode comme étant **virtuelle**, pour indiquer que sa définition sera donnée dans les classes dérivées. Alors, on n'indique **que le type** de la méthode :

```
# class virtual abstractCompteur i = object (self)
  val mutable virtual x : int
  method valeur
  method increme
  method virtual
end;;
class virtual abst
'a ->
object
  val mutable virtual x : int
  method virtual ajoute : int -> unit
  method incremente : unit
  method valeur : int
end
```

Bien-entendu, on ne peut pas instancier un objet d'une classe virtuelle :

```
# let x = new abstractCompteur 4;;
Error: Cannot instantiate the virtual class
abstractCompteur
```

En vrac : interface de classe

On peut en déclarer une interface directement (comme pour les signatures de module) :

```
# class type compteurType = object
  method incremente : unit
  method valeur : int
end;;
class type compteurType =
    object method incremente : unit method valeur : int end
```

En vrac : interface de classe

```
# class compteur i : compteurType = object
    val mutable v = i
    method valeur = v
    method private ajoute d = v <- v + d
    method incremente = v <- v + 1
end;;
class compteur : int -> compteurType
# let x = new compteur 10;;
val x : compteur = <obj>
```

Polymorphisme

Pour se préparer : analogie avec le C++

1. fonction polymorphe sur des objets

```
template< class T >
type maFonction(T & t) { ... }
```

2. classe polymorphe

```
template< typename T >
class maClass { ... };
```

3. methode polymorphe

```
class maClass {
   template< typename T >
   type maFonction(params);
}
```

Fonction polymorphe

```
# let dixfois o = o#valeur * 10;;
val dixfois : < valeur : int; .. > -> int = <fun>
```

- `..` est un paramètre de type (comme `'a`, `'b`) qui signifie « et n'importe quelle(s) autre(s) méthode(s) »

Fonction polymorphe

```
# let dixfois o = o#valeur * 10;;
val dixfois : < valeur : int; .. > -> int = <fun>
```

.. est un paramètre de type (comme 'a, 'b) qui signifie « et n'importe quelle(s) autre(s) méthode(s) »

Si `t = < m1:t1; m2:t2; ···; mn:tn >` est un type d'objet. Alors, un **sous-type** `t'` de `t` est un type d'objet contenant **au moins** $\texttt{m}_1\texttt{:t}_1\texttt{; m}_2\texttt{:t}_2\texttt{; } \cdots \texttt{; m}_n\texttt{:t}_n$

Fonction polymorphe

```
# let dixfois o = o#valeur * 10;;
val dixfois : < valeur : int; .. > -> int = <fun>
```

`..` est un paramètre de type (comme `'a`, `'b`) qui signifie « et n'importe quelle(s) autre(s) méthode(s) »

Si `t = < m1:t1; m2:t2; ···; mn:tn >` est un type d'objet.
Alors, un **sous-type** `t'` de `t` est un type d'objet contenant **au moins** $\texttt{m}_1\texttt{:t}_1\texttt{;}\ \texttt{m}_2\texttt{:t}_2\texttt{;}\ \cdots\texttt{;}\ \texttt{m}_n\texttt{:t}_n$

AINSI : `< valeur : int; .. >` signifie donc
« n'importe-quel sous-type de `< valeur : int >` »

ATTENTION : « sous-type » $\neq$ « classe dérivée »

Coercion

- Soit τ le type de l'objet `p` : τ `= {<···>}`
- Supposons que le type τ est un sous-type de τ'. (τ a les même méthodes que τ', et éventuellement d'autres méthodes.)

Alors on peut *coercer* `p` en un objet de type τ' :

```
# let p' = p :> τ';;
val p' : τ' = <obj>
```

L'objet `p'` est le même que `p`, mais vu comme un objet de type τ'.

C'est tout à fait similaire au *cast* en C++ :

τ'`* p' = dynamic_cast<`τ'`*>(p);`

Classe et méthode polymorphes

Interfaces graphiques et OCaml

Deux librairies facile à obtenir pour créer des interfaces graphiques en OCaml :

- `lablgtk`
- `lablgtk`

Par ailleurs :

- `ocamlsdl` : binding ver libSDL
- `lablgl` : OpenGL et GLUT

Pour le projet « a-maze-ing », utilisez ce que vous voulez pour l'interface graphique (`SDL` ou `lablgtk` ou `lablgtk` ou votre propre code, par exemple sur la base du TP d'aujourd'hui)