Modules

Modules et interfaces

- rôle des interfaces: *masquer*
  - cacher des valeurs: une valeur existe, mais on ne peut en parler (pas de nom)
  - *type abstrait*: déclarer l'existence d'un type en en masquant l'implémentation (seul un nom est fourni)
- utilisation dans le développement logiciel
  - deux modules, une interface

    implémentations différentes d'une même structure de données

    Démo `complexes.ml`
  - deux interfaces, un module (sans type abstrait)

    Démo `gensym-mod.ml`

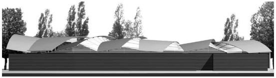
    ↪ aspect "sécurité" – cf. aussi l'encapsulation d'un état (accès contrôlé aux effets de bord)
  - différentes vision d'un même module

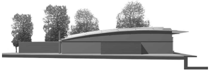
Un module avec un type abstrait

```
module M =
  ( struct
    type t = int ref
    let create() = ref 0
    let add x = incr x
    let get x = if !x>0
       then (decr x; 1)
       else failwith "Empty"
  end
```

```
sig
  type t
  val create :  unit -> t
  val add :  t -> unit
  val get :  t -> int
end )
```

projeter suivant des *vues*





M. Mimram, *Piscine à Viry-Chatillon*

Deux vues sur le module M

```
 module type S1 =     (* propriétaire *)
  sig
    type t
    val create :  unit -> t
    val add :  t -> unit
  end

module type S2 =     (* utilisateur *)
  sig
    type t
    val get :  t -> int

  end


module M1 = (M:S1);;

module M2 = (M:S2);;

let x = M1.create() in M1.add x ; M2.get x ;;

  This expression has type M1.t but is here used with type M2.t
```

Modifier une signature à la volée

- la construction `with`

```
module M1 = (M:S1 with type t = M.t) ;;
module M2 = (M:S2 with type t = M.t) ;;
let x = M1.create() in M1.add x ; M2.get x ;;
  - :  int = 1
```

- d'ailleurs

```
module type Order =
   sig
     type t
     val order :  t->t->bool
   end
```

```
module IntOrder :  Order =
   struct
     let minus x y = x-y
     let order x y =
         (minus y x) >= 0
     type t = int
   end
```

  - `IntOrder.order 3 4` $\rightsquigarrow$ erreur `IntOrder` est inutilisable
  - `module IntOrder2 :  (Order with type t = int) ...`

Cohérence entre les types

- ça non plus, ça ne marche pas:

```
module M1 = (M:S1) ;;
module M2 = (M:S2 with type t = M1.t) ;;
  Type declarations do not match:
    type t = M.t is not included in type t = M1.t
```

- on construit des types dont les *noms* diffèrent, bien qu'ils soient les mêmes *“en dessous”*
- par défaut, ces types sont incompatibles, sauf si on explicite le partage (`with`)
  - le problème surgit lors de la phase de typage
  - on a affaire à un **choix de conception** du langage