

3 – modules et foncteurs

lorsque les programmes deviennent gros il faut

- découper en unités (**modularité**)
- occulter la représentation de certaines données (**encapsulation**)
- éviter au mieux la duplication de code

en OCaml : fonctionnalités apportées par les **modules**

chaque fichier est un module

si arith.ml contient

```
let pi = 3.141592
let round x = floor (x +. 0.5)
```

alors on le compile avec

```
$ ocamlc -c arith.ml
```

utilisation dans un autre module main.ml :

```
let x = float_of_string (read_line ())
let () = print_float (Arith.round (x /. Arith.pi))
```

```
$ ocamlc -c main.ml
```

```
$ ocamlc arith.cmx main.cmx -o main
```

on peut restreindre les valeurs exportées avec une **interface**

dans un fichier `arith.mli`

```
val round: float -> float
```

```
$ ocamlOPT -c arith.mli
```

```
$ ocamlOPT -c arith.ml
```

```
$ ocamlOPT -c main.ml
```

```
File "main.ml", line 2, characters 33-41:
```

```
Unbound value Arith.pi
```

une interface peut restreindre la visibilité de la **définition** d'un type

dans set.ml

```
type t = int list
let empty = []
let add x l = x :: l
let mem = List.mem
```

mais dans set.mli

```
type t
val empty : t
val add : int -> t -> t
val mem : int -> t -> bool
```

le type t est un **type abstrait**

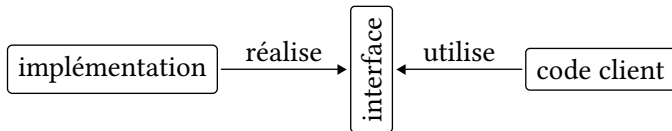
la bibliothèque d'OCaml fournit un module `Hashtbl`

- sans révéler la définition du type `Hashtbl.t`
- sans exporter les fonctions auxiliaires

la documentation nous dit qu'il s'agit de tables de hachages, on peut même lire le code par curiosité, mais le compilateur ne nous laisse pas accéder à ce qui est caché

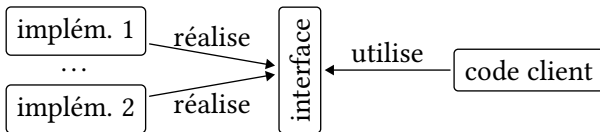
c'est un bon principe que de **cacher** tout ce que l'on peut cacher

le langage OCaml assure une vraie **barrière d'abstraction**, qu'il n'est pas possible de contourner



c'est un bon principe que de **cacher** tout ce que l'on peut cacher

le langage OCaml assure une vraie **barrière d'abstraction**, qu'il n'est pas possible de contourner



modules non restreints aux fichiers

```
module M = struct
  let c = 100
  let f x = c * x
end
```

```
module A = struct
  let a = 2
  module B = struct
    let b = 3
    let f x = a * b * x
  end
  let f x = B.f (x + 1)
end
```

de même pour les signatures

```
module type S = sig
  val f: int -> int
end
```

contrainte

```
module M: S = struct
  let a = 2
  let f x = a * x
end
```

M.a

Unbound value M.a

on peut rendre visible tous les éléments d'un module avec `open`

```
open Printf
...
... printf "%d" ...
...
... printf "%s" ...
...
```

nous évite d'écrire systématiquement `Printf.printf`

- modularité par découpage du code en unités appelées **modules**
- encapsulation de types et de valeurs, **types abstraits**
- organisation de l'**espace de nommage**

foncteur = **module paramétré** par un ou plusieurs autres modules

exemple : table de hachage **générique**

il faut paramétrer par rapport aux fonctions de hachage et d'égalité

```
module HashTable(K: KEY) = struct ... end
```

avec

```
module type KEY = sig
  type key
  val hash: key -> int
  val eq: key -> key -> bool
end
```

```
module HashTable(K: KEY) = struct
  type 'a t = (K.key * 'a) list array
  let create n = Array.make n []
  let add t k v =
    ... K.hash k ...
  let find t k =
    ... if K.eq k' k then ...
end
```

```
module HashTable(K: KEY): sig
  type 'a t
  val create: int -> 'a t
  val add: 'a t -> K.key -> 'a -> unit
  val find: 'a t -> K.key -> 'a
end
```

```
module Int = struct
  type key = int
  let hash x = x
  let eq x y = x=y
end
```

```
module Hint = HashTable(Int)
```

```
let table = Hint.create 16
let () = Hint.add table 1729 "Ramanujan-Hardy"
...
```

(ici table a le type string Hint.t)

écrire un foncteur pour calculer modulo m

```
module Modular(M: sig val m: int end): sig
  type t
  val of_int: int -> t
  val add: t -> t -> t
  val sub: t -> t -> t
  val mul: t -> t -> t
  val div: t -> t -> t
  (** divise x par y, en supposant y premier avec m *)
  val to_int: t -> int
end
```