

# Chapitre 8

## Objets

### Sommaire

---

<b>8.1 Simulation à l'aide de modules ou enregistrements</b>	<b>67</b>
<b>8.2 Objets</b>	<b>68</b>
<b>8.3 Classes</b>	<b>69</b>
8.3.1 Définition	69
8.3.2 Héritage	69
8.3.3 Variables de classe	70
8.3.4 Classes et méthodes virtuelles	71
<b>8.4 Typage des objets</b>	<b>71</b>
8.4.1 Les types de classes	71
8.4.2 Classes paramétrées	72
<b>8.5 Comparaison avec les objets de Java</b>	<b>74</b>
<b>8.6 Conception de gros projets avec OCaml</b>	<b>75</b>
<b>8.7 Types sommes, modules ou objets ?</b>	<b>75</b>
8.7.1 Comparaison	78
8.7.2 Organisations mixtes	79
<b>8.8 Design patterns</b>	<b>80</b>
<b>8.9 Exemple d'utilisation : Lablgtk2</b>	<b>80</b>
8.9.1 Hello world	80
8.9.2 Les modules	80
8.9.3 Glade	83

---

**Avertissement** Ce chapitre est en version préliminaire. Merci de me signaler tout problème.

**Avertissement 2** Les objets ne sont pas au programme de l'examen.

Pour cette partie, je suppose que le lecteur a connaissance d'un langage objet. En particulier, j'essaierai de comparer les objets d'*OCaml* à ceux du langage *Java*.

### 8.1 Simulation à l'aide de modules ou enregistrements

```
type tcompteur = {  
  raz : unit -> unit;  
  get : unit -> int;
```

```

    inc : unit -> unit;
  }

  let creecompteur () =
    let c = ref 0 in
    { raz = (fun () -> c := 0);
      get = (fun () -> !c);
      inc = (fun () -> c := !c+1)};;

  let c = creecompteur ();;

  c.get ();;

```

On peut faire la même chose avec des modules.

Évidemment, un objet n'est pas juste un enregistrement ou un module. Il nous manque en particulier les notions d'héritage et de liaison tardive.

## 8.2 Objets

```

# let o = object
  val plop = 1 (* variable d'instance non mutable *)
  val mutable plip = 2 (* variable d'instance mutable *)
  val f = fun x -> x+1 (* variable d'instance fonctionnelle *)
  method setplip n = plip <- n (* méthode ;
                               mise à jour d'un champ mutable *)
  method getplop = plop (* méthode sans paramètre *)
  method getplip = plip
  method private toto x y = x (* méthode invisible de l'extérieur *)
  initializer print_endline "Bonjour" (* corps du constructeur *)
end;;
Bonjour
val o : < getplip : int; getplop : int; setplip : int -> unit > = <obj>

```

Remarquez le type. Les variables d'instances comme les méthodes privées ne sont pas accessibles de l'extérieur (privées).

### Appels de méthodes

Appel de méthode : `monobjet#getplip`

En programmation orientée objets, les appels (ou invocations) de méthodes sont vus comme des envois de messages. Un envoi de message déclenche un traitement. En partant de ce point de vue, on peut autoriser les méthodes sans paramètre.

```

# let o = object
  val plop = print_string "ciao\n"; 4
  method toto = print_string "hello\n" ; 5
end;;
ciao
val o : < toto : int > = <obj>
# o#toto;;
hello
- : int = 5
# o#toto;;
hello
- : int = 5

```

## self

Il est possible de nommer l'objet pour lui permettre de s'envoyer un message à lui-même en utilisant la syntaxe suivante.

```
let o = object (moi)
  ...
end
```

C'est l'équivalent du mot-clef *this* en *Java* (ou *self* dans certains langages).

## 8.3 Classes

### 8.3.1 Définition

```
class maclasse i = object  (* Une classe est vue comme une fonction
                           qui crée une instance (constructeur) *)
                           (* Noter le paramètre i du constructeur *)
  val mutable plip = i
  method toto = 3
  initializer
    print_string "Bonjour"  (* corps du constructeur *)
end;;
```

Création d'un objet : `new maclasse 4`

Différences entre une classe et une fonction qui crée un objet :

- la classe définit simultanément une abréviation de type ;
- la classe permet l'héritage.

### 8.3.2 Héritage

Le gros avantage de la programmation objet est 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.

Contrairement à *Java*, *OCaml* autorise l'héritage multiple. Syntaxe :

```
class classefille v1 v2 = object
  inherit classe1
  inherit classe2 as pere
  inherit classe3 v1
  ...
end
```

Le mot-clé `as` permet d'appeler les méthode des parents (équivalent du mot-clé `super` en *Java*).

Les valeurs et méthodes privées sont héritées (donc accessibles à partir de la classe fille).

L'héritage multiple peut entraîner des problèmes de conflits de noms de variables si plusieurs champs ont le même nom.

**Redéfinition** une classe fille peut *redéfinir* une méthode du père (*overriding*).

**Rappel sur la liaison tardive** Lorsque l'on envoie un message à un objet le code réellement exécuté en réponse ne peut être décidé qu'au moment de l'exécution (tout simplement parce que cet objet peut être une instance d'une classe héritière et on ne sait pas laquelle à l'avance). C'est en fait principalement ce comportement qui fait l'intérêt des objets, et c'est là que l'on voit clairement la différence entre une méthode (qui répond à un message) et une fonction (qui est appelée à une adresse fixe déterminée à la compilation ou l'édition de liens).

Regardez par exemple le comportement du petit programme suivant :

```
class a = object (moi)
  method f = print_string "f de a\n"
  method g = print_string "g de a\n"; moi#f
end

class b = object
  inherit a
  method f = print_string "f de b\n"
end

# let ob = new b;;
val ob : b = <obj>
# ob#g;;
g de a
f de b
- : unit = ()
```

**Masquage ou redéfinition** Les champs peuvent aussi être redéfinis. (En fait depuis la version 3.10 de OCaml seulement. Avant ils étaient *masqués*). Regardez l'exemple suivant :

```
# class c = object val a = 1 method g = a end;;
class c : object val a : int method g : int end
# class d = object inherit c val a = 2 end;;
Characters 30-36:
  class d = object inherit c val a = 2 end;;
                ~~~~~
Warning V: the instance variable a is overridden.
The behaviour changed in ocaml 3.10 (previous behaviour was hiding.)
class d : object val a : int method g : int end
# let b = new d;;
val b : d = <obj>
# b#g;;
- : int = 2
```

### 8.3.3 Variables de classe

Les variables de classes (*static* en *Java*) sont des variables communes à toutes les instances d'une même classe. En *OCaml*, il n'y a pas besoin de syntaxe particulière pour définir des variables de classe. En effet, la syntaxe :

```
class c p1 ... pn = object ... end
```

est en fait l'abréviation de

```
class c = fun p1 ... pn -> object ... end
```

(Rappelons qu'une classe est vue comme la fonction de construction d'une instance.) Ceci permet par exemple de faire

```
class c =
  let v = ref 0 in
  fun p1 ... pn -> object ... end
```

Ici  $v$  est une variable de classe, commune à toutes les instances (même principe que dans l'exemple du compteur page 67).

Comme en *Java*, une variable (fonction) de classe ne peut pas être redéfinie (d'ailleurs la liaison est statique).

**Méthodes de classe** (ou méthodes statiques) La notion de méthode de classe permet de retrouver le comportement des fonctions (liaison statique) dans des langages tout objet comme *Java*. En *OCaml*, cette notion n'est pas utile.

### 8.3.4 Classes et méthodes virtuelles

On peut définir des classes virtuelles (non instanciables, *abstract* en *Java*), qui peuvent contenir des méthodes virtuelles (définies juste par leur type). Exemple :

```
class virtual c p1 ... pn = object
  ...
  val virtual v : int
  ...
  method virtual plop : int -> int
end
```

## 8.4 Typage des objets

*OCaml* est le seul langage avec une extension objets (et polymorphisme paramétrique et d'inclusion) qui conserve un typage complètement statique et l'inférence des types. Voyons comment sont typés les objets.

### 8.4.1 Les types de classes

Un *type de classe* contient l'interface d'une classe, c'est-à-dire le nom et le type de ses méthodes publiques. **Il ne faut pas confondre type d'un objet et classe.** Une classe n'est pas un type. On peut la voir comme une fonction qui crée une instance, donc plutôt une valeur, même si ce n'est pas un objet de première classe (la seule façon de l'utiliser est après un `new`). Il faut plutôt rapprocher les classes des modules ou foncteurs, qui eux non-plus ne sont pas des objets de première classe. On dit que *OCaml* fait du *typage structurel* (par opposition au *typage nominal*).

Pour pouvoir utiliser une contrainte de type, on peut définir l'interface d'une classe de la manière suivante :

```
class type interf_a = object
  method m : int
end ;;

let f (o:interf_a) = 44;;
```

Par défaut, le nom de la classe est utilisé comme abréviation de son type :

```
# let ob = new a;;
val ob : a = <obj>
# ob;;
- : a = <obj>
```

Notons bien que deux classes peuvent avoir les mêmes noms de méthodes (alors que c'est faux avec les enregistrements). Conséquence : **deux objets de classes différentes peuvent avoir le même type!**

```

class a = object
  method m = 3
end;;

class b = object
  method m = 888
end;;

let oa = new a;;
let ob = new b;;

# [oa; ob];;
- : a list = [<obj>; <obj>]

```

Quand l'inférence de type ne sait pas à quelle classe appartient un objet, elle utilise des *types ouverts* :

```

# let f o = o#m;;
val f : < m : 'a; .. > -> 'a = <fun>

```

Ce qui se lit « un objet contenant *au moins* une méthode appelée *m* de type *'a* ».

Syntaxe : le type ouvert correspondant à une classe *c* se note *#c*, ce que l'on peut lire « objet qui a au moins les méthodes de la classe *c* ».

**Coercion** Les types ouverts induisent une notion de sous-typage. Attention à ne pas confondre les notions de sous-typage et de sous-classe (héritage) ! On peut coercer un type de classe vers un sur-type (syntaxe (*v* : *t1* :> *t2*)). Exemple :

```

class a = object
  method m = 3
end;;

class b = object
  method m = 888
  method mm = 111
end;;

let oa = new a;;
let ob = new b;;

# [oa; ob];;
Characters 5-7:
[oa; ob];;
^^

This expression has type b but is here used with type a
Only the first object type has a method mm

# [oa;(ob :> a)];;
- : a list = [<obj>; <obj>]

```

Attention : la coercion dans l'autre sens est bien évidemment interdite en *OCaml*, car dangereuse (alors que certains langages objets comme *Java* l'autorisent !).

## 8.4.2 Classes paramétrées

En *OCaml*, les types ne peuvent pas contenir de variables de types libres.

```

# type t = 'a list;;
Characters 9-11:
type t = 'a list;;
^^

```

```
Unbound type parameter 'a
# type 'a t = 'a list;;
type 'a t = 'a list
```

Comme la création d'une classe entraîne la définition d'un type de classe, il ne faut pas que ce type contienne des variables de type libres (ni de type ouvert).

```
# class a = object
  method id x = x
end;;
Characters 5-38:
..... a = object
  method id x = x
end..
Some type variables are unbound in this type:
class a : object method id : 'a -> 'a end
The method id has type 'a -> 'a where 'a is unbound
```

Solutions à retenir :

1. Ajouter des contraintes de type (par exemple `method id (x : int) = x`),
2. Mettre un quantificateur de type, pour obtenir une *méthode polymorphe* (c'est autorisé aussi pour les champs des records) par exemple `method id : 'a. 'a -> 'a = fun x -> x`, (« 'a. » se lit « pour tout 'a »),
3. ou bien, lorsque l'on veut que le type soit fixé pour chaque instance, utiliser des classes paramétrées :

```
class ['a] maclasse = object
  method id x:'a = x
end;;
```

Lorsque l'on veut plusieurs paramètres de type, on écrit : `class ['a,'b] maclasse = ...`

On devra parfois donner des contraintes sur les paramètres de type :

```
class ['a] maclasse = object
  constraint 'a = #printable (* 'a doit avoir au moins les méthodes
                             du type de classe printable *)
  method id x:'a = x
end;;
```

Autre exemple un peu plus compliqué :

```
# class d = object
  method mm = 3
end;;
class d : object method mm : int end
# class c = object
  method m a = a#mm
end;;
Some type variables are unbound in this type:
class c : object method m : < mm : 'a; .. > -> 'a end
The method m has type (< mm : 'b; .. > as 'a) -> 'b where 'a is unbound
```

C'est le même problème qu'avec les variants polymorphes : les types ouverts contiennent une *variable de rangée*.

Solutions :

```
# class c = object
  method m (a : d) = a#mm
end ;;
class c : object method m : d -> int end
```

```
# class c = object
  method m : 'a. (< mm : int; .. > as 'a) -> int = fun a -> a#mm
  end ;;
class c : object method m : < mm : int; .. > -> int end

# class ['a] c = object
  method m : (< mm : int; .. > as 'a) -> int = fun a -> a#mm
  end ;;
class ['a] c :
  object constraint 'a = < mm : int; .. > method m : 'a -> int end
```

## 8.5 Comparaison avec les objets de *Java*

Principales différences entre les objets de *Java* et de *OCaml* :

Java	Ocaml
Tout objet	extension objet
surcharge des méthodes	pas de surcharge
types déclarés	types inférés
héritage simple	héritage multiple
pas de classes paramétrées	classes paramétrées
Sous-typage = héritage	Sous-typage $\neq$ héritage

...

Exemple illustrant la différence entre les deux notions de sous-typage :

```
class Carre { void nom() { System.out.println("carre"); } }
class Cercle { void nom() { System.out.println("cercle"); } }
class Obj1 {
  static void f (Cercle c) { c.nom(); }
  public static void main(String[] a) {
    Cercle ce = new Cercle();
    f(ce);
    Carre ca = new Carre();
    f(ca);
  }
}
```

```
$ javac Obj1.java
```

```
Obj1.java:21: f(Cercle) in Obj1 cannot be applied to (Carre)
    f(ca);
    ^
```

```
1 error
```

```
# class carre = object method nom = print_endline "carre" end;;
class carre : object method nom : unit end
# class cercle = object method nom = print_endline "cercle" end;;
class cercle : object method nom : unit end
# let f (c : cercle) = c#nom;;
val f : cercle -> unit = <fun>
# let o = new carre;;
val o : carre = <obj>
# f o;;
carre
```

On pourra tomber sur quelques points qui posent problème en *OCaml* mais pas en *Java*, notamment lors de l'implémentation de design patterns particuliers. Par exemple il est difficile de créer des objets mutuellement récursifs (pas de types de classes récursifs)

*Java* ne possède pas de classe paramétrée. Il a deux types de polymorphisme : *ad hoc* pour la surcharge et *d'inclusion* pour la redéfinition (liaison tardive). *OCaml* a aussi le polymorphisme d'inclusion, mais pas de surcharge. Les deux disposent du polymorphisme dit *paramétrique* (celui habituel de *Caml* avec les types 'a, appelé *generics* en *Java*), que l'on peut utiliser conjointement au polymorphisme d'inclusion grâce aux classes paramétrées.

*Java* permet par ailleurs les conversions de type d'une sur-classe vers une sous-classe ! Ce qui peut bien entendu entraîner des erreurs d'exécution (levée d'une exception) si l'on appelle une méthode inexistante. Cela utilise du typage dynamique. *OCaml* interdit bien-sûr de telles conversions de types, et même si ce trait peut-être utile (notamment lors de la programmation d'interfaces graphiques), on peut trouver des équivalents plus propres en *OCaml*, notamment avec les classes paramétrées.

Pas de méthodes de classes en *OCaml*, mais cela n'a pas vraiment de sens, puisqu'il est beaucoup plus naturel d'écrire une fonction.

## 8.6 Conception de gros projets avec *OCaml*

### « *Programming in the large* »

*OCaml* est un langage dit *multi-paradigme*, dans le sens où il permet la programmation fonctionnelle, la programmation impérative et la programmation orientée objets, mais n'impose aucune des trois.

Lors de la conception d'un gros projet, les équipes de développement doivent choisir très tôt l'organisation « à grande échelle » du programme. Alors que *Java* impose le tout objet et donne des patrons de conception (*design patterns*) tout prêts bien connus, la réflexion doit être un peu différente lors de la programmation en *OCaml*. En particulier plaquer certains patrons de conception de *Java* peut être une erreur s'il existe en *OCaml* une façon de faire plus adaptée ou plus simple sans objet. De nombreux design patterns expliquent seulement comment traduire dans un langage tout objet ce que l'on sait très bien faire dans d'autres langages sans avoir recours aux objets (singleton, visiteur... même si on peut être amené à les utiliser dans certains cas).

En particulier, ce que l'on programmerait sans réfléchir avec des objets en *Java* peut être implémenté à l'aide d'objets en *OCaml*, mais on préférera parfois utiliser des modules et foncteurs (lorsque les objets sont utilisés comme unité de programmation) ou des types sommes (lorsque les objets sont utilisés pour remplacer une structure de données).

Nous allons voir dans cette section comment choisir entre objets, modules et types sommes, et nous essaierons de montrer que l'on peut tirer partie de cette liberté supplémentaire pour améliorer encore l'organisation du programme en mêlant ces concepts. On donnera notamment quelques patrons de conception adaptés à *OCaml*.

## 8.7 Types sommes, modules ou objets ?

Les modules et les classes sont deux façons très différentes d'organiser un programme. Ils permettent de répondre à des problèmes différents mais il y a aussi de la redondance, comme le montre le résumé suivant (tiré du livre de Chailloux Manoury Pagano chez O'Reilly) :

- organisation logique d'un programme : module ou classe
- compilation séparée : module simple
- types abstraits de données : module (type abstrait) ou objet
- réutilisation des composants : foncteurs/partage de types avec polymorphisme paramétrique ou héritage/sous-typage avec des classes paramétrées
- modifiabilité des composants : liaison tardive (objet)

Des travaux de recherches portent actuellement sur un nouveau paradigme, appelé *mixin*, qui combinerait les fonctionnalités des deux approches. Mais à l'heure actuelle aucun langage ne l'implémente et il est peu probable qu'il soit inclus un jour dans *OCaml*. En attendant, pour bénéficier des deux simultanément, la solution consiste à combiner les modules avec les objets.

Les objets mêlent les données avec leurs traitements. En programmation fonctionnelle, on sépare les données (représentées par exemple dans un type somme) de leur traitement, et très souvent en *OCaml*, on regroupera le tout dans un module en abstrayant le type (encapsulation).

**Exemple :** Voici un petit type de données arborescent (ici un simple arbre binaire) représenté tour à tour de manière fonctionnelle et objet :

```

type tarbre = Feuille of string | Noeud of tarbre * tarbre ;;

let rec affiche = function
  | Feuille a -> print_string a
  | Noeud (g,d) ->
    print_string "<g>" ;
    affiche g ;
    print_string "</g>\n" ;
    print_string "<d>" ;
    affiche d ;
    print_string "</d>\n"
;;

# affiche
  (Noeud
    (Noeud
      (Feuille "coucou",
        Feuille "ciao")),
      Feuille "hello"));;
<g><g>coucou</g>
<d>ciao</d>
</g>
<d>hello</d>
- : unit = ()

```

Maintenant avec des objets. Pour traduire des types sommes en objets, on utilise généralement le design pattern *composite*. Ici une version très simple.

```

class feuille v = object
  method affiche = print_string v
end;;

class noeud g d = object
  method affiche =
    print_string "<g>" ;
    g#affiche ;
    print_string "</g>\n" ;
    print_string "<d>" ;
    d#affiche ;
    print_string "</d>\n"
end;;

# (new noeud
  (new noeud
    (new feuille "coucou")
    (new feuille "ciao"))
  (new feuille "hello"))#affiche;;
<g><g>coucou</g>
<d>ciao</d>
</g>

```

```
<d>hello</d>
- : unit = ()
```

Dans cette exemple j'ai utilisé une vision assez « fonctionnelle » des objets, sans champ mutable. En fait on a plutôt l'habitude de voir un objet comme une structure contenant des champs mutables, comme dans la version suivante. Au passage je rajoute une méthode qui renverse l'arbre.

```
class feuille v = object
  val mutable valeur = v
  method affiche = print_string valeur
  method retourne = ()
end;;
```

```
class noeud g d = object
  val mutable gauche = g
  val mutable droite = d
  method affiche =
    print_string "<g>" ;
    gauche#affiche ;
    print_string "</g>\n" ;
    print_string "<d>" ;
    droite#affiche ;
    print_string "</d>\n"
  method retourne =
    gauche#retourne ;
    droite#retourne ;
    let aux = gauche in
    gauche <- droite ;
    droite <- aux
end;;
```

```
# let a =
  (new noeud
    (new noeud
      (new feuille "coucou")
      (new feuille "ciao")))
  (new feuille "hello"));
val a : noeud = <obj>
```

```
# a#retourne;;
- : unit = ()
```

```
# a#affiche;;
<g>hello</g>
<d><g>ciao</g>
<d>coucou</d>
</d>
- : unit = ()
```

En programmation fonctionnelle, la fonction retourne s'écrit :

```
let rec retourne = function
| Feuille a -> Feuille a
| Noeud (g, d) -> Noeud(retourne d, retourne g);;

# retourne
  (Noeud
    (Noeud
      (Feuille "coucou",
        Feuille "ciao"),
      Feuille "hello"));
- : tarbre =
Noeud (Feuille "hello", Noeud (Feuille "coucou", Feuille "ciao"))
```

Je définis ci-dessous une fonction qui transforme un arbre en peigne :

```
let rec peigne = function
| Feuille a -> Feuille a
| Noeud (g, Feuille a) -> Noeud((peigne g),Feuille a)
| Noeud (g, Noeud(gg, dd)) -> peigne (Noeud ((Noeud(g, gg)), dd));;
```

Je vous laisse la définir dans le cadre objet. (Bon courage).

### 8.7.1 Comparaison

Voici quelques points de comparaison entre approche objet et approche fonctionnelle (sommes + modules).

**Facilité d'utilisation** L'exemple de la section précédente montre que l'approche fonctionnelle est souvent plus concise et le filtrage (pattern matching) est un outil très confortable. Voir par exemple la fonction `peigne` ci-dessus.

Par ailleurs les objets obligeront à des solutions tordues quand on veut définir une opération binaire sur un type de données... Une méthode agit sur l'objet auquel elle appartient. Mais que faire lorsque l'on veut agir sur deux objets en même temps ? On obtient des solutions dissymétriques. Exemple : une classe pour définir des ensembles de valeurs. Quel va être le type de la méthode implémentant l'union d'ensembles ? On appelle ces méthodes des méthodes binaires.

L'utilisation d'objets se prête donc très bien à certains problèmes mais pas à tous.

**Efficacité** Une invocation de méthode est moins efficace qu'un appel de fonction, puisqu'il faut résoudre la liaison tardive au moment de l'exécution.

**Types abstraits** Seuls les modules permettent d'abstraire un type, ce qui permet de masquer l'implémentation et de préserver ainsi l'intégrité des données. Cela dit, l'encapsulation (champs privés) peut parfois jouer un rôle similaire.

**Ajout de traitements** Ajouter une fonction de traitement des données en fonctionnel est peu coûteux. Il suffit d'écrire la fonction (cf `retourne` ci-dessus).

En objets, on peut ajouter un traitement en écrivant une nouvelle méthode dans chaque classe, ce qui implique la modification d'une grande partie du code. Si l'on ne veut pas (ou si l'on ne peut pas) modifier les classes existantes, il faut en créer des nouvelles, qui héritent des précédentes. Mais c'est très lourd et cela oblige à modifier le code à tous les endroits où l'on crée une de ces classes pour créer l'une des nouvelles... Certains design patterns (comme le visiteur) sont conçus pour réduire cet inconvénient.

Voir l'exemple (déjà traité) de l'évaluation d'expressions arithmétiques et de la distributivité.

**Extension du type de données** Avec les objets, étendre une structure de données est souvent assez peu coûteux (par exemple ajouter un nouveau type de noeuds ternaires dans nos arbres). Il suffit de créer une nouvelle classe.

Avec les types sommes, étendre un type de données (pour rajouter un constructeur) oblige à revoir toutes les fonctions qui utilisent ce type.

Exemple : interfaces graphiques

**Limitations des objets d'OCaml** Le typage d'OCaml et notamment l'inférence de types impose certaines contraintes au programmeur de classes qui peuvent rendre l'utilisation d'objets plus lourde en OCaml, et faire préférer parfois l'approche fonctionnelle. C'est la cause des principaux inconvénients de l'extension objets d'OCaml, résumés ici :

- pas de méthode contenant un paramètre de type libre, ce qui introduit une rigidité (comblée par les classes paramétrées au prix d'une certaine lourdeur);

- pas de surcharge (comme dans le reste de *Caml*). Cela est dû au choix du typage complètement statique de *Caml*. On trouve en général facilement un moyen de s'en passer (modification de la hiérarchie des classes, utilisation de paramètres optionnels,...);
  - difficulté d'échappement du type de la classe dans une de ses méthodes;
  - absence de contrainte de type d'ancêtre vers descendant;
  - pas de hiérarchie de classes dans la bibliothèque standard
- Plus de détails dans le Chailloux-Manoury-Pagano.

**En résumé** on utilisera les objets lorsque l'on veut pouvoir étendre facilement nos structures alors que les traitements à effectuer sont assez figés. Exemple typique : pour une interface graphique, on veut pouvoir créer de nouveaux composants en assemblant entre eux des composants existants. Au contraire si la structure de donnée est figée mais que l'on veut pouvoir étendre les traitements à volonté, on utilisera une approche fonctionnelle (par exemple implémentation d'une structure de données usuelle comme les listes).

Lorsque l'on veut étendre à la fois les données et les traitements, une approche mixte objets + fonctions est utilisable (voir un exemple dans le livre de Chailloux-Manoury-Pagano).

On pourra retenir également que :

- l'approche objet est la seule à permettre la liaison tardive
- pour des types de données abstraits, en particulier avec opérations binaires, on préférera les modules

### 8.7.2 Organisations mixtes

**Organisation générale** Une organisation typique consistera à utiliser les modules et foncteurs pour l'organisation de haut niveau du programme (compilation séparée, structuration du code, abstraction des types, encapsulation), et les classes pourront être des composants des modules (extensibilité, liaison tardive).

**Classes amies** Lorsque l'on rend une méthode publique dans une classe, elle est visible du monde entier. On peut avoir envie de n'autoriser que les autres classes du modules à accéder à certaines données. Typiquement, une méthode binaire (méthode implémentant une opération binaire, donc prenant un paramètre un objet de la même classe) aura souvent besoin de regarder à l'intérieur de l'objet qui lui est passé en paramètre.

On peut utiliser les modules pour rassembler des « classes amies », ce qui permettra de donner plus de possibilités aux classes définies dans le même module qu'au monde extérieur.

**Classes vues comme des « pré-modules »** Une autre organisation possible consiste à créer une classe virtuelle paramétrée et d'en donner dans un module une implémentation pour un type donné, abstrait. Par exemple on peut définir une classe virtuelle `group` pour définir la notion mathématique de groupe, paramétrée par le type des éléments du groupe. Ensuite, on peut définir un groupe concret (par exemple le groupe  $Z/2Z$ ) en créant un module de signature :

```
module type GROUP = sig
  type t
  val meth : t group
end
```

On peut même créer un foncteur qui va construire le module  $Z/pZ$  en fonction de  $p$ .

**Classes et foncteurs** Je cite la Bible :

« On propose maintenant de mixer modules paramétrés et liaison retardée pour profiter de la puissance de ces deux traits. L'application du foncteur produira de nouveaux modules contenant des classes utilisant le type et les fonctions du module paramètre. Si, de surcroît, la signature obtenue est compatible avec la signature du module paramètre, il est alors possible de ré-appliquer le module paramétré sur le module résultat, permettant ainsi de construire automatiquement de nouvelles classes. »

À creuser... (voir l'exemple du serveur dans le livre).

## 8.8 Design patterns

\*\*\* à suivre \*\*\*

## 8.9 Exemple d'utilisation : Lablgtk2

Cette section est tirée en partie du README de Lablgtk2. Les exemples sont tirés du tutoriel Lablgtk2 <http://plus.kaist.ac.kr/~shoh/ocaml/lablgtk2/lablgtk2-tutorial/>

### 8.9.1 Hello world

```
# #use "topfind";; (* J'utilise findlib pour simplifier *)
# #require "lablgtk2";; (* #require est une directive créée par findlib *)
# #load "gtkInit.cmo";;
# let main () =
  let window = GWindow.window () in
  window#show ();
  GMain.Main.main ();
;;
val main : unit -> unit = <fun>
# main ();;
```

La fonction `GMain.Main.main` lance la boucle principale de Gtk dans un thread séparé.

Le script `lablgtk2` lance la toplevel OCaml avec la bibliothèque `lablgtk2` déjà chargés. Cela évite les premières lignes ci-dessus.

### 8.9.2 Les modules

Les modules de la bibliothèque LablGTK2 :

<code>Gdk</code>	low-level interface to the General Drawing Kit
<code>Gtk*</code>	low-level interface to the GIMP Tool Kit
<code>GtkThread</code>	main loop for threaded version
<code>G[A-Z]*</code>	object-oriented interface to GTK
<code>GdkObj</code>	object-oriented interface to GDK

Les modules `Gtk*` sont composés d'un sous-module pour chaque classe `Gtk2`.

Le modules `G[A-Z]*` contiennent les widgets :

GPango	Pango font handling
GDraw	Gdk pixmaps, etc...
GObj	gtkobj, widget, style
GData	data, adjustment, tooltips
GContainer	container, item_container
GWindow	window, dialog, color_selection_dialog, file_selection, plug
GPack	box, button_box, table, fixed, layout, packer, paned, notebook
GBin	scrolled_window, event_box, handle_box, frame, aspect_frame, viewport, socket
GButton	button, toggle_button, check_button, radio_button, toolbar
GMenu	menu_item, tearoff_item, check_menu_item, radio_menu_item, menu_shell, menu, option_menu, menu_bar, factory
GMisc	separator, statusbar, calendar, drawing_area, misc, arrow, image, pixmap, label, tips_query, color_selection, font_selection
GTree	tree_item, tree, view (also tree/list_store, model)
GList	list_item, liste, clist
GEdit	editable, entry, spin_button, combo
GRange	progress, progress_bar, range, scale, scrollbar
GText	view (also buffer, iter, mark, tag, tagtable)

Practically, each widget class is composed of :

- a `coerce` method, returning the object coerced to the type widget.
- an `as_widget` method, returning the raw Gtk widget used for packing, etc...
- a `destroy` method, sending the destroy signal to the object.
- a `get_oid` method, the equivalent of `00.id` for Gtk objects.
- a `connect` sub-object, allowing one to widget specific signals (this is what prevents width subtyping in subclasses.)
- a `misc` sub-object, giving access to miscellaneous functionality of the basic `gtkwidget` class, and a `misc#connect` sub-object.
- an `event` sub-object, for Xevent related functions (only if the widget has an Xwindow), and an `event#connect` sub-object.
- a `drag` sub-object, containing drag and drop functions, and a `drag#connect` sub-object.
- widget specific methods.

Here is a diagram of the structure (methods and sub-objects) :

```

method coerce : widget
method as_widget : Gtk.widget obj
method destroy : unit -> unit
method get_oid : int
method ...
val connect : mywidget_signals
| method after
| method signal_name : callback:(... -> ...) -> GtkSignal.id
val misc : misc_ops
| method show, hide, disconnect, ...
| val connect : misc_signals
val drag : drag_ops
| method ...
| val connect : drag_signals
val event : event_ops
| method add, ...
| val connect : event_signals

```

Pour créer un widget : `<Module>.<widget name> options ... ()`

Il y a beaucoup d'arguments optionnels, dont :

- `?packing` pour appeler une fonction sur votre nouveau widget ;
- et `?show` pour décider s'il doit être affiché tout de suite (défaut oui).

Exemple qui ajoute un bouton dans notre interface :

```
let main () =
```

```

let window = GWindow.window ~border_width:10 () in
let button = GButton.button ~label:"Hello World" ~packing:window#add () in
window#show ();
GMain.Main.main ()

```

```
let _ = main ()
```

Lorsqu'un événement se produit (par exemple un clic sur un bouton), un *signal* GTK<sup>1</sup> est émis par le widget. Pour associer des *callbacks* aux signaux :

```
widget#connect#signal_name ~callback:(unit -> unit) -> GtkSignal.id
```

Il y a aussi une notion d'*événements* pour tout ce qui concerne la fenêtre (focus, destroy, etc). On peut aussi y associer des *callbacks* :

```
widget#event#connect#event_signal_name ~callback:(event -> bool) -> GtkSignal.id
```

```
(* This is a callback function. *)
```

```
let hello () =
  print_endline "Hello World";
  flush stdout
```

```
(* Another callback function.
```

```
* If you return [false] in the "delete_event" signal handler,
* GTK will emit the "destroy" signal. Returning [true] means
* you don't want the window to be destroyed.
* This is useful for popping up 'are you sure you want to quit?'
* type dialogs. *)
```

```
let delete_event ev =
  print_endline "Delete event occurred";
  flush stdout;
```

```
(* Change [true] to [false] and the main window will be destroyed with
* a "delete event" *)
```

```
true
```

```
let destroy () = GMain.Main.quit ()
```

```
let main () =
```

```
  let window = GWindow.window ~border_width:10 () in
```

```
(* When the window is given the "delete_event" signal (this is given
* by the window manager, usually by the "close" option, or on the
* titlebar), we ask it to call the delete_event () function
* as defined above. *)
```

```
window#event#connect#delete ~callback:delete_event;
```

```
(* Here we connect the "destroy" event to a signal handler.
* This event occurs when we call window#destroy method
* or if we return [false] in the "delete_event" callback. *)
```

```
window#connect#destroy ~callback:destroy;
```

```
(* Creates a new button with the label "Hello World".
* and packs the button into the window (a gtk container). *)
```

```
let button = GButton.button ~label:"Hello World" ~packing:window#add () in
```

```
(* When the button receives the "clicked" signal, it will call the
* function hello(). The hello() function is defined above. *)
```

```
button#connect#clicked ~callback:hello;
```

```
(* This will cause the window to be destroyed by calling
* window#destroy () when "clicked". Again, the destroy
* signal could come from here, or the window manager. *)
```

```
button#connect#clicked ~callback>window#destroy;
```

```
(* The final step is to display the window. *)
```

---

1. notion différente de signal Unix

```
window#show ();

(* All GTK applications must have a GMain.Main.main (). Control ends here
 * and waits for an event to occur (like a key press or
 * mouse event). *)
GMain.Main.main ()

let _ = main ()
```

### 8.9.3 Glade

Une solution plus simple pour créer des interface est d'utiliser Glade. Glade permet de décrire l'interface sous la forme d'un fichier xml (qui peut éventuellement être créé à partir d'une interface graphique - cf programme glade).

Le programme lablgladec2 permet de générer le fichier ml à partir du fichier xml Glade.