8
Les objets

Les programmes tels que nous les avons décrits jusqu'à présent sont formés de déclarations de types, de déclarations de variables globales, de définitions de fonctions et d'un programme principal — la fonction `main`.

Comme nous l'avons vu, l'utilisation de fonctions permet de mieux structurer les logiciels puisqu'un programme constitué uniquement d'un programme principal de quelques milliers de lignes serait très difficile à lire et à comprendre. Cependant, le mécanisme des fonctions n'est pas suffisant pour structurer les longs programmes, qui deviennent, à leur tour, difficiles à lire quand ils dépassent quelques dizaines de fonctions. D'autres constructions, comme les *modules* et les *objets*, permettent de structurer les programmes davantage.

8.1 Les classes

8.1.1 Les fonctions font partie du type

Nous avons vu au chapitre 6 que, de même qu'une structure mathématique est définie, non seulement par un ensemble, mais également par des opérations sur cet ensemble, les notions de pile ou de file étaient définies, non seulement par un type, mais également par des opérations sur ce type. De ce fait, il serait raisonnable que les fonctions `push`, `pop`, ... fassent partie de la définition du type `Pile`. Un type ainsi muni de fonctions s'appelle une *classe* et les fonctions

s'appellent les *méthodes* de la classe.

Ainsi, au lieu de définir le type des piles, puis des fonctions, puis le programme principal

```
class Pile  {
 List c;

 Pile ...
}

class Prog {

 static Pile empty ...

 static boolean testempty ...

 static void push ...

 static void pop ...

 static int top ...

 static int f ...

 public static void main (String [] args) ...
}
```

on déplace les fonctions `push`, `pop`, ... dans la classe `Pile`

```
class Pile  {

 List c;

 Pile ...

 static Pile empty ...

 static boolean testempty ...

 static void push ...

 static void pop ...
```

```
 static int top ...
}

class Prog {

 static int f ...

 public static void main (String [] args) ...
}
```

Une classe est désormais constituée de trois choses : une liste de champs, une liste de constructeurs et une liste de méthodes.

Pour appeler la méthode `push`, on n'écrit plus simplement `push(p) ;` mais `Pile.push(p) ;`. Il est important de distinguer les deux usages du symbole `.` : quand on accède à un champ, on utilise la notation `p.c` où `p` est une expression de type `Pile`, alors que quand on accède à la méthode `push`, on utilise la notation `Pile.push` où `Pile` est la classe elle-même.

Quand on utilise une méthode d'une classe `T` à l'intérieur de cette même classe, il n'est pas nécessaire de faire précéder le nom de la méthode du nom de la classe. Ainsi, à l'intérieur de la classe `Pile`, le nom `push` est une abréviation pour le nom `Pile.push`.

8.1.2 La sémantique des classes

Pour étendre la fonction Σ, il est nécessaire que, dans la liste des classes, chaque classe soit associée à un triplet formé d'une liste de champs, d'une liste de constructeurs et d'une liste de méthodes.

Exercice 8.1

Définir la fonction Σ de manière à inclure l'appel de méthodes. On supposera, pour cet exercice, que les méthodes ne peuvent avoir accès à aucune variable globale.

8.2 Les méthodes dynamiques

Beaucoup de méthodes d'une classe `T` ont un objet de la classe `T` parmi leurs arguments. C'est le cas, par exemple, des méthodes `push` et `pop` de la classe `Pile`. Une telle méthode s'appelle par l'instruction ou l'expression

`T.f(`b_1`,...,a,...,`b_p`)` où `a` est un argument de type `T`.

Il est possible, lors de la définition d'une méthode de la classe `T`, de distinguer un tel argument de type `T` : on omet le mot clé `static` dans la définition de la méthode, on ne fait pas figurer d'argument formel pour l'argument distingué et on utilise, dans le corps de la fonction, une variable supplémentaire `this` pour cet argument distingué. Ensuite, on appelle cette méthode non par l'instruction ou l'expression `T.f(`b_1`,...,a,...,`b_p`)` mais par l'instruction ou l'expression `a.f(`b_1`,...,`b_p`)`. Une méthode ainsi définie s'appelle une méthode *dynamique*.

Par exemple, au lieu de définir les méthodes `push` et `pop` ainsi

```
static void push (final int a, final Pile l) {
 l.c = new List(a,l.c);}

static void pop (final Pile l) {
 l.c = l.c.tl;}
```

on peut les définir de manière plus concise comme des méthodes dynamiques

```
void push (final int a) {
 this.c = new List(a,this.c);}

void pop () {
 this.c = this.c.tl;}
```

On les appelle ensuite ainsi `p.pop() ; p.push(5) ;`. On peut ensuite exécuter le programme suivant

```
Pile p = Pile.empty ();
p.push(5);
p.push(6);
System.out.println(p.top());
p.pop();
System.out.println(p.top());
```

qui affiche `6` puis `5`.

En fait, il est possible d'être plus concis encore, car dans le corps d'une méthode dynamique de la classe `T`, si `c` est un champ de la classe `T` et qu'il n'y a pas de variable locale homonyme, l'expression `c` est une abréviation pour `this.c`. Ainsi, on peut définir les méthodes `push` et `pop` de la manière suivante

```
void push (final int a) {
 c = new List(a,c);}

void pop () {
 c = c.tl;}
```

Il est important de noter qu'une méthode dynamique ne peut être appelée que si l'objet avec laquelle on l'appelle n'est pas la valeur `null`. Ainsi, si `append` est une méthode dynamique, on ne peut écrire `l1.append(l2)` que quand la liste `l1` est non vide. Si l'on veut pouvoir appeler la fonction `append` y compris avec la liste vide, il faut ou bien garder une méthode statique, ou bien utiliser un type enveloppé. Par exemple, la pile vide est représentée, non par la valeur `null`, mais par un objet qui contient un champ `c` dont la valeur est `null`. De ce fait, même si la pile `p` est vide, l'instruction `p.push(5) ;` est correcte et ajoute la valeur `5` au sommet de la pile `p`.

Une erreur fréquente est d'écrire une méthode dynamique

```
List f () {
 if (this == null) ...}
```

En effet, le booléen `this == null` sera toujours égal à `false`, car le fait que la méthode `f` ait pu être appelée implique que l'objet d'appel soit différent de `null`.

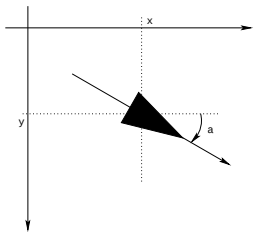
L'extension de la fonction Σ aux méthodes dynamiques n'est pas difficile, même s'il faut prendre garde à la confusion entre l'affectation `c = 1 ;` de la variable `c` et du champ `c` de l'objet `this`.

Exercice 8.2

Donner la définition de la fonction Σ, pour les méthodes dynamiques.

Exercice 8.3 (Une tortue Logo)

Dans le plan euclidien, un objet solide a trois degrés de liberté. Une tortue Logo est un objet à trois champs `x`, `y` et `a`. Les champs `x` et `y` sont les coordonnées de la tortue et `a` l'angle qu'elle forme avec l'axe des abscisses.



Une tortue a deux méthodes : `avance` qui prend un nombre `l` en argument et modifie la position de la tortue en la faisant avancer droit devant elle d'une longueur `l` et `tourne` qui prend un nombre `b` en argument et fait tourner la tortue sur elle-même d'un angle `b`. En outre, quand la tortue avance, elle trace un segment qui va de son point de départ à son point d'arrivée.

Que dessine le programme suivant ?

```
for (int i = 0; i < 4; i = i + 1) {r.avance(50); r.tourne(90);}
```

Écrire un programme qui dessine un pentagone régulier. Et un polygone régulier à `n` cotés.

Écrire un programme qui dessine le flocon de Von Koch — voir Exercice 3.6 — avec une tortue.

8.3 Les méthodes et les champs fonctionnels

On peut dire qu'une méthode dynamique est associée, non à une classe, mais à un objet d'une classe et que, quand on exécute l'instruction `p.push(5);`, on appelle la méthode `push` de la pile `p` et non de la classe `Pile`. Ainsi, on peut voir la pile `p` comme un enregistrement qui, en plus de son champ `c`, a des champs `push`, `pop` qui sont, non des valeurs, mais des fonctions. Un tel enregistrement dont certains champs sont des fonctions s'appelle un *objet* et les champs fonctionnels d'un objet s'appellent des méthodes.

Quand on voit les méthodes comme des champs fonctionnels, on comprend que l'objet `null`, qui n'a pas de champs, n'ait pas non plus de méthodes.

En principe, dans la classe `Pile`, deux objets `p` et `q` pourraient avoir des méthodes `push` totalement différentes, de même qu'ils peuvent avoir des champs `c` différents. Cependant, en Java, la dépendance de la méthode `push` par rapport à l'objet auquel elle appartient est limitée : c'est la même fonction qui s'exécute quand on appelle la méthode `push` de la pile `p` et de la pile `q`, la seule différence est que la variable `this` est égale à `p` dans le premier cas et à `q` dans le second.

Nous verrons cependant qu'une construction de Java, l'*héritage*, permet, dans certains cas, d'attribuer des méthodes différentes à deux objets d'une même classe.

8.4 Les champs statiques

Une méthode dynamique appartient donc à un objet. Une méthode statique, en revanche, appartient à une classe. C'est la même méthode pour tous les objets de la classe.

De même, un champ peut être statique ou dynamique. On déclare un champ statique en le faisant précéder du mot clé `static`. Quand on modifie un champ statique, il est modifié pour toute la classe. Ainsi, si l'on définit la classe

```
class M {
 static int mem;

 void modify (final int x) {
  mem = x;}

 void print () {
  System.out.println(mem);}}
```

le programme

```
M a = new M();
M b = new M();
a.modify(4);
b.modify(5);
a.print();
```

affiche `5` et non `4`. En effet, puisque le champ `mem` est statique, le corps `mem = x ;` de la méthode `modify` est une abréviation pour l'instruction `M.mem = x ;` et non `this.mem = x ;`.

8.5 Les classes entièrement statiques

Une classe est *entièrement statique* si tous ses champs et méthodes sont statiques. Les objets d'une telle classe n'ont ni champ ni méthode qui leur soit propre, et il n'y a pas d'intérêt à créer de tels objets.

En revanche, une telle classe est un moyen de regrouper ensemble des fonctions qui traitent d'un même sujet. Par exemple, la classe `Math` regroupe les fonctions mathématiques : `Math.sin`, `Math.cos`, ...

Les classes entièrement statiques permettent de regrouper ensemble des fonctions. Dans un langage de programmation, une telle construction qui permet de regrouper des fonctions s'appelle un *module*. En Java, les classes entièrement statiques jouent donc le rôle de modules.

En Java, le programme est lui-même une classe entièrement statique, ce qui explique que l'on écrive `class Prog` au début du programme `Prog`. Les variables globales du programme sont simplement les champs statiques de cette classe, ce qui explique qu'on les fasse précéder du mot clé `static`. Les fonctions du programme sont les méthodes de cette classe. La seule spécificité d'un programme est de contenir une méthode `main`.

Ainsi, alors que nous avions à la section 8.1.2 défini une fonction Σ à cinq arguments Σ`(p,e,m,G,C)` où `p` est une instruction, `e` un environnement, `m` une

mémoire, `G` un environnement global et `C` une liste de classe, on s'aperçoit que l'environnement global `G` n'est qu'une classe comme les autres, et qu'il doit faire partie de la liste `C`. La définition définitive de la fonction Σ est donc une fonction à quatre arguments Σ(`p,e,m,C`).

Exercice 8.4

Donner la définition de la fonction Σ en prenant en compte le fait que l'environnement global est une classe.

8.6 L'héritage

```
class Horloge {
 int h;
 int mn;

 Horloge () {}

 Horloge (final int x, final int y) {h = x; mn = y;}

 void drawCadran (final int x, final int y, final int r) {
  Ppl.drawCircle(x,y,r);}

 void dessineAiguille
       (final int x, final int y, final double l, final int a) {
  double b = (a - 15) * 3.1415926 / 30;
  int x2 = x + (int) Math.round(l * Math.cos(b));
  int y2 = y + (int) Math.round(l * Math.sin(b));
  Ppl.drawLine(x,y,x2,y2);}

 void dessinePetiteAiguille
       (final int x, final int y, final int r) {
  dessineAiguille(x,y,r * 0.5,h * 5 + mn / 12);}

 void dessineGrandeAiguille
       (final int x, final int y, final int r) {
  dessineAiguille(x,y,r * 0.7,mn);}

 void dessine (final int x, final int y, final int r) {
```

```
  dessineCadran(x,y,r);
  dessinePetiteAiguille(x,y,r);
  dessineGrandeAiguille(x,y,r);}}
```

Un objet de la classe `Horloge` est une date, par exemple `{h = 13, mn = 58}`. La méthode `dessine` dessine une horloge de rayon `r` et de centre `(x,y)`. Cette méthode en utilise d'autres : `dessineCadran` qui dessine un cercle, `dessineAiguille` qui dessine une aiguille de taille `l` et d'angle `a`, cet angle étant mesuré depuis la verticale et compté en soixantièmes de tour, `dessinePetiteAiguille` et `dessineGrandeAiguille`.

On veut maintenant définir une autre classe `HorlogeAvecSecondes` telle que les objets de cette classe soient des dates constituées de l'heure `h`, des minutes `mn` et des secondes `s`, par exemple `{h = 14, mn = 2, s = 30}`.

Parmi les différentes solutions pour définir ce type, une consiste à définir un type `HorlogeAvecSecondes` complètement indépendant

```
class HorlogeAvecSecondes {
 int h;
 int mn;
 int s;}
```

et une autre à définir une horloge avec les secondes comme un couple formé d'une horloge traditionnelle et d'un entier pour les secondes

```
class HorlogeAvecSecondes {
 Horloge ho;
 int s;}
```

Java comporte une construction qui permet de définir une telle classe qui étend une classe déjà définie en ajoutant des champs et des méthodes : l'*héritage*.

On définit la classe `HorlogeAvecSecondes` comme une extension de la classe `Horloge` avec le mot clé `extends`

```
class HorlogeAvecSecondes extends Horloge {
 int s;

 HorlogeAvecSecondes (final int x, final int y, final int z) {
  h = x; mn = y; s = z;}

 void dessineTrotteuse (final int x, final int y, final int r) {
  dessineAiguille(x,y,r * 0.8,s);}

 void dessine (final int x, final int y, final int r) {
```

```
  dessineCadran(x,y,r);
  dessinePetiteAiguille(x,y,r);
  dessineGrandeAiguille(x,y,r);
  dessineTrotteuse(x,y,r);}}
```

Tous les champs et méthodes de la classe `Horloge` sont hérités dans la classe étendue.

Il est également possible de redéfinir certaines méthodes, comme ici la méthode `dessine` qui doit être différente de la méthode héritée puisqu'elle doit dessiner la trotteuse également.

Toutes les expressions de type `HorlogeAvecSecondes` sont également de type `Horloge`. Si `h` est une expression de type `Horloge` construite avec le constructeur `Horloge` et `k` est une expression de type `Horloge` construite avec le constructeur `HorlogeAvecSecondes`, les instructions `h.dessine(40,40,30);` et `k.dessine(40,40,30);` dessinent des horloges sans la trotteuse dans le premier cas et avec la trotteuse dans le second

L'héritage permet donc de donner à ces deux objets des méthodes `dessine` différentes.

Pour terminer cette courte introduction à la notion d'héritage, dont les multiples subtilités pourraient occuper un volume entier, voyons comment elle permet d'exprimer d'une manière différente les types disjonctifs.

Exercice 8.5 (Les types disjonctifs)

On rappelle qu'une expression arithmétique est ou bien une constante, ou bien une variable, ou bien la somme de deux expressions arithmétiques, ou bien le produit de deux expressions arithmétiques.

On définit un type `Expr` sans champs ni constructeurs, mais avec une méthode `print`, dont le corps importe peu car on verra qu'il ne sera jamais exécuté.

```
class Expr {

 void print () {System.out.println("Bonjour");}}
```

1. Définir quatre classes `Constant`, `Variable`, `Sum`, `Product` qui étendent la classe `Expr` avec respectivement un champ de type `int`, un champ de type `String`, deux champs de type `Expr` et deux champs de type `Expr` et redéfinissent la méthode `print`.

2. Définir la valeur de type `Expr` correspondant à l'expression `x + 3`. Imprimer cette expression.
3. Pourquoi l'instruction `System.out.println("Bonjour");` n'est-elle jamais exécutée ?
4. Que se passe-t-il si l'on supprime la classe `Expr` et que l'on ne garde que les quatre classes `Constant`, `Variable`, `Sum`, `Product` ? Que se passe-t-il si l'on garde la classe `Expr` mais supprime la méthode `print` ?
5. Écrire une méthode qui dérive une expression arithmétique par rapport à une variable.
6. Comment faire pour ajouter un nouveau cas, par exemple la soustraction, aux expressions arithmétiques ?

Il y a donc essentiellement quatre manières de définir des types disjonctifs dans un langage de programmation. La première n'est possible que quand on cherche à définir la disjonction d'un produit cartésien avec un singleton — comme dans le cas des listes —, elle consiste à identifier l'élément du singleton avec la valeur `null`, si le langage comporte une telle valeur. La deuxième est d'utiliser un champ sélecteur, à condition que le langage permette de construire une valeur par défaut dans chaque type. La troisième est d'utiliser une construction primitive pour les types disjonctifs, si le langage en comporte une. La quatrième consiste à utiliser l'héritage, si le langage comporte une telle construction.

8.7 Caml

Le mécanisme des objets de Caml présente quelques différences avec celui de Java. Alors qu'en Java les enregistrements sont des objets particuliers, en Caml les enregistrements et les objets sont des constructions différentes. Une autre différence est que, en Caml, seules ses méthodes peuvent accéder aux champs d'un objet. On doit donc écrire une méthode d'accès, comme la méthode `get_c` *ci-dessous, si l'on veut pouvoir accéder à un champ d'un objet ailleurs que dans l'une de ses méthodes. Une dernière différence est qu'il n'y a pas de méthodes statiques en Caml.*

Comme en Java, chaque champ peut être final ou mutable, ce que l'on indique par le mot clé `mutable`*. On crée un nouvel objet avec le mot clé* `new` *et on accède à ses méthodes en utilisant le symbole* `#`*.*

La classe des piles se définit, par exemple, ainsi

```
class pile = object
```

```
 val mutable c = ([]:int list)
 method get_c () = c
 method testempty () = c = []
 method push x = c <- x::c
 method pop () = c <- List.tl c
 method top () = List.hd c
end
```

Comme il n'y a pas de méthodes statiques en Caml, la fonction `empty` *qui crée la pile vide ne peut pas être une méthode de la classe* `pile` *et on la définit hors de la classe*

```
let empty () = new pile
```

Enfin on peut utiliser ces méthodes et cette fonction dans un programme

```
let p = empty () in
(p#push 5;
 p#push 6;
 print_int (p#top());
 print_newline();
 p#pop();
 print_int (p#top());
 print_newline())
```

Il n'y a pas d'objets en C. Mais il y en a dans un langage issu de C appelé C++.