Programmation Par Objets et Langage Java
Partie I. Fondement de la POO
(Modularité et Abstraction)

Najib Tounsi

Lien permanent : http://www.mescours.ma/Java/PooJavaPart-1-tdm.html

Table des matières

Modularité/Abstraction

Modularité

Programmation par morceaux appelés **modules**.

Module qui implémente (variables et codes) une **fonctionnalité bien définie** et dont la description est faite à travers une interface **précise** de sorte qu'on *n'a pas* besoin de connaître son code pour utiliser un module.

Cette décomposition de logiciel en modules

- facilite le développement d'un logiciel
- ... la localisation et le correction des erreurs
- limite leur impact
- et permet la réutilisation des modules

Reste à savoir décomposer en modules : comment?

...Modularité/Abstraction

Abstraction

L'abstraction tente de réduire les détails pour se concentrer sur les concepts importants.

> L'abstraction est d'ailleurs l'idée importante derrière la notion de type. On ne manipule pas les nombres comme les chaînes de caractères. On n'utilise pas un entier comme un réel...

En poussant cette idée plus loin, **L'abstraction de données** permet la programmation sans se préoccuper des détails de représentation des objets manipulés, détails qui sont définis séparément (ou précisés ultérieurement). L'important c'est de savoir *quoi* faire avec un objet.

Cette **séparation** entre utilisation et représentation permet

- de changer de représentation sans changer les aspects significatifs d'un programme.
  - Un point du plan peut être représenté (dans un programme) en coordonnées polaires ou cartésiennes, il restera toujours un point qu'on peut *tracer*, *déplacer*, *imprimer* ses coordonnées etc.
- de partager des définitions (interfaces communes) par une famille d'objets qui ont des nuances dans la représentation ou le comportement, tout en ayant une même signification.
  - Par exemple, des *figures géométriques* ont toutes une aire et un pourtour, mais leur calcul est différent selon que c'est un *carré*, un *cercle* ou autre.

Exemples Introductifs

Programmer sans abstraction de données.

On va énumérer quelques exemples qui illustrent comment on peut améliorer et faciliter l'écriture de programme en faisant abstraction des détails de représentation des données.

Programmer sans abstraction: Exemple 1

Soit le programme qui lit une chaîne de caractères et l'imprime en ordre inverse (image miroir)..

(Exemple1, Le programme lit et range des caractères dans un tableau, et les imprime ensuite en parcourant le tableau en ordre inverse.)

```
#include <stdio.h>
#define MAX 80
main(){
    int curChar=0;
    char c, t[MAX];

    /* On lit et mémorise des caractères */
    while ((c=getchar())!='\n') {
```

```
        t[curChar++] = c;
        if(curChar==MAX) break;
    }

    /* On les imprime en ordre inverse */
    while (curChar>0)
        putchar( t[--curChar] );
    }
```

Point à noter: On a choisi un tableau, on manipule directement le tableau. (Autre méthode)
Programme développé par raffinements successifs (on définit les grandes lignes et on les dégrossit au fur et à mesure).

Programmer sans abstraction: Exemple 2

Soit maintenant à réutiliser/adapter programme pour tester si la chaîne est un palindrome (le problème est proche, l'image miroir d'une chaîne lui est identique).

(Exemple2)

```
#include <stdio.h>
#define MAX 80
char t[MAX], s[MAX];
main(){
    int curChar=0, i=0;
    char c;

    /* On lit et mémorise des caractères */
    while ((c=getchar())!='\n') {
        t[curChar++]=c;
        if(curChar==MAX) break;
     }

     /* On duplique la chaîne en ordre inverse */
     i = 0;
     while (curChar>0)
           s[i++] = t[--curChar];

     /* et on l'imprime pour voir (ou on teste l'identité des deux) */
      printf("%s %s\n", t, s);
 }
```

A noter: Même remarque. On a utilisé un deuxième tableau pour comparer. On aurait pu tester sur le même tableau... Mais cela reste un programme dépendant de la notation tableau!

Programmer sans abstraction: Exemple 3

Soit maintenant un programme qui lit une "expression arithmétique" et vérifie si elle est bien parenthésée (symboles miroirs en correspondances).

(a * (c + d) - (-5))

Réutiliser/modifier le même programme devient plus difficile. Mais le problème est proche. (Exemple3)

```
        #include <stdio.h>
        #define MAX 80
```

```
char t[MAX];
main(){
      int curChar=0;
      int nbO=0; // nombre parenthèses Ouvrantes
      int nbF=0; // nombre parenthèses Fermantes
      char c;
      int i;
      /* On lit et mémorise uniquement les parenthèses */
      while ((c=getchar())!='\n') {
            if (c=='(' || c==')' )
                  t[curChar++] = c;
            else;
            if(curChar==MAX) break;
       }

     /* On parcourt le tableau est compte les parenthèses */
       for (i=0; i<curChar; i++){
            if (t[i] == '(') nbO++;
            else {
               nbF++;
               if (nbF >nbO) break;
            }
        }
        if (i<curChar || nbO != nbF)
             printf ("Mal Parenthésée :-(");
        else
             printf ("bien Parenthésée :-)");
  }
```

A noter: Cela reste un programme dépendant des détails de sa structure de données : on est contraint de penser et coder en terme de tableau avec la notation indicée.

Programmer sans abstraction: Exemple 4

Problème: Vérifier si une expression générale est bien parenthésée.

avec **{ ([< >]) }**

Reprendre le programme précédent et l'adapter se complique. Il faut chercher une autre idée.

L'idée c'est de ***mémoriser dans leur ordre d'arrivée*** les symboles ouvrants au fur et à mesure de leur rencontre, et de vérifier si chaque symbole fermant rencontré correspond bien au ***dernier symbole ouvrant mémorisé*** (lequel sera ensuite retiré). C'est la notion de ***pile***.

Programme avec abstraction (Pile)

Considérer les données sous la forme d'une pile, et raisonner avec la pile.

Exemple-3 résolu avec la pile

A la place d'un tableau `t [MAX]` des programmes précédents, on va déclarer *une abstraction* PILE matérialisée avec un objet `p`.

(Exemple3-pile).

```
#include <stdlib.h>
#include "pile.h"          // Voir plus loin
#define MAX 20
main(){
    int curChar=0;
    char c; int b=0;
    PILE p; // a la place t[MAX]

    p = create ();

    /* On lit et empile (resp. depile) les parenthèses
     * ouvrantes (resp. fermantes)
     */
    while ((c=getchar())!='\n') {
          if (c=='(') empiler (p, c); /* On mémorise */
          else if (c==')')
               if(estVide(p)) b=1;
               else if (sommet(p) != '(') b=1;
                    /* on vérifie la correspondance */
                    else depiler (p);  /* On retire */
               else;
          if(curChar==MAX) break;
     }

    /* On teste si la pile est vide et b=0. Cas favorable. */

     if (estVide(p) && b==0)
          printf ("bien Parenthésée :-)");
     else printf ("Mal Parenthésée :-(");
 }
```

Points A noter :

- On conçoit son programme à un niveau abstrait
- On ne se préoccupe pas des détails dus à une structure particulière de données (e.g. tableau...)
- *P* est un objet abstrait.
- On le connaît à travers les opérations que l'on peut y effectuer : *empiler*(), *sommet*(), *depiler*() ...
- L'algorithme est dirigé par cette conception de l'objet de type *Pile*.
- On dit un ***Type Abstrait de Données.***

Type Abstrait de Données Pile

La définition de la Pile (en C)

```
/*
 Type Abstrait PILE de caractères
 CopyLeft: :-)
 Author: Najib Tounsi
 Date: 06/10/04 00:19
 Description: TAD pile en langage C
 */

/*
 * Structure de données d'une pile
 */

#define MAX 20
typedef struct {
    char t[MAX]; /* tableau des éléments empilés */
    int top;     /* indice du sommet de la pile */
} * PILE;
```

```
/*
 * Interface abstraite : Liste des opérations que l'on fait sur une pile
 (déclarations des profiles)
 */

PILE create();
   /* crée et initialise une pile vide */

void empile(PILE, char);
   /* empile le caractère donné */

char sommet(PILE);
   /* retourne le caractère au sommet de la pile */

void depiler(PILE);
   /* décapite la pile (retire le sommet ) */

int estVide(PILE);
   /* teste si la pile est vide */

int estPleine(PILE);
   /* teste si la pile est pleine */



/*
 * Représentation (Implementation)
 * On choisit une structure de données
 * et on programme les opérations
 */

/* cf. structure de données plus haut */
/* Programmation des opérations */

PILE create(){
  /* On crée une pile et on retourne son adresse */
    PILE p = (PILE) malloc(MAX);
    p->top = -1;
    return p;
}

void empiler(PILE p, char c){
    if (!estPleine(p))
       p->t[++p->top] = c;
    else
       printf("Pile Pleine : %c\n", p->t[p->top]);
}

char sommet(PILE p){
     return p->t[p->top];
}

void depiler(PILE p){
    if (!estVide(p))
        p->top--;
    else
        printf("Pile vide\n");
}

int estVide(PILE p){
    return (p->top < 0);
}

int estPleine(PILE p){
    return (p->top >= (MAX - 1));
}
```

Type Abstrait de Données Pile

On a deux parties :

- La *description des opérations* qu'on fait sur une pile (sous forme de déclaration de fonctions)
  - Avec commentaires ;-)
- La *description de l'algorithme de ces opérations* (ici définition de fonctions) et la spécification de la *représentation choisie* pour une pile (ici structure C avec un tableau)

Tout ce qui dépend du "tableau" est alors ***confiné*** dans les définitions de fonctions. Donc ***localisé***.

Type Abstrait de Données Pile: Intérêt

Abstraction

- Le programme *main* ici (ou un autre programme) est développé sans être contraint par les détails de représentation des objets qu'il manipule.
  - La mise au point est facilitée.
  - Le programme peut évoluer (ou être *réutilisé*) vers la solution d'un problème proche sans grande difficulté.
  - Les erreurs peuvent facilement être localisées.
  - L'évolutivité aussi est facilité
  - On peut par exemple tester si une expression générale est bien parenthésée.
- C'est parce qu'on a conçu la solution du problème en terme de données, plutôt qu'en terme de traitements à effectuer.

Modularité

- *Pour maîtriser la complexité d'un système logiciel, il est nécessaire de le décomposer en éléments plus simples, appelés* ***modules****. Le système sera vu comme composé de plusieurs unités, chacune réalisant une tâche bien spécifique.*

- La programmation par abstraction est implicitement "modularisable" en plusieurs parties logiques (et peut en plus être réparties et plusieurs fichiers sources)

Type Abstrait de Données

> On appelle ***type abstrait de données*** (TAD) un ensemble d'objets caractérisés par les opérations qui leur sont tous applicables. Ang. *Abstract Data Type* (ADT). Exemples :

- Pile
  - *empiler*(), *dépiler*(), *sommet*() ...
- Rectangle
  - *tracer*(), *périmètre*(), *surface*(), *zoomer*()...
- Fenêtre
  - *afficher*(), *cacher*(), *agrandir*(), *deplacer*()...
- Collection
  - *ajouter*(), *supprimer*(), *appartient*()...

Ce qui importe, c'est le concept véhiculé par un objet (et non la structure de données sous-jacente)

/* C'est plus stable */

Spécification d'un TAD

- Aspect Syntaxique
  - Profile des opérations (entête, prototype)
    - Ex. *Plus*: $\mathbb{N} \times \mathbb{N} \rightarrow \mathbb{N}$
- Aspect Sémantique
  - Sémantique des opérations, i.e. sens / comportement du type de donnée.
    - Ex. *Plus* (*Succ* (*x*), *y*)) = *Succ* (*Plus*(*x*, *y*))
- Représentation
  - Choix d'implantation (algorithme des opérations et déclaration de données)

Aspect Syntaxique d'un TAD

Profile des opérations (ou entêtes)

Soit **B** (les booléens), **P** (les piles) et **N** (les entiers)

```
Type P =
 vide :              →  P
          /* Crée une pile vide et la retourne en résultat.*/
 sommet  :   P       → ℕ
 emPiler :   P x ℕ   → P
 dépiler :   P       → P
 pileVide:   P       → B
 pilePleine: P       → B
Fin
```

Aspect sémantique d'un TAD

Spécification du comportement (Fait partie de la description d'un TAD)

- **Par Commentaires**: Verbosité, mais simple et facile (Clarté ?)
  - *vide* () : Crée une pile vide et la retourne en résultat.
  - *empiler* (*p*, *n*) : Rajoute l'entier n au sommet de la pile p, et la retourne en résultat.
  - *pileVide* (*p*) : Teste si la pile p est vide.
  - etc.

- **Pré/Post conditions**:
  - Ce qu'exige une opération, et ce qu'elle garantit
  - Pas toujours faciles mais plus formelles.
  - *Empiler* (*p*, *n*)
    - **Requiert**: Non *PilePleine*(*p*)
    - **Garantit**: Non *PileVide*(*p*), *Sommet*(*p*) = *n*

  cf. Programmation par contrat, langage Eiffel, ou assertions de Java.

- **Par Axiomes:** On dit aussi de *Spécifications algébriques*. (Formelles)

Dépiler (*Vide* ()) = ERREUR
Dépiler (*Empiler* (*p*, *n*)) = *p*
Sommet (*Vide* ()) = ERREUR
Sommet (*Empiler* (*p*, *n*)) = *n*
PileVide (*Vide*()) = VRAI
PileVide (*Empiler* (*p*, *n*)) = FAUX

Les axiomes traduisent les invariants qui caractérisent les propriétés d'un objet.

Notion de Constructeur

Opérations permettant d'obtenir (construire) tous objets potentiels d'un certain type.

Les axiomes spécifient l'effet de certaines opérations (dites sélecteurs ou fonctions d'accès) sur d'autres (dites ***constructeurs***)

Constructeurs pour le TAD *Pile*:

- *vide*()
- *empiler*()

Exemples:

- *empiler* (*empiler* (*empiler* (*vide*(), 1), 4), 3)
- *empiler* (*empiler* ((*vide*(), 1), 4)
- *empiler* (*empiler* (*empiler* (*vide*(), 1), 4), 5)

☞ Notion importante reprise dans les langages à objets.

Autre exemple de TAD. Une date

Objet *Date* (constituée de 3 entiers)

```
Type Date =

Opérations
 Date   : ℕ X ℕ X ℕ → Date
 Jour   : Date      →  ℕ
 Mois   : Date      →  ℕ
 Année  : Date      →  ℕ
 Demain : Date      →  Date

Axiomes
 Jour ( Date ( j, m, a))    =  j;
 Mois ( Date ( j, m, a))    =  m;
 Année ( Date ( j, m, a))   =  a;
 Demain ( Date ( j, m, a)) =  si j< 28 alors Date(j+1, m, a)
                              sinon /* faire le plus dur ...*/
                              Date(1, m+1, a) ...
Fin
```

Remarque : Pour plus d'abstraction, introduire les TADs *Jour*/*Mois*/*Année* au lieu des trois entiers.

Exercices:

- Quelles seraient les pré/post conditions de l'exemple *Date*.
- Définir les TADs correspondant aux objets :
  - Collection (d'entiers)
  - Un article de commerce
  - Un autre TAD de votre choix (inspiré de l'informatique matériel/logiciel)
  - Entiers naturels (utiliser axiomes de Peano)
  - Indiquer les constructeurs éventuels.

Autre exemple de TAD. Une collection

```
Type Collection C =
Opérations
 Vide       :       → C
 Ajouter    : C X ℕ → C
 Appartient : C X ℕ → B
 Supprimer  : C X ℕ → C
 EstVide    : C     → B

Axiomes
 /* Ensemble sans double */

 Collection c; x, y entiers */
 Appartient ( Vide(), x) = Faux
 Appartient ( Ajouter (c, y), x) = si (x=y) alors Vrai
                                   sinon Appartient( c, x)
 Supprimer ( Vide(), x) = Vide()
 Supprimer ( Ajouter (c, y), x) = si (x=y) alors Supprimer (c, x)
                                  sinon Ajouter ( Supprimer (c, x), y)
 EstVide ( Vide()) = Vrai
 EstVide ( Ajouter (c, y)) = Faux

Fin
```

Exercice: Collection avec doubles?

Représentation de TAD

- Choix d'implémentation du TAD (concrétisation).

Structure de données + algorithmes des opérations

```
Type Pile:
 Représentation
    tableau t [MAX];   /* Tableau pour les éléments de la pile*/
    entier top;        /* Indice du sommet de la pile +/

 opérations
        Vide (p) { p.top = -1; }
        Empiler (p, n) { p.top = p.top+1;
             p.t[p.top] = n;
         }
        Depiler(p) { p.top = p.top-1; }
        Sommet(p) { return p.t[p.top]; }
        PilePleine(p) { return p.top = (MAX-1); }
        PileVide(p) { return p.top < 0; }
  fin
```

Remarque: Prise en compte des cas d'exceptions dans les algorithmes (spécification. sans *Pré/post*):

On teste les cas d'erreurs à l'intérieur des opérations

```
Empiler (p, n) {
    if top<(MAX)
          p.top = p.top+1;
          p.t[p.top] = n;
   else
         erreur("pile pleine");
    endif
}
```

où *erreur*() est une fonction donnée de traitement d'erreurs.

(Mais voir plus tard les *exceptions*)

Exercices:

1. Programmer le reste des opérations.
2. Choisir et programmer la représentation des TADs des exercices précédents.

En Résumé

Au lieu de programmer avec

t [*expression*], *top*++, ...

on fait

empiler (*p*, *c*), *sommet* (*p*) etc.

On ne connaît pas comment est fait l'objet *p*. On a uniquement besoin de savoir avec quoi le manipuler (quelles opérations?). On appelle cela **Interface** de l'objet

- *p* est donc comme une **boite noire** avec des boutons (qui sont les opérations qu'on appelle).
- Si on change le contenu de la boite, on ne change pas ses programmes.
  - On peut définir la pile (i.e. fichier `pile.h`) différemment ou améliorer les algorithmes, on n'a pas besoin de refaire les programmes qui l'utilisent. Tant qu'on n'a pas touché à l'entête des fonctions, i.e. l'interface de la pile).
- On peut en avoir plusieurs versions.
  - On peut donc définir la pile de plusieurs façons et choisir celle qui convient à un cas d'utilisation le moment venu.
- En plus, on peut *réutiliser* cette boite dans plusieurs systèmes qui ont en besoin. Sa spécification est parfaitement déterminée.

Notion d'encapsulation

- Le TAD Pile (boîte noire) **encapsule** en son sein (cf. fichier `pile.h`) les variables représentant un objet pile et les instructions qui les utilisent dans les opérations sur l'objet.
- En dehors de ce fichier ces variables *ne sont pas connues*.

    - En fait, il vaut mieux qu'on y accède pas, pour les protéger de toute fausse manipulation.
- On dit aussi ***Information Hiding***.
- Critère très important
    - mise au point et maintenance facilitées
    - Les corrections/modifications sont *localisées*.