TP Java

1 Introduction

Ce document vise à donner les bases en Java nécessaires au projet de Programmation du second semestre. Il donne quelques pistes sur la programmation orientée objet et la syntaxe Java. Les exercices proposés dans la partie 5 visent à faire découvrir par la pratique les notions (intuitives) de l'approche objet. *Ceci n'est pas un cours de conception objet (qui sera dispensé dans le cours de Programmation).*

1.1 Quelques aspects de génie logiciel

Le logiciel est souvent l'aspect critique des systèmes actuels, dans la mesure où les dépassements de budget et/ou de planning lui sont souvent imputables . Alors que les autres domaines d'ingénierie (mécanique, électronique, ...) sont matures et plutôt bien maîtrisés, la production logicielle en est encore à un stade presque artisanal. Faire un petit programme est facile, maintenir un gros programme ou construire du code à moindre coût à partir de code existant l'est moins.

Une des pistes suivie pour pallier ces problèmes a été l'amélioration des langages de programmation. Ainsi alors que les langages de première génération (1950) sont du code binaire, fortement dépendant de l'architectre matérielle et difficile à maintenir/étendre, les langages de troisième génération (1970) comme C ou Pascal offrent des structures de données évoluées organisées en modules.

Cependant les problèmes rencontrés pour les grands projets sont encore nombreux. On peut citer particulièrement :

- la vitesse de développement,
- la portabilité,
- la maintenance (qui peut coûter plus cher que le développement),
- la fiabilité.

1.2 Origine de Java

Java est un langage orienté objet récent (1995) conçu par Sun. Il a été désigné pour des systèmes embarqués ou distribués. Aux points mentionnés plus haut, Java apporte les solutions suivantes.

vitesse de développement

- réutilisabilité du code par *approche objet* (encapsulation, généricité, héritage),
- gestion primitive de mécanismes évolués (exemple : threads)
- très bon support de Sun (nombreuses bibliotèques, tutoriels, ...),

portabilité utilisation d'une *machine virtuelle* pour du code indépendant du matériel,

maintenance encapsulation des données,

fiabilité

- typage fort,
- politique d'accès des données,
- gestion mémoire transparente.

Les principaux points noirs de Java sont la vitesse d'exécution et la consommation mémoire.

1.3 Approche objet

L'approche orientée objet permet la réutilisabilité du code et assure (avec d'autres mécanismes) la sécurité. Le principe de base est l'*encapsulation des données* : les détails internes des structures de données sont cachés à l'utilisateur, seuls les traitements (fonctions) utiles sont accessibles. La généricité et l'héritage sont deux autres possibilités importantes qui permettent de réutiliser du code facilement. Cette approche est décrite en 3.5.

1.4 Machine virtuelle

La portabilité du code est obtenue en compilant le programme Java en *bytecode*. Ce bytecode est le langage d'entrée d'une *machine virtuelle Java* (JVM). On peut voir la machine virtuelle comme un modèle d'architecture matérielle, et le bytecode comme un assembleur abstrait. La machine virtuelle traduit au vol (langage interprété) l'assembleur abstrait en un assembleur spécifique à l'architecture utilisée. De cette façon, le bytecode est commun à toutes les architectures, seule l'implantation de la machine virtuelle change d'une plateforme à l'autre.

2 L'approche objet

2.1 Principes

La conception orientée objet consiste à décomposer une appplication en un certain nombre de composants autonomes (*objets*) qui vont collaborer. Les objets sont à penser en termes *d'interface publique*, c'est à dire ce qu'ils savent faire (*le service qu'ils rendent*), et non en termes d'implantation interne (*comment ils le font*). Ainsi on peut remplacer facilement dans une application un objet par un autre objet ayant une interface identique (proposant les mêmes services).

Il faut différencier la conception objet de la programmation objet. Ainsi on peut très bien faire une application orientée objet avec un langage de programmation comme C. Le principe de base de l'approche objet est une conception modulaire rigoureuse. Les langages de programmation orientés objet fournissent cependant des facilités pour faire une application orientée objet.

2.2 Encapsulation des données

En terminologie objet, les *classes* sont les types. Un objet est une instance d'une classe. Chaque objet possède ses propres

- *membres* (données), ce qui le définit,
- *méthodes* (fonctions), ce qu'il sait faire.

Cependant tous ces membres et méthodes ne sont pas accessibles (*visibles*) à l'utilisateur [1]. Normalement les membres sont cachés (*masqués*) et seuls les méthodes importantes sont visibles. C'est la notion de service rendu évoquée plus haut.

Exemples :

[1] Le terme *utilisateur* désigne tout ce qui est extérieur à l'objet, typiquement un autre objet

- *un objet Raytracer a une seule méthode accessible, qui prend en entrée une description textuelle d'image et retourne l'image calculée sous un format prédéfini. L'utilisateur n'a pas besoin de savoir autre chose.*
- *un objet Horloge doit avoir une méthode qui donne l'heure, mais peu importe qu'elle prenne l'horloge du PC ou qu'elle se synchronise sur un site internet.*

2.3 Restriction de l'accès des données

On vient de distinguer deux niveaux différents d'accessiblité : *publique* (l'utilisateur peut y accéder) ou *privé* (interne à l'objet). Les langages orientés objet offrent d'autres niveaux d'accès, variables d'un langage à l'autre. Ceux spécifiques à Java sont précisés dans la section correspondante.

2.4 Membres et méthodes de classe

Parfois tous les objets d'une classe partagent la même information. Typiquement des constantes. Dans ce cas on peut partager cette information par des membres non pas propres à un objet, mais propre à la classe. On parle de *membre de classe*. De la même façon on trouve des méthodes de classes. Typiquement pour modifier ou retourner la valeur d'un membre de classe. Les restrictions d'accès s'appliquent aussi aux membres et méthodes de classes.

2.5 Héritage et réutilisation de code

Une fonctionnalité importante des langages objets est *l'héritage*. Dire que la *classe fille* F hérite de la *classe mère* M signifie que F propose toutes les fonctionalités de M, plus éventuellement des fonctionalités propres. Ainsi toute méthode utilisée avec un objet de classe M s'appliquera aussi à un objet de classe F.

Différence avec le sous-typage : attention, la notion de sous-classe est beaucoup plus syntaxique que la notion de sous-type. Cela signifie seulement que l'objet propose des méthodes avec les mêmes noms et les mêmes arguments, mais pas que ces méthodes font la même chose (même si c'est souvent le cas en pratique).

Intérêts pratiques : il y a deux grandes utilisations. Tout d'abord, le code est facile à étendre, puisque pour coder une classe F qui descend de M, il suffit de déclarer que F descend de M et coder les différences. Ensuite comme toute classe fille F peut être vue comme appartenant à sa classe mère, on peut faire des méthodes génériques qui travaillent sur une famille de classes, et pas sur une seule classe.

Exemple : on a une classe Point2D définie par son abscisse et son ordonnée, pour laquelle des opérations de base (translation, rotation, ...) sont disponibles. On veut écrire une classe Pixel2D, définie par une abscisse, une ordonnée et une couleur. On peut bien sûr tout recoder, ou encore dire qu'un Pixel2D est une union {Point2D,couleur} et recoder les opérations en appelant celles de Point2D. Mais en disant que Pixel2D descend de Point2D, on récupère automatiquement le code de Point2D et on a juste à coder la gestion de la couleur.

2.6 Classes abstraites et code générique

Comme les objets sont définis, pour l'extérieur, par le service qu'ils rendent, il est possible de définir des méthodes sur des *objets abstraits*, pour lesquels on a spécifié le service rendu, mais pas comment le réaliser ! ! Pour utiliser en pratique ces méthodes, on crée une classe concrète FC qui descend de la classe mère abstraite MA, et qui implante effectivement les méthodes abstraites de MA.

Exemple : on définit une classe abstraite Objet_ Ordonné, qui contient une seule méthode $\leq$. On écrit une fois pour toute des algorithmes de tri pour des listes de Objet_ Ordonné. Ensuite ces algorithmes seront utilisables pour toute classe qui aura une méthode $\leq$ bien définie.

3 Java

Java manipule deux sortes de types : les types de base et les objets. Les types de base sont passés par *valeur* tandis ques les objets sont passés par *référence*.

3.1 Types de base

Dans cette partie on présente les types de base, ainsi que deux types d'objet (array et string), qui bien que considérés comme des objets par le compilateur, se manipulent comme des types de base.

Les types de base sont en nombre fini, on ne peut pas en créer. Les types de base sont :
- boolean (true et false),
- char (les caractêres),
- byte (8 bits), short (16 bits), int (32 bits), long (64 bits) (quatre types d'entiers) ,
- float (32 bits), double (64 bits) (deux types de flottants).

3.1.1 Passage par valeur

Lors de l'appel d'une fonction, les types de base sont *passés par valeur*, c-à-d que la variable est déjà copiée avant d'être passée en argument. Une fonction ne peut donc modifier la valeur d'un de ses arguments, puisqu'elle ne travaille que sur une copie de celui-ci.

3.1.2 Déclaration et initialisation

Les variables sont déclarées comme en C. Par exemple :
- `int truc ;`
- `double d1,d2 ;`
- `boolean estVrai ;`
- `char a ;`

On peut aussi passer une valeur à la déclaration.
- `int truc=42 ;`
- `double d1=3.14,d2=2*3.14 ;`
- `boolean estVrai=true ;`
- `char a='a' ;`

Si aucune initialisation n'est donnée, les types numériques sont initialisés à 0, les caractères au caractère nul et les booléens à false.

3.1.3 Opérations de base

pour tous

- **affectation :** `a=b`
- **égalité :** `a==b`
- différence : `a != b`
- caster le type : `(newtype) variable ;` modifie le type, attention

types numériques `+,-,*,/,%,++,-, >=,<=,<,>, ...`

boolean & (et), | (ou), ! (non)

3.2 Flot de contrôle

Une instruction se termine par un " ;".

Lorsqu'on veut grouper plusieurs instructions on les met entre "{}" : `instruction1 ;instruction2 ;... ;intructionk ;`

Les structures disponibles sont

```
if (condition) instructions else instructions
if (condition) instructions

while (condition) instructions

for (initialisation ; condition ; incrementation) instructions
```

Exemple

```
if (monentier==0) monentier=monentier+1; else monentier=monentier-1;

while (monentier!=0) {monentier=monentier-1; monbooleen=!monbooleen;}

for (int i=0 ; i=10 ; i++) {...}
```

3.3 Arrays

On crée les tableaux presque comme en C. On parle ici de tableaux avec un espace mémoire fixé.

déclaration `int[] montableau = new int [42] ;` crée un tableau de 42 entiers

accès à l'élément numéro i `montableau[i]`

affectation `montableau[4]=2 ;`

accès à la taille `montableau.length ;`

tableau à plusieurs dimensions `int[][] montableautableau = new int [42][24] ;`

3.4 Strings

Les objets String ont deux facilités qui les font ressembler à des types de base :
- on peut créer un String sans passer directement par un constructeur. En effet `"blabla"` est un String.
- ils ont un opérateur propre, +, qui permet la concaténation.

Beaucoup d'objets ont une méthode `toString()` qui renvoie le String correspondant.

3.5 Les objets

On peut créer de nouveaux objets. Lors des passages d'arguments, les objets sont passés par référence. Une référence est un identifiant (unique) qui permet d'accéder à l'objet référencé. On peut donc le modifier. En Java l'utilisation des références est transparente.

3.5.1 Null

`null` indique une référence sur rien. Tenter d'accéder aux membres d'une référence nulle provoque une exception. On peut tester si une référence vaut `null`. Un objet juste déclaré est mis à `null`.

3.5.2 Utilisation

Une méthode s'utilise avec la commande `monobjet.methode(arguments)`. Dans cette écriture, `monobjet` est alors un argument implicite de `methode`. On accède à un attribut de la même façon : `monobjet.attribut`.

Pour déclarer un objet, on utilise `maclasse monobjet = new maclasse(arguments)`. Cette construction fait appel au `constructeur` de l'objet qui initialise les attributs. Remarquons qu'on a ici plusieurs constructeurs : c'est ce qu'on appelle la *surcharge* de méthodes.

3.5.3 Définition d'accessibilité

Dans la déclaration de la classe, on met l'un des mots clés suivants devant le membre (ou devant la méthode) : `private, package, protected, public`. Voir la reference card pour plus de détails.

Une bonne conception impose d'interdire l'accès aux membres. La lecture se fait alors par appel d'une fonction (publique) `getData()` et la modification se fait (éventuellement) par une fonction (publique) `setData(newVal)`.

3.5.4 Membres et méthodes de classe

On y accède par `maclasse.membre` ou `maclasse.methode(args)`. On peut restreindre l'accès. Ces méthodes et membres sont introduits par `static`.

3.5.5 Héritage et interfaces

En java, on ne peut hériter que d'une classe à la fois (d'où les interfaces). Tous les objets desendent de la classe `Objet`.

En Java, on note l'héritage par

```
class fille extends mere{
...
}
```

3.5.6 Classes abstraites

Une classe est dite abstraite quand certaines de ses méthodes ne sont pas implantées (méthode abstraite). On ne peut créer d'objets de telles classes. Par contre on peut créer des objets de classes filles qui implantent les méthodes abstraites.

On doit déclarer les méthodes abstraites par `abstract`. Le corps de la méthode est vide. Une classe qui contient au moins une méthode abstraite est abstraite.

3.5.7 Interface

Pour pallier le manque d'héritage multiple, on utilise en Java des interfaces. Une interface est une classe ne contenant que des méthodes abstraites. Une classe peut *implanter* plusieurs interfaces.

Par rapport à une classe abstraite :

- on hérite moins de code
- mais on peut implanter plusieurs interfaces.

On déclare une interface par `interface` à la place de `class`. Une classe qui implémente une interface se note

```
class foo implements monInterface{
...
}
```

3.5.8 Écrire une classe

```
class maclasse
{
  // attributs de classe

    static public int FOU =1;
    static public int TOUR =2;
    static public int REINE =3;
    static public int ROI =4;

   //attributs

    protected int type;
    protected int x;
    protected int y;


    // constructeur

    maclasse(int a, int b, int c)
    {
```

```
        type=a;
        x=b;
        y=c;
    }

    classe(int a)
    {
        type= a;
        x=0;
        y=0;
    }


    //methodes


    protected  boolean depCorrect (int dx, int dy)
    {
      ...
    }

    ...

}

}
```

4 Les exceptions

Les exceptions permettent une gestion simple des erreurs. Plutôt que d'utiliser des conventions (par exemple pour telle fonction -1 représente une erreur), une méthode retournera une exception en cas d'erreur. Toute méthode a donc deux types de retour : le type déclaré dans sa signature et le type des exceptions (`Exception`).

Une exception se *lance* avec `throw`. Une méthode qui lève une exception doit le déclarer.

```
public static int divide(int x,int y) throws Divide_by_ZeroException {
  ...
}
```

On peut *récupérer* une exception avec `try/catch`.

```
int x =4;
int y=0;

try {
  divide(x,y);
}
catch (Divide_by_ZeroException e) {
  gestion de l'erreur
```

```
}
```

On peut utiliser plusieurs `catch` si plusieurs exceptions sont possibles. Pour créer de nouvelles exceptions, on les déclare comme des classes filles de la classe `Exception` (qui elle esxiste déjà). Une exception de base est juste un "signal", mais les classes dérivées peuvent avoir des membres contenant les données responsables de l'erreur par exemple.

4.1 Compilation

Il faut respecter les règles suivantes :
- une seule classe par fichier (sauf pour les classes "nested" mais ce n'est pas notre propos)
- le nom du fichier est : nomclasse.java

Le fichier principal doit posséder une méthode

```
public static void main (String[] arg)
{
}
```

qui sera lancée à l'exécution.

effacer les fichiers .class : `rm *.class`
compilation : `javac monfichierprincipal.java`
exécution : `java monfichierprincipal`

5 Exercices

Chacun des cinq exercices résoud le même problème en ajoutant une fonctionnalité à chaque fois. Le problème général est, étant donné des pièces de jeu d'échec, de savoir si leur déplacement est valide. On ne tient compte ni de la taille de l'échiquier, ni des autres pièces présentes. Le premier exemple est complet et correct. Il faut compléter le code des autres exemples.

1. Exemple de base, pour vous donner une base fiable. Vous avez juste à lire, compiler et exécuter le code.
2. On veut maintenant vérifier que le déplacement est correct. Utilisation des attributs de classe.
3. On veut améliorer la gestion des erreurs. Utilisation des exceptions.
4. On utilise les interfaces.
5. On utilise les classes abstraites.

Vous trouverez les exercices en `/import/fchevali/TPjava/`

6 Liens vers des documentations

Général :
`http://java.sun.com/`

L'arborescence de classes :

http ://java.sun.com/j2se/1.4.2/docs/api/index.html

Tutoriels :

`http://java.sun.com/docs/books/tutorial/index.html`
`http://java.sun.com/docs/books/tutorial/essential/index.html`