Premier cours
(23 septembre)

Cours programmation-orientée objet en Java

Licence d'informatique

Hugues Fauconnier

hf@liafa.jussieu.fr

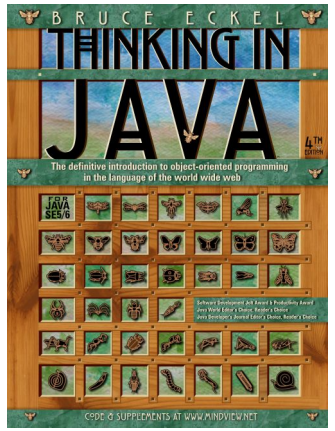
Plan du cours

- Introduction:
  - programmation objet pourquoi? Comment? Un exemple en Java
- Classes et objets (révision)
  - Méthodes et variables, constructeurs, contrôle d'accès, constructeurs
- Héritage: liaison dynamique et typage
  - Extension de classe, méthode et héritage, variables et héritage, constructeurs et héritage
- Héritage: compléments
  - classes abstraites et interfaces, classes internes et emboîtées, classe Object, clonage,
- Introduction à Swing
- Exceptions
  - Exceptions, assertions
- Divers: Noms, conversions, héritage et tableaux
- Généricité
  - Généralités, types génériques imbriqués, types paramètres bornés, méthodes génériques
- Types de données
  - String et expressions régulières, Collections, Conteneurs, itérations
- Entrées-sorties
- Threads
- Compléments
  - Reflections, annotations, documentation…

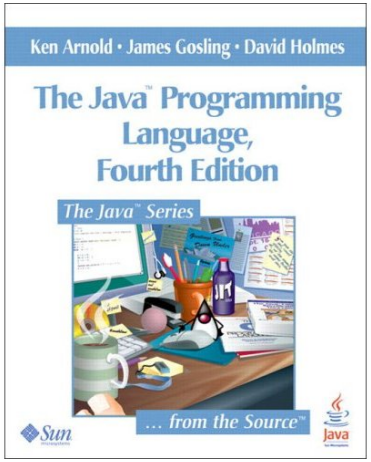
Le site du cours: http://www.liafa.jussieu.fr/~hf/verif/ens/an11-12/poo/L3.POO.html
Didel POO

Bibliographie

- De nombreux livres sur java (attention java >= 1.5)
- En ligne:
  - http://mindview.net/Books/TIJ4
  - Thinking in Java, 4th edition Bruce Eckel
  - http://java.sun.com/docs/index.html



- Livre conseillé:
  - The Java Programming language fourth edition AW Ken Arnold, James Gosling, David Holmes



Chapitre I

Introduction

A) Généralités

- Problème du logiciel:
  - Taille
  - Coût : développement et maintenance
  - Fiabilité
- Solutions :
  - Modularité
  - **Réutiliser le logiciel**
  - Certification

- Comment?

Typage…

- Histoire:
  - Fonctions et procédures (60 Fortran)
  - Typage des données (70) Pascal Algol
  - Modules: données + fonctions regroupées (80) ada
  - Programmation objet: classes, objets et héritage

B) Principes de base de la POO

- Objet et classe:
  - Classe = définitions pour des données (variables) + fonctions (méthodes) agissant sur ces données
  - Objet = élément d'une classe (instance) avec un état
  - (une méthode ou une variable peut être
    - de classe = commune à la classe ou
    - d'instance = dépendant de l'instance

)

Principes de bases (suite)

- Encapsulation et séparation de la spécification et de l'implémentation
  - Séparer l'implémentation de la spécification.
    - Ne doit être visible de l'extérieur que ce qui est nécessaire, les détails d'implémentation sont « cachés »
- Héritage:
  - Une classe peut hériter des propriétés d'une autre classe: un classe peut être une extension d'une autre classe.

Principes de bases de la POO

- Mais surtout notion de *polymorphisme*:
  - Si une classe A est une extension d'une classe B:
    - A doit pouvoir *redéfinir* certaines méthodes (disons f())
    - Un objet a de classe A doit pouvoir être considéré comme un objet de classe B
    - On doit donc accepter :
      - B b;
      - b=a; (a a toutes les propriétés d'un B)
      - b.f()
        - Doit appeler la méthode redéfinie dans A!
      - C'est le *transtypage*
    - (exemple: méthode paint des interfaces graphiques)

Principes de bases

- Polymorphisme:
  - Ici l'association entre le nom 'f()' et le code (code de A ou code de B) a lieu dynamiquement (=à l'exécution)

    Liaison dynamique
  - On peut aussi vouloir « paramétrer » une classe (ou une méthode) par une autre classe.

    Exemple: Pile *d'entiers*

    Dans ce cas aussi un nom peut correspondre à plusieurs codes, mais ici l'association peut avoir lieu de façon statique (au moment de la compilation)

C) Comment assurer la réutilisation du logiciel?

- Type abstrait de données
  - définir le type par ses propriétés (spécification)
- Interface, spécification et implémentation
  - Une interface et une spécification (=les propriétés à assurer) pour définir un type
  - Une (ou plusieurs) implémentation du type abstrait de données
    - Ces implémentations doivent vérifier la spécification

Comment assurer la réutilisation du logiciel?

- Pour l'utilisateur du type abstrait de données
  - Accès uniquement à l'interface (pas d'accès à l'implémentation)
  - Utilisation des propriétés du type abstrait telles que définies dans la spécification.
  - (L'utilisateur est lui-même un type abstrait avec une interface et une spécification)

Comment assurer la réutilisation du logiciel?

- Mais en utilisant un type *abstrait* l'utilisateur n'en connaît pas l'implémentation
  - il sait uniquement que la spécification du type abstrait est supposée être vérifiée par l'implémentation.
- Pour la réalisation *concrète*, une implémentation particulière est choisie
- Il y a naturellement polymorphisme

Notion de contrat (Eiffel)

- Un *client* et un *vendeur*
- Un *contrat* lie le vendeur et le client (*spécification*)
- Le client ne peut utiliser l'objet que par son *interface*
- La réalisation de l'objet est cachée au client
- Le contrat est conditionné par l'utilisation correcte de l'objet (*pré-condition*)
- Sous réserve de la pré-condition le vendeur s'engage à ce que l'objet vérifie sa spécification (*post-condition*)
- Le vendeur peut déléguer: l'objet délégué doit vérifier au moins le contrat (*héritage*)

D) Un exemple…

- Pile abstraite et diverses implémentations

Type abstrait de données

NOM
  pile[X]
FONCTIONS
  vide : pile[X] -> Boolean
  nouvelle : -> pile[X]
  empiler : X x pile[X] -> pile[X]
  dépiler : pile[X] -> X x pile[X]
PRECONDITIONS
  dépiler(s: pile[X]) <=> (not vide(s))
AXIOMES
  forall x in X, s in pile[X]
    vide(nouvelle())
    not vide(empiler(x,s))
    dépiler(empiler(x,s))=(x,s)

Remarques

- Le type est paramétré par un autre type
- Les axiomes correspondent aux pré-conditions
- Il n'y pas de représentation
- Il faudrait vérifier que cette définition caractérise bien un pile au sens usuel du terme (c'est possible)

Pile abstraite en java

```
package pile;

abstract class Pile <T>{
    abstract public T empiler(T v);
    abstract public T dépiler();
    abstract public Boolean estVide();
}
```

Divers

- `package`: regroupement de diverses classes
- `abstract`: signifie qu'il n'y a pas d'implémentation
- `public`: accessible de l'extérieur
- La classe est paramétrée par un type (java 1.5)

Implémentations

- On va implémenter la pile:
  - avec un objet de classe `Vector` (classe définie dans `java.util.package`) en fait il s'agit d'un `ArrayList`
  - Avec un objet de classe `LinkedList`
  - Avec `Integer` pour obtenir une pile de `Integer`

Une implémentation

```
package pile;
import java.util.EmptyStackException;
import java.util.Vector;
public class MaPile<T> extends Pile<T>{
    private Vector<T> items;
    // Vector devrait être remplacé par ArrayList
    public MaPile() {
        items =new Vector<T>(10);
    }
    public Boolean estVide(){
        return items.size()==0;
    }
    public T empiler(T item){
        items.addElement(item);
        return item;
    }
    //...
```

Suite

```
//…
public synchronized T dépiler(){
        int len = items.size();
        T item = null;
        if (len == 0)
            throw new EmptyStackException();
        item = items.elementAt(len - 1);
        items.removeElementAt(len - 1);
        return item;
    }
}
```

Autre implémentation avec listes

```
package pile;
import java.util.LinkedList;
public class SaPile<T> extends Pile<T> {
    private LinkedList<T> items;
    public SaPile(){
        items = new LinkedList<T>();
    }
    public Boolean estVide(){
        return items.isEmpty();
    }
    public T empiler(T item){
        items.addFirst(item);
        return item;
    }
    public T dépiler(){
        return items.removeFirst();
    }
}
```

Une pile de Integer

```
public class PileInteger extends Pile<Integer>{
    private Integer[] items;
    private int top=0;
    private int max=100;
    public PileInteger(){
        items = new Integer[max];
    }
    public Integer empiler(Integer item){
        if (this.estPleine())
            throw new EmptyStackException();
        items[top++] = item;
        return item;
    }
    //…
```

Suite…

```
public synchronized Integer dépiler(){
        Integer item = null;
        if (this.estVide())
            throw new EmptyStackException();
        item = items[--top];
        return item;
    }
    public Boolean estVide(){
        return (top == 0);
    }
    public boolean estPleine(){
        return (top == max -1);
    }
    protected void finalize() throws Throwable {
        items = null; super.finalize();
    }
}
```

Comment utiliser ces classes?

- Le but est de pouvoir écrire du code utilisant la classe Pile abstraite
- Au moment de l'exécution, bien sûr, ce code s'appliquera à un objet concret (qui a une implémentation)
- Mais ce code doit s'appliquer à toute implémentation de Pile

Un main

```
package pile;
public class Main {
    public static void vider(Pile p){
        while(!p.estVide()){
            System.out.println(p.dépiler());
        }
    }
public static void main(String[] args) {
        MaPile<Integer> p1= new MaPile<Integer>();
        for(int i=0;i<10;i++)
            p1.empiler(i);
        vider(p1);
        SaPile<String> p2= new SaPile<String>();
        p2.empiler("un");
        p2.empiler("deux");
        p2.empiler("trois");
        vider(p2);
    }
}
```

E) java: quelques rappels…

- Un source avec le suffixe `.java`
- Une classe par fichier source (en principe) même nom pour la classe et le fichier source (sans le suffixe `.java`)
- Méthode

```
public static void main(String[]);
```

  - `main` est le point d'entrée
- Compilation génère un `.class`
- Exécution en lançant la machine java