TP - Programmation Orientée Objet

Romain PELISSE

ESME Sudria

4 août 2008

Notions de Base

La compréhension des notions suivantes est un *prérequis* à la réalisation de ce tp :

- Classe et Objet
- Constructeurs et Destructeurs
- Visibilité des classes, méthodes et variables (private, public, protected,...)
- Héritage et Polymorphisme Dynamique
- Surcharge de méthodes et d'opérateurs

Si vous avez des doutes ou des questions sur ces notions, c'est le moment !

Différences conceptuelles

C++

- Simple *surcouche objet* au C et non un "pur" langage Objet,
- Puissant mais **complexe**, utilise **beaucoup de mot-clés**,
- La **libération de la mémoire** est à la charge du développeur,
- Le langage est peu structurant dans l'organisation des fichiers.

Java

- Langage *très orienté* objet,
- Syntaxe **simple** et structurante, **peu de mots-clés**,
- Gestion automatique de la **libération mémoire**,
- Une classe = un fichier (imposée par le compilateur).

La "super" classe 'Object'

Tout objet hérite de manière implicite de 'Object' :

- 'String toString()' retourne l'objet sous forme de chaîne.
- 'boolean equals(Object other)' permet de comparer 2 objets.
- 'Class getClass()' qui permet de récupérer une instance de la classe de l'objet (voir plus loin la partie sur la réflexion).

Héritage implicite

```
package org.esme.samples;
// This class implicitly inherits from 'Object'

public class ImplicitInheritance {
                    //'extends Object' is implicit
}
```

Absence d'héritage multiple

Pourquoi il n'y a pas d'héritage multiple ?

- **Simplicité**, le code est moins complexe, la hiérarchie des classes est plus claire
- L'héritage multiple est reconnu comme une "**bad practice**"
- Les **Interfaces** sont utilisées à la place

Gestion automatique de la mémoire

Le *bytecode*

- Compilation génère du *bytecode*, soit du code binaire.
- Le *bytecode* est interprétable par la machine Java et non par le processeur physique (PowerPPC, Intel, Vax,...)
- Java se charge de l'interprétation du *bytecode* et le traduit pour le processeur physique, ce qui assure **la portabilité** des programmes.

En quoi cela change la gestion de la mémoire ?

Gestion automatique de la mémoire

Plus de delete !

Si l'objet n'est plus *référencé*, la mémoire est libérée.

Attention...

- Ne supprime pas le risque de fuite mémoire
- Réflechir à l'allocation de tout objet

Surcharge d'opérateur en Java

Java interdit la surcharge d'opérateur :

```
package org.esme.samples;

public class MilkBottle {
    float quantity;    // represents the amount
            // of milk in the bottle.

    public void add(MilkBottle otherBottle) {
        this.quantity += otherBottle.quantity;
    }
}
```

Les nouveaux modificateurs

- `final, static`, détaillé plus loin.
- `transient, volatile`, qualifie la persistance de l'objet, son usage est généralement déconseillé. Dans le cadre du TP, ce mot clé est interdit.
- `strictfp, native`, utilisé pour faire des ponts vers le langage C. Dans le cadre du TP, ce mot clé est interdit.
- `synchronised`, utilisé pour gérer les sections de **code critique** en programmation parallèle. Dans le cadre du TP, ce mot clé est interdit.

Final

- Une classe `final` ne peut avoir de classe fille (interdit l'héritage).
  - Assure le respect d'une certaine conception
  - Améliore les performances dans certaines conditions (pour les gurus)
- Une variable `final` ne peut être affectée qu'une seule fois.
- Une méthode `final` ne peut être surchargée (comme si la classe était `final`)

Par exemple, la classe String en Java est final, on ne peut en hériter.

Quel interêt voyez vous à l'utilisation de ce modificateur ?

Static

- Une classe ne peut pas être `static`.
- Une variable ou une propriété `static` n'est instanciée qu'une seule fois en mémoire
- Une méthode `static` n'est pas dépendante d'une instance de la classe et peut donc être appelée ou que ce soit dans le code : c'est une *fonction* !

Quel est l'interêt d'avoir une propriété `static` ?

Static et Final

- Une variable `final` et `static` n'est instanciée qu'une seule fois en mémoire et ne peut être initialisée qu'une seule fois : c'est une *constante* !

```
package org.esme.samples;

/**
 * This class represents an Asynchronous Engine,
 */
public class AsynchronousEngine {
    // Represents how many coils has the engine
    private final static int NB_COILS = 10;
    // Represents the maximum output current
    public final static int MAX_CURRENT = 100;
}
```

Variables primitives

- En Java, tout est objet !
- ... sauf les types primitifs (integer, char, float...),
- Mais on peut les transformer en Objets

`int`	Integer
`float`	Float
`double`	Double
`char`	Character

Autoboxing

Java 5 permet d'effectuer ce genre d'opérations automatiquement

Principale nouveauté du langage Java par rapport au C++

- Permet de remplacer l'**héritage multiple**
- Assure un **couplage lâche** (*loose coupling*)
- **Evite de nombreux transtypage** (*cast*) inutiles.

C++

```
public class Checksumable
{
 public:
   virtual int checksum();
}
```

Java

```
package org.esme.samples;

interface Checksumable {
  public int checksum();
}
```

Pourquoi une interface remplace l'héritage multiple ?

Supposons que nous voulions que deux objets de types bien différents (une classe représentant un Fichier et une autre représentant un Utilisateur, par exemple), disposent TOUS les deux d'une méthode checksum().

Comment faire ?

Erreur de conception...

En C++, il aurait suffit de faire un héritage multiple, mais cela aurait entraîné une arborescence d'objets peu cohérente. En effet, en quoi un Utilisateur devrait-il hériter d'un Fichier ?

Solutions à l'aide d'interface

```
package org.esme.samples;
import java.util.ArrayList;
import java.util.List;

public class InterfaceUsage {
  public static void main(String[] args) {
    List<Checksumable> items;
    items = new ArrayList<Checksumable>(2);
    items.add(new MyFile());
// MyFile extends File and implements Checksumable
    items.add(new User());
// User also implements Checksumable
    for ( Checksumable item : items ) {
      System.out.println(item.checksum());
    }
  }
```

Limites par rapport à l'héritage multiple

On n'hérite pas de l'implémentation !

- Il faut coder 'checksum()' pour chaque classe !
- Cependant, en pratique, ce n'est pas un problème comme dans ce cas, car les implémentations requises pour une même méthode sur deux objets distincts sont généralement très différentes...

Pour s'en rendre compte, il suffit de réflechir à l'implémentation de checksum() pour un fichier et pour un objet Utilisateur...

Couplage lâche

```
package org.esme.samples;

public class LooseCoupling {
// This class can use ANY object implementing
// the Checksumable interface...
    private Checksumable field;

    public LooseCoupling(Checksumable field) {
        this.field = field;
    }
}
```

En quoi cette solution est-elle *meilleure* ?

Exemple d'utilisation

```
package org.esme.samples;

public class LooseCouplingExample {
    public void usingMyFileOrUser(){
        LooseCoupling obj;
        obj = new LooseCoupling(new MyFile());
        LooseCoupling other;
        other = new LooseCoupling(new User());
    }
}
```

Pourquoi une notion de **package** ?

- Pour assurer la séparation du code, sous forme de modules :
  - `org.esme.tppoo.ihm`
  - `org.esme.tppoo.process`
  - `org.esme.tppoo.persistance`
- Pour assurer une unicité de type

Convention de nommage

- URL "inversée" (`org.esme.tppoo` par exemple)
- toujours en **minuscules**,**pas de chiffres**

Convention pour les TPs

`org.esme.tppoo`, `org.esme.tpihm`, `org.esme.tpxml`,...

Les Collections en Java

- Tableaux d'objets, **alloués dynamiquement**
- Listes chainées "génériques" :
  - Simple (`Set`)
  - Ordonnées (`List`)
  - Indexées (`Map`)
- Elles peuvent aussi être triées (`SortedSet`,`SortedMap`)
- Depuis Java 5, elles sont **typées** (`List<String>`,...)

Utilisez les !

- Plus simples d'utilisation
- Optimisées à chaque version de la machine virtuelle Java

Utilisation d'une Map

```
package org.esme.samples;

import java.util.HashMap;
import java.util.Map;

public class MapUsage {
    public void usingMap() {
        Map<String,Object> map;
        map = new HashMap<String,Object>(3);
        map.put("key","value");
        System.out.println(map.get("value"));
    }
}
```

Gestion des exceptions

Que faisiez vous en C pour gérer une erreur ?

- Rien, le programme plante.
- Renvoyer pointeur NULL ou un code d'erreur (STATUS)
- Retourner une structure décrivant l'erreur : plus d'autre retour possible !

Java, comme C# et même le C++, dispose d'un mécanisme d'**Exception** !

Mécanisme

Quand on souhaite signaler une erreur à l'appelant, il suffit de "jeter" **(throws)** une exception décrivant l'erreur. Charge à l'appelant de "l'attraper" (catch), et d'agir en conséquences, ou de la "jeter" à son propre appelant...

A votre avis, quels sont les avantages d'un tel mécanisme ?

Avantages

- La signature de la méthode est plus claire (les cas d'erreurs seront explicitement décrits)
- L'Exception contient toutes les informations pour analyser le problème.
- L'erreur est remontée à la couche appropriée

Note : Toute exception hérite de la classe Exception ou Error, ou sinon elle implémente l'interface Throwable.

```java
package org.esme.samples;

public class Developer {
  /**
   * Call this when a developer is working...
   * @throws ProjectException, issues related to
   *  management of the project is not the
   *  responsability of the developer. */
  public void doingAProject() throws ProjectException {
   try {
     this.code();
   }
   catch ( TechnicalException e ) { //
     // Taking care of technical isssues is
     // of the developer responsability.
     this.doingSomethingAppropriate(e);
   }
  }
```

Utilisation des exceptions

- Les exceptions sont souvent **peu ou mal utilisées** malgré leur rôle crucial.
- Java vient avec **5 exceptions déjà définies** pour les cas les plus courants :
  1. **IllegalArgumentException**
  2. **IllegalStateException**
  3. **IndexOutOfBoundsException**
  4. **ConcurrentModificationException**
  5. **UnsupportedOperationException**

  Si aucune d'elle ne semble convenir à la situation, à vous de créer une exception adaptée au problème.
- La conception des exceptions fait autant parti de la conception d'un programme en langage orienté objet que l'héritage !

Fonctionnalités Avancées

- Chargement dynamique de classe : `Classloading`
- API de `Reflexion`

Première partie

1. Réalisation de l'arborescence du projet (src, docs, bin,...) : "A l'aide d'Eclipse, créer un nouveau projet Java (New Project - >Java Project). Quels répertoirs sont créés ? A quoi servent-ils ?".
2. Package : "Créer un nouveau paquetage dans le répertoire 'src' nommé 'org.esme.tppoo'."
3. Réalisation d'une classe d'exemple : "Créer une classe dans le paquetage org.esme.tppoo, nommée *Version*. Cette classe disposera d'une méthode *getVersion* qui renverra la chaîne **"1.0"**.

Deuxième partie

1. Réalisation d'un programme principal : "A l'aide d'Eclipse (regardez les options du formulaire de création de classe), créer un 'programme principal', où vous créerez une instance de votre objet Version et où vous utiliserez la méthode getVersion."
2. Utilisation du **Debugger**
   - Fixer un point d'arrêt
   - Examiner la valeur d'une variable
   - Changer la valeur d'une variable
3. Compilation et utilisation en ligne de commande (en // à Eclipse) et avec Ant