Introduction à la Programmation Objet:

Du bon usage des références Java

Rachid Guerraoui[1] & Jamila Sam[2]

[1]Laboratoire de Programmation Distribuée (LPD)
[2]Laboratoire d'Intelligence Artificielle (LIA)

Faculté Informatique & Communications
Ecole Polytechnique Fédérale de Lausanne

Objectif

La manipulation d'objets **via des indirections** (comme les références Java) nécessite quelques précautions.

Le but de ce cours d'approfondissement est de vous y rendre attentifs.

Nous aborderons pour cela les thèmes suivants :

- "Failles d'encapsulation" (privacy leaks)
- copie de surface et copie profonde
- portée et durée de vie des objets

Nous conclurons par quelques mises en gardes quant à la transition vers d'autres langages

Bonne encapsulation

Principe fondamental : **la modification d'une instance ne peut se faire que via son interface** (ses méthodes publiques)

☞ But : garantir que chaque objet puisse contrôler l'intégrité des données qui le caractérisent et puisse en sécuriser l'accès

Un programme présente des "failles d'encapsulation" (privacy leaks) lorsque le principe fondamental ci-dessus n'est pas respecté

Attention : déclarer soigneusement tous les attributs d'une classe comme `private` n'est pas toujours suffisant pour garantir l'absence de telles failles !

Voyons cela sur un exemple ...

Failles d'encapsulation : Exemple (1)

Un programmeur code une classe pour représenter des horaires :

```
class Horaire {
   private String jour;
   private double heure;
   public Horaire(String unJour, double uneHeure) {
         // controle que le jour et l'heure
         //sont ok de façon générale, et si oui:
         jour = unJour;
         heure = uneHeure;
   }
   public void setHeure(double uneHeure) {
         // controle que l'heure est entre 0 et 24, et si oui :
         heure = uneHeure;
   }
   public String  toString() { ...}
}
```

Une instance de `Horaire` peut aussi bien représenter l'horaire d'une séance de cinéma (*"Dimanche à 23h"*) que celui d'un cours (*"Jeudi à 8h15"*).

Failles d'encapsulation : Exemple (2)

Un autre programmeur trouve la classe `Horaire` pratique pour modéliser des examens :

```
class Examen {
    private String nom;
    private Horaire horaire;
    public Examen (String unNom){
            nom = unNom;
    }
    public void setHoraire(String jour, double heure){
       // contrôle que jour et heure sont plausibles comme
       // horaire d'examen (pas un dimanche à 23h par exemple !)
       //  si ok:
         horaire = new Horaire(jour, heure);
    }
    public Horaire getHoraire(){
       return horaire;
    }
}
```


Failles d'encapsulation : Exemple (3)

Le programmeur de la classe `Examen` souhaite garantir une bonne encapsulation et une manipulation correcte des données :

- il déclare en `private` tous les attributs ;
- il veille à ce que la méthode `setHoraire` contrôle que l'on assigne un horaire plausible à l'examen

Failles d'encapsulation : Exemple (4)

Il est donc impossible à un utilisateur de la classe `Examen` d'écrire :

```
Examen serie = new Examen("Série notée");
serie.horaire = new Horaire("Dimanche", 23.0); // impossible car
                                               // serie.horaire
                                               // est privé
serie.setHoraire("Dimanche", 23.0);  // valeurs refusées
                                     // par la méthode setHoraire
```

Ce qui est très bien ! Mais, notre utilisateur peut aussi écrire ceci :

```
Examen serie = new Examen("Série notée");
serie.setHoraire("Jeudi", 10.0);// valeurs ok pour un examen
Horaire hor;
hor  = serie.getHoraire(); // SOURCE DU PROBLEME ICI
// ...
//  ah il y a une séance de cinema les jeudis à 23h
hor.setHeure(23); // OK pour un horaire général
// ...
//  j'affiche l'horaire de mon test
System.out.println(serie.getHoraire());  // jeudi à 23h (Argh !!)
```


Failles d'encapsulation : Exemple (5)

Explications :

1. la méthode publique `getHoraire` de la classe `Examen` retourne la référence à l'instance de `Horaire` qui représente l'horaire de l'examen
2. l'affectation `hor = serie.getHoraire();` fait donc en sorte que la variable locale `hor` et l'attribut `horaire` de `serie` **référencent le même objet en mémoire**
   - ☞ Toute modification de l'objet référencé par la variable locale `hor` correspond à une modification de l'objet référencé par l'attribut `horaire`
3. la classe `Horaire` contient une méthode publique `setHeure` permettant de modifier ses instances

☞ l'attribut `horaire` de la classe `Examen` peut être modifié sans passer par l'interface de `Examen` !

Failles d'encapsulation : Exemple (6)

Solution : faire en sorte que `getHoraire` retourne une référence à une **copie** de l'horaire de l'examen.

```
class Examen {
    ....
    public Horaire getHoraire() {
       return new Horaire(horaire); // prévoir un constructeur
                                    // de copie dans la classe
                                    // Horaire
    }
}
```

Classes `Mutable` et `Immutable`

- Une classe est dite `Mutable` si elle contient des méthodes publiques permettant de modifier ses instances (modifier les valeurs des attributs)
  Les instances d'une classe `Mutable` sont aussi qualifiés de `Mutable`
- Dans l'exemple précédent, on n'aurait pas pu modifier l'objet référencé par `serie.horaire` via la variable locale `hor` si la classe `Horaire` n'avait pas contenu la méthode `setHeure`
  - ☞ La classe `Horaire` est `Mutable` en raison de la présence de la méthode `setHeure`
- Par opposition, une classe est dite `Immutable` s'il ne contient aucune méthode (à part les constructeurs) permettant de modifier l'objet
  - ☞ C'est le cas de `String`

Un getter (et la plupart des méthodes) ne doivent jamais retourner une référence à un objet `Mutable` et `private`

Classes `Mutable` et `Immutable` (2)

Exemple :

```
class Examen {
    ....
    public String getNom() {
       return nom;          // COPIE INUTILE (String est Immutable)
    }
    public Horaire getHoraire() {
       return new Horaire(horaire); // COPIE NECESSAIRE
                                    // (Horaire est Mutable)
    }
}
```

Note : vous verrez plus de détails sur la programmation de classes immutables au semestre de printemps.

Failles d'encapsulation : ça n'est pas fini !

Retourner une référence à un objet mutable et privé n'est pas la seule façon de porter atteinte au principe d'encapsulation

Exercice : trouvez la faille dans cette façon de définir ce second *setter* pour la classe `Examen` de l'exemple précédent :

```
class Examen {
    ....
   public void setHoraire(Horaire unHoraire) {
      // controle que unHoraire est ok pour un exa;
      // et si oui:
        horaire = unHoraire;
    }
}
```

☞ Faire très attention à ce que l'on fait lorsque l'on utilise l'opérateur `=` en Java !

Copie d'objets

Nous avons vu précédemment qu'il était fondamental qu'un *getter* retourne toujours une **copie** de l'attribut concerné si celui-ci est un objet mutable et privé.

Nous avons utilisé la notion de construction de copie pour le garantir dans l'exemple précédent

Nous allons voir maintenant que la façon de réaliser la copie est aussi très importante pour prévenir les failles d'encapsulation ...

Copie d'objets : exemple (1)

Reprenons notre classe `Examen` et dotons la d'un constructeur de copie :

```
class Examen
{
   private String nom;
   private Horaire horaire;

   public Examen (String unNom) {
         nom = unNom;
   }
   public Examen(Examen autreExa) { // Constructeur de copie
         nom = autreExa.nom;
         horaire = autreExa.horaire;
   }
     ...
}
```

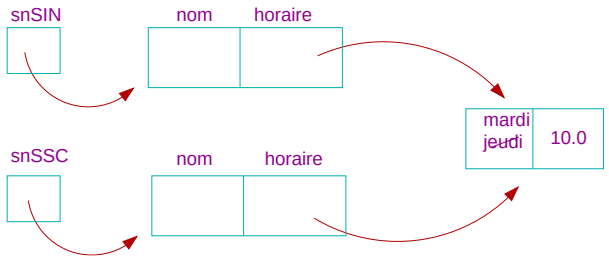
☞ Copie de la valeur de chaque attribut de `autreExa` dans l'attribut de même nom de `this`

Copie d'objets : exemple (2)

Examinons maintenant l'utilisation du constructeur de copie :

```
Examen snSIN = new Examen("Série notée");
snSIN.setHoraire("Jeudi", 10.0);
Examen snSSC = new Examen(snSIN); // Copie
```

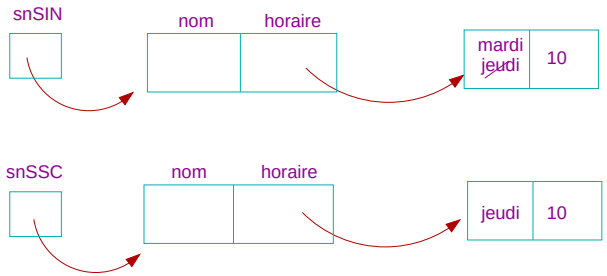
Attention ! en copiant l'attribut `horaire` de `snSIN` dans l'attribut `horaire` de `snSSC`, on a copié une référence :



☞ L'instance `snSSC` peut être modifiée sans passer par les méthodes publiques de son interface (via l'instance `snSIN`) !

Copie de surface / Copie profonde

- Une méthode de copie d'objets qui se contente de faire une copie "champ à champ" des valeurs des attributs (y compris lorsque la valeur est une référence) fait une **copie de surface**
- Afin d'éviter toute faille d'encapsulation, la copie d'un objet doit produire une autre objet complètement indépendant :



☞ **copie profonde**

Copie profonde

Copie profonde : il faut créer une copie indépendante pour tout attribut dont la classe est mutable

```
class Examen
{
   ....
   public Examen(Examen autreExa) {
         nom = autreExa.nom; // inutile de créer une copie ici :
                             // nom est de type String qui
                             // est immutable
         horaire = new Horaire(autreExa.horaire); //Copie profonde
   }
   ....
}
```

Copie profonde et tableaux (1)

Exemple de ce qu'il faut faire :

```
class Etudiant {
   //tableau d'objets mutables!
   private Examen[] examens;

   // méthode utilitaire créant une copie indépendante
   // d'un tableau d'examens
   private Examen[] copieExamens(Examen[] autreExamen)
      {
         int size = autreExamen.length;
         Examen[] temp = new Examen[size];
         for (int i=0; i < size; i++){
            temp[i] = new Examen(autreExamen[i]);
         }
         return temp;
      }

   // suite -->
```

Copie profonde et tableaux (2)

Exemple de ce qu'il faut faire (suite) :

```
class Etudiant {
   .....
   public Etudiant(Etudiant autreEtudiant) //constructeur de copie
      {
         examens = copieExamens(autreEtudiant.examens);
      }
   public Examen[] getExamens() //getter "sécurisé"
      {
         return copieExamens(examens);
      }
};
```

☞ La méthode utilitaire pour la copie profonde d'un tableau d'examens utilise le constructeur de copie de la classe `Examen` qui **doit aussi faire de la copie profonde** !

Constructeur de copie et polymorphisme (1)

Dans certaines situations le recours au constructeur de copie n'est pas la bonne solution pour produire une copie d'objets.

☞ Les constructeurs, quels qu'ils soient, **ne peuvent agir de façon polymorphique** !

Reprenons l'exemple précédent pour examiner ce point ...

Constructeur de copie et polymorphisme (2)

Supposons que notre classe `Examen` soit dotée d'une sous-classe `Oral` :

```
class Oral extends Examen {
   private String expert;
   public Oral(String unNom, String unExpert) { // constructeur
         super(unNom);
         expert = unExpert;
      }
   public Oral(Oral autreOral) { // constructeur de copie
         super(autreOral);
         expert = autreOral.expert;
      }
}
```

Le tableau de la classe `Etudiants` :

```
class Etudiant {
   private Examen[] examens; .... }
```

peut maintenant contenir aussi bien des instances de `Examen` que des instances de `Oral` !

Constructeur de copie et polymorphisme (3)

Que fait la méthode `copieExamens` dans ce cas :

```
class Etudiant {
   ....
   private Examen[] copieExamens(Examen[] autreExamen)
      {
         int size = autreExamen.length;
         Examen[] temp = new Examen[size];
         for (int i=0; i < size; i++){
            temp[i] = new Examen(autreExamen[i]);
         }
         return temp;
      }
      ....
```

☞ Comme les constructeurs ne sont pas polymorphiques, si `autreExamen[i]` est un `Oral`, il sera copié comme un `Examen` : plus d'attribut `expert` !

Clonage d'objet

Il est recommandé en Java d'avoir recours à la méthode :

```
Object clone()
```

héritée de la classe `Object` pour produire des copies d'objets

- `clone` remplit le même rôle qu'un constructeur de copie, à la différence près qu'**elle peut agir de façon polymorphique**
- `clone` doit être redéfinie dans chaque classe nécessitant la copie

Codage de la méthode `clone`

Une façon simple de re-définir la méthode `clone` ... est d'utiliser les constructeurs de copie.
Pour notre exemple :

```
class Examen {
   ....
   public Examen clone() {
         return new Examen(this); // crée une copie
                                  // de l'instance courante
   } ....}
class Oral extends Examen {
   ....
   public Oral clone() {
         return new Oral(this);
      } ....}
```

Note : Le fait d'avoir pu mettre `Examen` et `Oral` comme type de retour à la place de `Object` est une facilité introduite depuis la version 5.0 de Java (≪types de retour covariant≫).

Clonage d'objet : exemple de copie profonde

Le code de notre méthode de copie d'examen devient alors :

```
class Etudiant {
   ....
   private Examen[] copieExamens(Examen[] autreExamen)
      {
         int size = autreExamen.length;
         Examen[] temp = new Examen[size];
         for (int i=0; i < size; i++){
            temp[i] = autreExamen[i].clone(); // copie profonde
         }
         return temp;
      }
   ....
}
```

☞ La copie va maintenant **s'adapter au type de l'objet** contenu dans chaque entrée du tableau (polymorphisme)

Codage recommandé de la méthode `clone`

Le codage de la méthode `clone` au moyen des constructeurs de copie va fonctionner correctement dans la plupart des situations (mais pas toutes !)

Pour cette raison, le procédé "officiel" pour coder `clone` comprend :

- l'invocation de la méthode `clone` des super-classes ;
- l'utilisation de l'interface `Cloneable`,
- la gestion des exceptions

☞ semestre de printemps !

Le but ici était principalement de vous donner une introduction.

Durée de vie des objets

En Java un objet **existe tant qu'il est utilisé** : la durée de vie peut donc dépasser la portée :

```
class A {...}
class B { A monA; }
....
B unB = new B();
{ // un bloc
   A unA = new A(...);
   unB.monA = unA; // unB.monA pointe sur unA
                   // qui est locale au bloc
} // fin du bloc
```

après le bloc :

- la variable `unA` n'est plus accessible ici (fin de la portée)
- l'objet associé au départ à `unA` existe toujours car il est utilisé par l'attribut `monA` de `unB`

Le *garbage collector* est en charge de récupérer la mémoire associée aux objets qui ne sont plus référencés du tout.

Référence Java et autres langages (1)

1. La façon particulière qu'a Java de gérer systématiquement les objets via des indirections (références) ne se retrouve pas dans tous les langages.
2. La mise en oeuvre ou signification de concepts fondamentaux tels que :
   - l'affectation d'objets
   - la comparaison d'objets
   - leur durée de vie
   - ...

   peut être différente en dehors de Java

☞ **Rappelez-vous en lorsque vous transiterez vers d'autre langages !**

Deux exemples pour vous donner une idée des implications que cela peut avoir ...

Référence Java et autres langages (2)

Exemple de l'affectation

Soit `Obj` une classe dotée d'un méthode `modifie` modifiant l'instance courante.

```
Obj a;
Obj b;
 ..
a = b;  // En Java: copie de la référence
        // En C++: copie ''champ à champ'' de b dans a

b.modifie(); // En Java : modifie aussi a
             // En C++ : b est une copie distincte de a
             //          a n'est pas modifié
```

Notez que le **code** ici est **identique dans** les deux langages !

Référence Java et autres langages (3)

Exemple de la durée de vie

- en Java : **la durée de vie peut dépasser la portée**

```
class A {...}
class B {
   A monA; // indirection (référence) vers un objet A
}
....
B unB ...; // création d'un objet B
{ // bloc
   A unA ...; // création d'un objet A
   unB.monA = unA; // stockage de la référence (adresse)
                   // de unA dans unB.monA
} // fin du bloc
```

après le bloc :

- `unA` n'est plus accessible (fin de portée)
- l'objet qui était associé à `unA` continue d'exister car référencé par `unB.monA`

Référence Java et autres langages (4)

- en C++ : la **durée de vie** est strictement **égale à la portée**

```
class A {...}
class B {
   A* monA; // indirection (pointeur) vers un objet A
}
....
B unB ...; // création d'un objet B

{ // bloc
   A unA ...; // création d'un objet A
   unB.monA = &unA; // stockage de l'adresse de unA dans
                    // unB.monA
} // fin du bloc
```

après le bloc :

- `unA` n'est plus accessible (fin de portée)
- l'objet qui était associé à `unA` est détruit
- `unB.monA` pointe vers un objet invalide

Ce que j'ai appris aujourd'hui

- qu'assurer une bonne encapsulation, garante de la robustesse du code face au changement, nécessite une certaine discipline !
- que la copie de surface (par opposition à la copie profonde) peut-être dangereuse
- que la façon de gérer la durée de vie des objets peut-être différente d'un langage à l'autre
- qu'il vaut la peine de s'intéresser à la sémantique et mise en oeuvre des concepts fondamentaux lorsque l'on passe à d'autres langages