Séance 2

Partie 1 : Instances & Encapsulation

Dans cette première partie, nous allons approfondir la notion d'instances d'une classe, en considérant la copie d'une instance, le passage d'instances en paramètre, les méthodes et les attributs de classe. Nous aborderons également l'encapsulation, qui permet de protéger l'information contenue dans un objet.

Sommaire

Le code de base est le suivant :

```
class Etudiant {

    String nom;

    Etudiant(String s) {
        this.nom = s;
    }
}

class Binome {
    Etudiant etu1;
    Etudiant etu2;

    public Binome(Etudiant etu1, Etudiant etu2) {
        this.etu1 = etu1;
        this.etu2 = etu2;
    }

}

class Projet {
    Binome b;
    String titre;
    float note;

    public Projet(Binome b, String titre, float note) {
        this.b = b;
        this.titre = titre;
        this.note = note;
    }
}

class Application {

    /**
     * @param args the command line arguments
     */
    public static void main(String[] args) {
        Etudiant a = new Etudiant("Saurion");
        Etudiant b = new Etudiant("Bilbo");
        Binome bi = new Binome(a,b);
        Projet p = new Projet(bi,"Le projet qui dechire",0.05f);
    }

}
```

L'archive `instances.zip` contient le code de ces classes et peut être importée directement dans `NetBeans/eclipse`.

2.1 Instances d'objets

Avant d'aborder des notions nouvelles, rappelons que les instances sont créées en faisant appel à un constructeur à l'aide de l'instruction `new`.

Ainsi en Java, le code suivant ne crée qu'une seule instance au final :

```
Coucou c1; // c1 est une reference nulle sur un objet Coucou
c1 = new Coucou(); // appel explicite du constructeur par defaut
Coucou c2 = c1; // copie de la reference - *un* *seul* objet
Coucou c3;
c3 = c1 ; // copie de la reference - *toujours* un seul objet
```

Remarque : ceci n'est pas forcément valable pour d'autres langages comme C++ par exemple.

2.1.1 Attributs et méthodes statiques

Définition 2.1 (attribut statique (ou attribut de classe)). *Un attribut statique est un attribut partagé par l'ensemble des instances d'une classe. Il ne requiert donc pas d'instance d'une classe : pour cette raison, on peut accéder à un attribut statique avec la syntaxe :*

```
Classe.nomAttribut
```

On déclare un attribut comme statique en le précédent avec le mot clef `static` :

```
static type nomAttribut;
```

Par exemple, `System.in`, `System.out` et `System.err` sont trois attributs statiques de la classe `System` qui désignent respectivement les flux "standard" en entrée, en sortie "normale" et en sortie pour les messages d'erreurs.

Exercice 21. *(⋆) On suppose que dans notre application, tous nos étudiants seront des étudiants de l'IUT de Bordeaux. On souhaite stocker cette information dans un attribut `IUT` de valeur `"IUT de Bordeaux"`.*

Modifiez la classe `Etudiant` en fonction, créez deux instances `Bob` et `Greg`, puis faites afficher dans le terminal la valeur de l'attribut `IUT` pour chacune de ces instances.

Quelle est la place mémoire occupée par l'ensemble de ces deux instances ?

Définition 2.2 (méthode statique (ou méthode de classe)). *Une méthode statique est une méthode qui n'utilise que des attributs statiques de la classe et ne fait appel qu'à d'autres méthodes statiques. Une méthode est déclarée comme statique à l'aide du mot-clef `static` :*

```
static typeRetour nomMethode( typeArg1 arg1, typeArg2 arg2 ...)
```

Une méthode statique ne requiert donc pas une instance d'une classe pour être exécutée : pour cette raison, on peut appeler une méthode statique avec la syntaxe :

```
Classe.nomMethode(args);
```

Par exemple, la méthode `currentTimeMillis()` est une méthode statique de la classe `System` qui retourne la date actuelle en nombre de millisecondes depuis le 1er Janvier 1970. Cette donnée est intrinsèquement une donnée qui ne peut dépendre d'une instance. Elle est donc enregistrée dans un attribut statique et la méthode qui retourne sa valeur peut donc être statique.

Exemple d'utilisation :

```
long nbMilliSecondes = System.currentTimeMillis();
System.out.println("Nb de millisecondes = "+ nbMilliSecondes);
```

La méthode `main` (en Java) est un exemple incontournable de méthode statique. Elle est le point de départ d'une application `Java`.

Exercice 22. *(⋆) Expliquer pourquoi la classe ci-après n'est pas correcte.*

```
class MainAppli {

    Etudiant bob;

    MainAppli(){
        System.out.println("Bienvenue");
    }

   public static void main(String[] args) {
        new MainAppli();
        bob = new Etudiant("Bob");
    }
}
```

Exercice 23. *(⋆⋆) Proposez un correctif pour l'exercice 22.*

Exercice 24. *(⋆) Dans la classe `Etudiant`, ajouter une méthode statique `afficherIUT()` qui affiche la valeur de `IUT` (cf exercice 21) dans le terminal, et donnez deux exemples de syntaxe différente d'appel de cette méthode.*

Exercice 25. *(⋆⋆⋆) On souhaite comptabiliser le nombre d'instances d'étudiants dans un attribut statique `nbTotalEtudiants` de la classe `Etudiant`. Comment faire pour actualiser celle-ci correctement ?*

2.1.2 Passage d'objets en paramètres d'une méthode

Les paramètres sont passés par valeur :

— dans le cas d'un type de base, une copie de la donnée est mise à disposition de la méthode - il s'agit d'un passage en entrée ;

— dans le cas d'un objet, une copie de la référence originale est effectuée : la méthode accède donc directement à l'objet et peut modifier la valeur de ses variables/attributs - il s'agit donc d'un passage en entrée-sortie.

Par exemple, le code :

```
static void passageInt(int i){
        System.out.println("Debut methode: i vaut "+i);
        i=12345;
        System.out.println("Fin methode: i vaut "+i);
    }

public static void main(String[] args) {
        int i = -27;
        System.out.println("Avant methode: i vaut "+i);
        passageInt(i);
        System.out.println("Apres methode: i vaut "+i);
    }
```

affiche :

```
Avant methode: i vaut -27
Debut methode: i vaut -27
Fin methode: i vaut 12345
Apres methode: i vaut -27
```

En effet, la variable `i` dans la méthode `passageInt` est une copie de celle de l'appel (car `int` fait partie des types primitifs) : après l'appel à la méthode `passageInt`, la variable `i` de `MainAppli` a conservé sa valeur : -27.

Autre exemple, le code :

```
static void passageEtudiant(Etudiant e){
        System.out.println("Debut methode: le nom de l'etudiant e vaut "+e.nom);
        e.nom="Tetouillou le terrible dragon";
        System.out.println("Fin methode: le nom de l'etudiant e vaut "+e.nom);
    }

public static void main(String[] args){
        Etudiant e = new Etudiant("Dark Vador le gentil");
        System.out.println("Avant methode: le nom de l'etudiant e vaut "+e.nom);
        passageEtudiant(e);
        System.out.println("Apres methode: le nom de l'etudiant e vaut "+e.nom);
    }
```

affiche :

```
Avant methode: le nom de l'etudiant e vaut Dark Vador le gentil
Debut methode: le nom de l'etudiant e vaut Dark Vador le gentil
Fin methode: le nom de l'etudiant e vaut Tetouillou le terrible dragon
Apres methode: le nom de l'etudiant e vaut Tetouillou le terrible dragon
```

En effet, la méthode `passageEtudiant` a modifié l'attribut `nom` de l'instance `e` passée en paramètre.

Exercice 26. *(⋆) Testez le fonctionnement des deux exemples précédents, en plaçant des points d'arrêt judicieusement pour observer les changements de valeur des variables.*

2.1.3 Copie d'une instance

Une solution pour que la méthode `passageEtudiant` ne modifie pas l'instance passée en paramètre est de lui passer une copie de l'instance, et non l'instance elle-même.

En Java, tout objet possède une méthode `clone()` qui réalise une copie d'une instance, en dupliquant les attributs de type primitifs.

On pourrait donc écrire [1] :

```
Etudiant e = new Etudiant("Dark Vador le gentil");
Etudiant copie = (Etudiant) e.clone();
passageEtudiant(copie);
```

Cependant cette méthode `clone()` possède certaines limites dans des cas d'utilisation impliquant des attributs de type non-primitifs.

Nous utiliserons une autre solution, exploitée dans d'autres langages objets comme C++ par exemple, qui consiste à implémenter un constructeur par copie, i.e. un constructeur qui prend un objet de même type en paramètre et qui recopie la valeur de chacun des attributs du paramètre dans les siens :

Voici une telle implémentation pour notre classe `Etudiant` :

```
public class Etudiant  {
    String nom;
    static String IUT = "IUT de Bordeaux";

    Etudiant(String nom){
        this.nom=nom;
    }

    Etudiant( Etudiant e){
        nom=e.nom;

    }

}
```

On peut maintenant faire l'appel en passant une copie :

```
Etudiant e = new Etudiant("Dark Vador le gentil");
Etudiant copie = new Etudiant(e);
passageEtudiant(copie);
```

ou plus simplement :

```
Etudiant e = new Etudiant("Dark Vador le gentil");
passageEtudiant(new Etudiant(e));
```

Considérons maintenant la classe `Professeur` suivante :

```
public class Professeur {

    String nom;

    Professeur(String nom){
        this.nom=nom;
    }

}
```

et ajoutons un attribut `Professeur` à la classe `Etudiant` :

1. la méthode `clone` retourne un `Object`, le code `(Etudiant)` permet ici de preciser que c'est un étudiant que l'on copie et recupère.

```java
public class Etudiant  {
    String nom;
    static String IUT = "IUT de Bordeaux";
    Professeur profPOO;

    Etudiant(String nom){
        this.nom=nom;
    }

    Etudiant( Etudiant e){
        nom=e.nom;

    }
}
```

Soit la méthode :

```java
void passageEtudiant(Etudiant e){
   System.out.println("Debut methode: le nom de l'etudiant e vaut "+e.nom);
   e.nom="Tetouillou le terrible dragon";
   e.profPOO.nom="Gothmog";
   System.out.println("Fin methode: le nom de l'etudiant e vaut "+e.nom);
}
```

Le code :

```java
Etudiant e = new Etudiant("Dark Vador le gentil");
e.profPOO=new Professeur("Alphonse");
System.out.println("Avant methode: le nom de l'etudiant e vaut "+e.nom);
Etudiant copie = new Etudiant(e);
passageEtudiant(copie);
System.out.println("Apres methode: le nom de l'etudiant e vaut "+e.nom);
```

provoque le message d'erreur :

```
Exception in thread "main" java.lang.NullPointerException
```

Exercice 27. *(⋆⋆) Pourquoi ? Comment faire pour empêcher cela ?*

2.2 Encapsulation

Définition 2.3 (encapsulation)**.** *L'encapsulation permet de protéger l'information contenue dans un objet (en particulier, ses attributs) et de ne proposer que des méthodes de manipulation de cet objet.*

Ainsi, les propriétés associées à l'objet sont assurées par les méthodes de l'objet. L'utilisateur extérieur ne peut pas modifier directement l'information et risquer de mettre en péril les propriétés de l'objet.

L'objet est ainsi vu de l'extérieur comme une boîte noire ayant certaines propriétés et ayant un comportement spécifié. La manière dont ces propriétés ont été implémentées est cachée aux utilisateurs de la classe.

En Java, l'encapsulation intervient dans la structuration des classes dans une hiérarchie de paquets, ainsi que via des règles de visibilités pour les attributs et les méthodes d'un objet.

2.2.1 Paquets

Définition 2.4 (paquet)**.** *Un paquet (package) est un ensemble de classes réunies dans un répertoire. On déclare le paquet (qui peut être un sous-paquet d'un paquet) avec le mot-clef* `package`*.*

Par exemple, pour déclarer une classe dans le sous-paquet `monpackage` du paquet `coucou`, on écrit en première ligne du code source de la classe :

```java
package coucou.monpackage;
```

Il est possible d'utiliser ces classes via un `import` :

```java
import coucou.monpackage.*
```

À noter : `coucou` est un répertoire contenant le sous-répertoire `monpackage`. Les binaires (les fichiers `.class`) doivent être placés dans le répertoire correspondant à leur paquet. Les fichiers sources (`.java`) ne sont pas soumis à cette contrainte.

Exercice 28. *(⋆) Mettre la classe* `MainAppli` *dans un paquet* `application` *et les deux classes* `Etudiant` *et* `Professeur` *dans un paquet* `personnes`

.

2.2.2 Visibilité

Déclaration d'une classe La visibilité d'une classe peut être définie à l'aide d'un mot-clef précédant `class` :

```java
[rien, public, final, abstract] class MyClass {
...
}
```

La signification de ces mots-clefs est la suivante :
— *rien* : accessible uniquement au sein du paquet (package)
— *public* : accessible par tout le monde
— *final* : la classe ne peut pas être dérivée (pas de classe fille)
— *abstract* : classe abstraite (ne peut pas être instanciée)
Le nom de la classe peut être suivi également d'autres mots-clefs : *extends* (héritage), *implements* (implémentation d'interface). Nous verrons le rôle de ces mots-clefs ultérieurement.

Par exemple :

```java
public class Etudiant extends Individu implements Studieux {
        ...
}
```

Comme la classe `Etudiant` est publique, Java impose que son code soit dans un fichier de nom `Etudiant.java`. D'une manière générale, il est recommandé de toujours appliquer cette convention, y compris pour les classes qui ne sont pas publiques.

Déclaration des variables Prototype de déclaration d'une variable :

```java
[Specificateur acces][Specificateurs particuliers] Type NomVariable ;
```

Spécificateur d'accès :
— *public* : accessible depuis partout où la classe est accessible ;
— *private* : accessible uniquement depuis la classe elle-même ;
— *protected* : accessible depuis la classe elle-même et ses classes filles ;
— *rien* (package private) : accessible uniquement depuis les classes du paquet.

Spécificateurs particuliers :
— rien
— *static* : définit une variable de classe (et non d'instance).
— *final* : impose l'initialisation lors de la déclaration, ne peut pas être modifié.

Déclaration d'une méthode Syntaxe :

```java
[Specif. d'acces][Specif. particuliers] TypeDeRetour NomMethode ([parametres]) [exceptions]
{
        ...
}
```

Spécificateurs d'accès : rien, public, private, protected (même signification que pour une variable)
Spécificateurs particuliers : rien, final (la méthode ne peut pas être redéfinie dans une classe fille), static (méthode statique)

2.2.3 Accesseurs

Définition 2.5 (Accesseurs)**.** *Les accesseurs sont des méthodes, et normalement les seules, qui accèdent directement aux attributs d'une instance.*

Ainsi, ces méthodes sont utilisées dans les constructeurs, mais aussi dans toute autre méthode qui nécessite une donnée particulière de l'objet.

Par convention, les accesseurs permettant d'accéder aux valeurs des attributs ont toujours pour nom `get + le nom` de la variable et les accesseurs permettant de modifier les valeurs des variables `set + le nom` de l'attribut.

Le grand intérêt des accesseurs est de rendre indépendant tout le reste du code de la représentation de l'objet.

Exercice 29. *(⋆) Déclarez l'attribut* `nom` *de la classe* `Etudiant` *comme privé, puis créez les accesseurs pour cet attribut. Adaptez le reste du code en fonction. Recommencez pour la classe* `Professeur` *en utilisant les outils dédiés de* `NetBeans/eclipse`*.*

On souhaite maintenant que nos étudiants puissent jouer au Seigneur des Anneaux. Le problème est de créer une classe `Anneau` qui certifie qu'il n'y aura qu'au plus une instance d'un anneau. Autrement dit, l'anneau dans l'application sera unique, quoi que fasse les autres classes.

Exercice 30. *(⋆⋆) Comment faire ? Justifiez votre réponse.*

Fiche de synthèse

Concepts de base de la programmation objet

- attributs statiques
- méthodes statiques
- constructeur par copie
- encapsulation : visibilité d'une classe, de ses attributs, de ses méthodes
- accesseurs

Java

- passages de paramètres à une méthode : par copie pour un type de base, par copie de la référence pour les objets (donc en modification de l'objet référencé)
- niveaux de visibilité : `rien`, `public`, `protected`, `orivate`
- paquets et les règles de visibilité associées : `package`, `import`

Exemple de code (utilisation d'un attribut statique pour comptabiliser le nombre d'instances) :

```
public class Etudiant {
    String nom;
    static int nbTotalEtudiants=0;

    Etudiant(String nom){
        this.nom=nom;
        nbTotalEtudiants++;
    }
}

public class MainAppli {

    MainAppli(){
        Etudiant bob = new Etudiant("bob");
        Etudiant greg = new Etudiant("Greg");
        Etudiant Ralf = new Etudiant("Ralf");
        System.out.println(Etudiant.nbTotalEtudiants);
    }

    public static void main(String[] args) {
        new MainAppli();
    }
}
```