Conception et Programmation Objet - GM3

Interfaces, classes abstraites et un peu plus

Mathieu Bourgais

2023

Plan

Raisons de la question de visibilité

- La modularité induite par la programmation orientée objet amène à la réutilisation de classes déjà existantes, et donc à la prise en compte de cette réutilisation dès la conception.
- Pour améliorer la réutilisation, on peut placer ses classes dans différents *packages* (notion qui sera abordée plus tard dans le semestre).
- Quoi qu'il en soit, on doit créer des classes qui peuvent être réutilisées en limitant les futures mauvaises utilisations ; donc en n'exposant que ce que l'on souhaite exposer.

Private, Protected, Public

On dispose, en Java, de 4 niveaux de visibilité :

- **Private** : L'élément n'est visible que de la classe elle-même.
- **Protected** : L'élément est visible de la classe, de ses sous-classes (quelles soient dans le même package ou pas) et des autres classes du package.
- **Public** : L'élément est visible de toutes les classes, même hors du package.
- " " : Si on ne spécifie rien, l'élément est visible uniquement des autres classes du même package.

On peut spécifier la visibilité de tous les éléments : classes, attributs ou méthodes.

Démonstration de visibilité

```
public class Test{
    public int attribut1; //accessible sous la forme
        test1.attribut1;
    protected int attribut2; //accessible aux sous-
        classes sous la forme test1.attribut2 sinon
        getter et setter
    private int attribut3; //setter et getter
        obligatoires pour etre manipule

    public void methode1(){
    /*methode accessible partout*/
    }
    private void methode2(){
    /*methode uniquement utilisable dans la classe
        Test*/
    }
}
```

Membres statiques

On peut avoir besoin qu'un élément (attribut ou méthode) ne soit pas lié à une instance particulière d'une classe. Dans ce cas, on le déclare comme **static**.

Les membres *static* :

- Ne sont pas rattachés à une instance
- Existent avant toute instanciation
- Ne peuvent pas accéder aux autres membres non statiques des instances
- Sont partagés par toutes les instances

On accède aux éléments *static* en écrivant MaClasse.attributStatic ou MaClasse.methodeStatic().

Exemples d'utilisation du static

- La méthode **main** est toujours déclarée comme *static* ; elle n'est pas reliée à une instance de sa classe et est même exécutée avant toute création d'instance.
- Un attribut *static* peut être vu comme une variable globale dans le programme ; on peut s'en servir pour faire un compteur par exemple.
- Une méthode *static* permet d'être utilisée sans avoir besoin d'une instance ; par exemple l'utilisation des fonctions de la classe Math (Math.random() ; Math.abs(double a) ; Math.sqrt(double a) ; etc.)

Membres finaux

On peut déclarer des constantes avec le mot-clé **final**.

- *final type attribut* : L'attribut prend une valeur lors de sa déclaration, cette valeur ne peut être modifiée par la suite.
- *methode(final type param, ...)* : Le paramètre est contant.
- *final type methode(...)* : La méthode ne peut pas être redéfinie lors d'un héritage.
- *final class MaClasse* : La classe ne peut pas être dérivée.

Les constantes ne sont pas obligatoirement liées à une instance, on peut donc avoir des *final static*.

Plan

Un peu de théorie

- Le paradigme de la programmation orientée objet met l'accent sur la modélisation tout en garantissant la sécurité des données et des méthodes.
- Pour assurer ces deux principes ensembles, on fait de l'**encapsulation**.
- Les classes sont issues de l'encaspulation (on réunit dans une même unité des données et des méthodes qui agissent sur ces données)
- Les packages sont issus de l'encapsulation (on regroupe des classes entre-elles)
- Les concepts de visibilité, d'héritage et de polymorphismes sont aussi liés à l'encapsulation.

Envoie de messages

Derrière l'encapsulation, il y a l'idée philosophique d'envoyer des messages (les méthodes) aux objets et de laisser les instances, potentiellement hétérogènes, s'occuper de l'exécution de ces messages en fonction de leur état interne.

Quand on écrit *instance1.maMethode()* et *instance2.maMethode()*, on envoie le même message (*maMethode()*) à deux instances différentes, potentiellement de deux objets différents. C'est alors l'instance qui est responsable de l'exécution du message.

La JVM va chercher la méthode maMethode() dans les classes dont sont issues instance1 et instance2. Si elle ne trouve pas la méthode, elle va remonter l'arbre des héritages jusqu'à trouver la méthode dans les super-classes.

Héritage de la classe Object

En Java, toutes les classes héritent "naturellement" de la classe **Object** ; les méthodes suivantes peuvent donc être redéfinies :

- **String toString()** : permet de "transformer" une classe en une chaîne de caractères lorsqu'on souhaite "écrire" cette classe
- **Object clone()** : permet de copier un objet
- **Boolean equals(Object o)** : permet de redéfinir l'égalité entre deux objets
- **int hashCode()** : méthode utilisée avec *equals()*
- **Class getClass()** : permet de connaître la classe d'une instance.

Redéfinition de toString()

```
public class Test{
    ...
    public String toString(){
        return "C'est␣une␣classe␣test";
    }
}

Test test1 = new Test();
System.out.println(test1);
//Le terminal affichera "C'est une classe test"
```

Classes abstraites

Du point de vue de la modélisation, avec l'encapsulation en tête, on peut souhaiter déclarer des méthodes dans des classes mères pour les définir uniquement dans des classes filles. On parle alors d'abstraction avec le mot-clé **abstract**.

- *abstract type methode() ;* : le comportement de la méthode sera défini dans des classes filles
- *abstract class MaClasse{...}* : la classe est abstraite, elle contient au moins une méthode abstraite

Une classe abstraite ne peut pas être instantiée. On dit qu'il faut alors une classe concrète, héritant de la classe abstraite et redéfinissant toutes les méthodes abstraites de celle-ci, pour créer une instance.

Interfaces

Toujours dans un but de propreté de la modélisation, on peut souhaiter imposer un ensemble de méthodes abstraites à plusieurs classes, sans pour autant gonfler artificiellement une classe abstraite. Pour contourner l'héritage simple de Java, on peut déclarer des **interfaces** qui seront alors implémentées par des classes (**implements**).

```
public interface MonInterface {
    public void uneMethode();
}

public class MaClasse implements MonInterface{
    public void uneMethode(){
        //on definit ce que fait la methode
    }
}
```

Bonus sur les interfaces

- Une interface peut être vue comme une classe "abstraite pure"
- Un attribut déclaré dans une interface est forcément une constante (donc *final* et *static*).
- Une interface peut hériter d'autres interfaces (de plusieurs interfaces en même temps).
- Sur les versions récentes de Java, il est possible d'écrire du code pour définir le comportement de méthodes directement dans une interface.
- Une classe peut implémenter plusieurs interfaces en même temps.

Plan

Modifications du jeu

- La méthode parler() peut être définie comme abstraite dans la classe Personnage
- On ajoute la possibilité pour chaque personnage d'attaquer un autre personnage. Les PJ attaquent des monstres en fonction de la valeur de défense de ces derniers, les monstres attaquent n'importe quel personnage en fonction de leur propre valeur d'attaque. Cette attaque utilisera le hasard
- Le nom des joueurs des PJ devient une constante
- Les magiciens sont obligatoirement des soigneurs
- On ajoute des personnages Medecins qui sont aussi des soigneurs
- La méthode toString() est redéfinie au niveau de Personnage

Exemple des nouveautés

```
public interface Soigneur {
    public void soigner(Personnage p);
}

public abstract class Personnage{
    public abstract String parler();
}

public class Medecin extends Personnage implements
    Soigneur{
    public void soigner(Personnage p){
        p.setVie(p.getVie()+5);
    }
    public String parler(){
        return "Je suis un medecin";
    }
}
```

Suite de l'exemple

```
public class PJ extends Personnage {
    private final String nomJoueur;

    public void attaquer(Monstre m) {
        int valeurAttaque = Math.toIntExact(Math.
            round(100.0*Math.random()));
        if(valeurAttaque>m.getDefense()){
            m.setVie(m.getVie()-5);
            System.out.println("Attaque reussi");
        } else {
            System.out.println("Attaque manquee");
        }
    }
}
```

Nouveau main

```
String nomMonstre;
Scanner scan = new Scanner(System.in);
System.out.println("Entrez le nom du monstre :");
nomMonstre = scan.next();
detraqueur.setNom(nomMonstre);

detraqueur.attaquer(dudley);
dudley.attaquer(detraqueur);
hermione.soigner(detraqueur);
quinn.soigner(dudley);

System.out.println(detraqueur.getVie());
System.out.println(dudley.getVie());

scan.close();
```

Résultat

```
Entrez le nom du monstre :
Patrick
Bonjour, je m'appelle Patrick
GROAR !
Bonjour, je m'appelle Hermione Granger et je suis joué par Mathieu
Bonjour, je suis un magicien
Hermione Granger dit : Voilà un terrible détraqueur
Bonjour, je m'appelle Dudley Dursley et je suis joué par Aurore
Dudley Dursley dit : oh, j'ai peur !
Attaque monstre réussi
Attaque manquée
20
12
```

Le code complet menant à ce résultat est à retrouver sur moodle.