TD 4 : encapsulation, listes chaînées

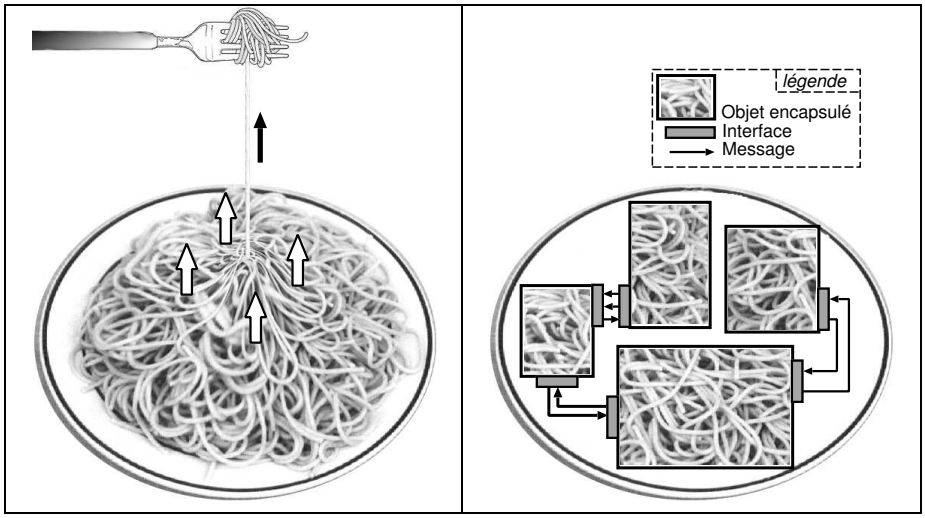
1 Encapsulation : `public` / `private`

Pour comprendre l'utilisation des protections, comme `public` et `private`, il faut d'abord en comprendre l'intérêt. Voici une brève introduction.

1.1 Intérêt du génie logiciel

Le génie logiciel, dont la conception orientée objet est un des aspects importants, est né d'un constat pratique. La gestion de la complexité d'un logiciel est le principal enjeu du développement logiciel. En d'autres termes, s'il est facile d'écrire un programme implémentant une fonctionnalité précise, il est très difficile d'écrire un système logiciel de plus grande taille qui puisse évoluer. Un petit programme (moins de 3000 lignes) peut être appréhendé dans son ensemble par un individu, et aucune méthodologie particulière n'est nécessaire à sa conception. Lorsque la taille et le nombre de développeurs croissent, on atteint rapidement un point de blocage. Le temps alors passé à comprendre, reconcevoir, et réimplémenter, dépasse de loin le temps passé à ajouter de nouvelles fonctionnalités. Une méthodologie devient donc indispensable.

1.2 Boîtes noires



Voici la métaphore du plat de spaghetti : sur la figure de gauche, un logiciel peut rapidement devenir très complexe (comme un plat de spaghetti collant). Lorsqu'on cherche à modifier un élément (retirer un spaghetti) tout est si imbriqué, que cela a des répercussions sur l'ensemble du logiciel. Sur la figure de droite, dans la conception orientée objet on encapsule la complexité dans des boîtes noires communiquant par des interfaces bien définies. Les répercussions d'une modification sont alors mieux contrôlées.

1.3 Les différentes casquettes / rôles

Lorsqu'on écrit un programme orienté objet, on va successivement prendre deux casquettes différentes : celle du *concepteur*} d'une classe et celle de l'*utilisateur* d'une classe. L'utilisateur de la classe ne devra la manipuler qu'au travers d'une interface bien définie. Le concepteur devra lui définir cette interface et décider ce que l'utilisateur aura le droit ou non de faire.

1.4 `public` / `protected` / `package` / `private`

Le concepteur de la classe pourra interdire l'accès de certains attributs et méthodes d'une classe en les déclarant `private`. Les éléments déclarés `public` constitueront l'interface pour l'utilisateur évoqué précédemment.

En ce qui concerne `protected`, il s'agit d'un rôle (casquette) supplémentaire : nous avons vu `public` (accessible l'utilisateur) et `private` (défini par le concepteur). Le nouveau rôle c'est celui de la personne qui étendrait la classe en utilisant le mécanisme de l'héritage. Les données `protected` sont donc accessibles aux sous-classes, mais pas à l'utilisateur.

Finalement il y a l'attribut `package`. C'est la protection par défaut (quand on met rien) en Java. Elle se comporte comme `public` pour les les classes dans le même paquetage et comme `private` pour les autres.

1.5 Quelle protection choisir ?

Le choix d'une protection est délicat. Il exige une réflexion approfondie sur l'utilisation qui sera faite de la classe qu'on est en train d'écrire. Pas assez de protection ... et on finira avec un plat de spaghettis. Trop de protection ... et le code sera excessivement alourdi. N'oublions pas que l'objectif est de maîtriser la complexité : en sur-protégeant on crée plus de complexité que ce que l'on réduit.

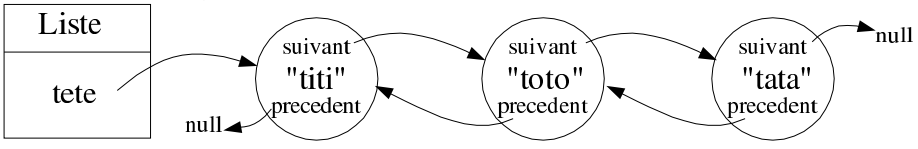
1.6 Exercices

- Écrire une classe `Point` qui représente un point en deux dimensions. Quelle protection faut-il utiliser pour les coordonnées x et y ?
- Dans la classe `Pile` ci-dessous, ajoutez les protections qui vous semblent necessaires, en argumentant.

```
class Pile
{
    int tailleMax=100;
    int position=-1;
    String donnees[]=new String[tailleMax];
    void empiler(String valeur)
    {
        if(position+1>=tailleMax){System.out.println("pile pleine");return;}
        donnees[++position]=valeur;
    }
    String hautDePile()
    {
        if(position<0){System.out.println("pile vide");return null;}
        return donnees[position];
    }
    String depiler()
    {
        if(position<0){System.out.println("pile vide");return null;}
        return donnees[position--];
    }
};
```

2 Liste chaînées

Une liste chaînée permet de stoker des informations de manière à ce que certaines opérations, comme l'insertion d'un élément, soient efficaces. Voici un schéma decrivant une liste chaînée :



Voici un exemple d'utilisation de cette liste chaînée :

```
class ListeTest
{
    public static void main(String [] args)
    {
        Liste liste=new Liste();
        liste.insererEnTete("titi");
        liste.insererEnTete("toto");
        liste.insererEnTete("tata");
        Noeud n=liste.debut();
        n=n.suivant;
        liste.insererAvant(n,"zozo");
        liste.afficher();
    }
}
```

- Écrire les corps des classes `Liste` et `Noeud` (on n'écrira pas encore le détail des méthodes).
- En vous inspirant du schéma ci-dessus, dessiner les différentes étapes nécessaires pour insérer un noeud au début de la liste (méthode `insererEnTete`).
- Écrire la méthode `insererEnTete`.
- En vous inspirant du schéma ci-dessus, dessiner les différentes étapes nécessaires pour insérer un noeud quelque part à l'intérieur de la liste, avant un noeud donné (méthode `insererAvant`). On se mettra dans le cas général, c'est à dire ni au début, ni à la fin de la liste.
- Écrire la méthode `insererAvant`.
- Tenir compte des cas particuliers (début et fin de la liste).
- Écrire les autres méthodes nécessaires pour que l'exemple puisse fonctionner.
- Réfléchissez à la protection (`public`/`private`) de chacun des attributs et méthodes des deux classes.