Séquence 8

Objets imbriqués

Les objets sont comme des poupées russes : à l'intérieur d'un objet, il peut y avoir un ou plusieurs objets. C'était déjà le cas de notre premier exemple, celui des comptes bancaires. Un compte bancaire comprend une variable `titulaire` qui est de type `String` : sa valeur est donc un objet.

Les variables d'instance d'un objet peuvent contenir des objets. C'est très fréquent et cela pose parfois des problèmes aux débutants. Nous allons voir comment les principes généraux de conception des classes doivent être mis en oeuvre dans le cas des objets imbriquées et plus particulièrement dans le cas des collections, c'est-à-dire des objets qui comportent dans leurs variables d'instance plusieurs objets d'un même type.

1 Comptes et titulaires

Nous allons proposer une version des comptes bancaires où les titulaires ne sont pas représentés par leur seul nom mais par un objet d'une classe que nous allons créer. Un compte contiendra un titulaire qui lui-même contiendra un nom, un prénom et une date de naissance qui sont aussi des objets. Il y aura donc trois niveaux d'imbrication d'objets.

```
1  import java.util.Date;
2  public class Titulaire{
3      private String nom, prenom, lieuNaissance;
4      private Date dateNaissance;
5      public Titulaire(String n, String p, Date d, String l){
6          nom = n;
7          prenom = p;
8          dateNaissance = d;
9          lieuNaissance = l;
10     }
11     public String getNom(){
12         return nom;
13     }
14     public String getPrenom(){
15         return prenom;
16     }
17     public String getLieuNaissance(){
18         return lieuNaissance;
19     }
20     public Date getDateNaissance(){
```

```
21              return dateNaissance;
22          }
23          public String toString(){
24              return nom + "␣" + prenom + "␣né␣le␣" + dateNaissance + "␣à␣" +
25                  lieuNaissance;
26          }
27      }
28      public class CompteTitulaire{
29          private Titulaire titulaire;
30          private int numero;
31          private double solde;
32          public CompteTitulaire(Titulaire tit, int num){
33              titulaire = tit;
34              numero = num;
35              solde = 0.0;
36          }
37          public double getSolde(){
38              return solde;
39          }
40          public void deposer(double montant){
41              if (montant<0){
42                  throw new IllegalArgumentException("Un␣montant␣ne␣peut␣pas␣être␣négatif");
43              }else{
44                  solde = solde + montant;
45              }
46          }
47          public void retirer(double montant){
48              if (montant<0){
49                  throw new IllegalArgumentException("Un␣montant␣ne␣peut␣pas␣être␣négatif");
50              }else{
51                  solde = solde -montant;
52              }
53          }
54          public int getNumero(){
55              return numero;
56          }
57          public Titulaire getTitulaire(){
58              return titulaire;
59          }
60          public String toString(){
61              return "Titulaire:␣" + titulaire + "␣numéro:␣" + numero + "␣solde:␣" +
62                  solde;
63          }
64      }
65      public class MainCT{
66          public static void main(String[] args){
67              Date date = new Date(1991-1900,8-1,13);
68              Titulaire tit = new Titulaire("Dupont","Jean",date,"Toulon");
69              CompteTitulaire compte = new CompteTitulaire(tit,101236);
70              Titulaire tit2;
71              String chaine;
72              int mois;
73              compte.deposer(1250);
```

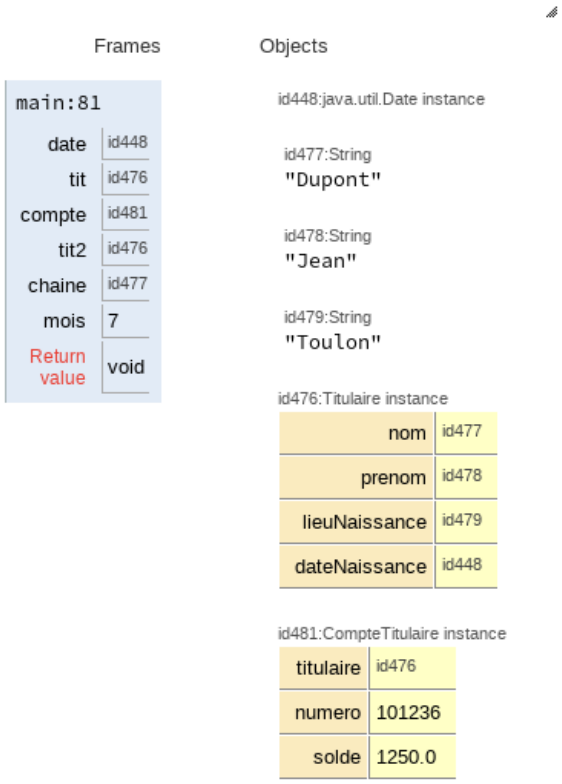
```
74              System.out.println(compte);
75              tit2 = compte.getTitulaire();
76              System.out.println(tit2);
77              chaine = compte.getTitulaire().getNom();
78              System.out.println(chaine);
79              mois = compte.getTitulaire().getDateNaissance().getMonth();
80              System.out.println(mois);
81      }
82  }
```

La méthode `main` illustre le fait qu'un accesseur peut renvoyer un objet et qu'il est possible d'invoquer une méthode sur cet objet. Prenons par exemple la ligne 77. La méthode `getTitulaire` de la `CompteTitulaire` renvoie un objet instance de la classe `Titulaire`. Sur cet objet, on peut appeler la méthode `getNom` de la classe `Titulaire`. Sur cette ligne, il y a deux invocations de méthodes qui sont invoquées chacune sur un objet distinct, l'un étant présent dans une variable d'instance de l'autre.

La ligne 79 reprend le même principet avec cette fois trois invocations sur trois objets différents et de types diéfférents. La méthode `getTitulaire` est appelée sur `compte` (instance de `CompteTitulaire`) ce qui renvoie un objet instance de la classe `Titulaire`. Sur cet objet est appelée la méthode `getDateNaissance` qui renvoie un objet instance de la classe `Date`. Sur cet objet est appelée la méthode `getMonth` qui renvoie un entier.

2 L'imbrication vue avec Pythontutor

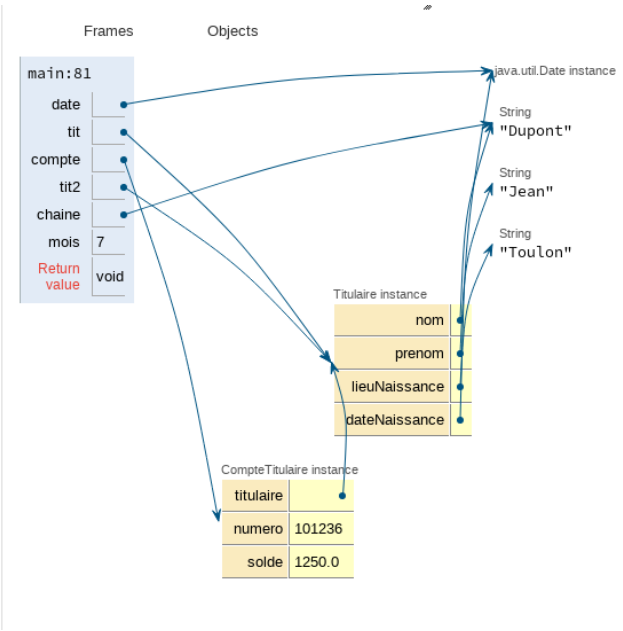
Prenons le programme proposé ci-dessus et faisons-le exécuter par `Pythontutor`. Nous allons d'abord choisir les options *render all objects on the heap* et *use text labels for pointers*. Cela donne la mémoire suivante (on n'a gardé ici que la partie de la page Pythontutor qui affiche la mémoire).



On voit que les variables d'instances des objets qui contiennent d'autres objets ne contiennent pas directement les objets en question mais des références permettant d'identifier ces objets qui sont situés ailleurs dans le tas. Les variables d'instance sont comme les variables de méthodes situées dans la pile : elles contiennent des références.

Par exemple, la variable `titulaire` de l'objet de type `CompteTitulaire` contient la référence `id476`. C'est la même référence que celle contenue dans les variables `tit` et `tit2` de la méthode `main` que l'on voit dans la pile.

Si l'on préfère la représentation des références au moyen de flèches, cela nous donne la représentation suivante.



On voit qu'il y a des flèches entre la pile et le tas, mais également des flèches entre divers objets du tas, partant d'un objet et arrivant à un autre. Cela matérialise l'imbrication de l'objet où arrive la flèche dans celui d'où part la flèche, depuis une de ses variables d'instance. Il est possible ainsi de définir des chemins qui partent de la pile, avec une variable de la méthode `main` et qui arrivent à un objet du tas en passant éventuellement par un ou deux objets intermédiaires.

3 Protéger les objets imbriqués ?

Dans le code du programme proposé, le programmeur peut avoir accès à l'objet `Titulaire` imbriqué dans l'objet représentant le compte. Pour ce faire, il lui suffit d'utiliser la méthode `getTitutlaire`. Cet objet peut être mis dans une variable et manipulé au moyen de toutes les méthodes de l'objet.

Il est possible de procéder différemment, de cacher les objets imbriqués dans l'objet englobant de façon à ce qu'il soit impossible d'y accéder autrement que via une méthode de l'objet englobant. Cela permet d'assurer des propriétés d'intégrité, notamment quand l'objet est mutable.

Prenons un autre exemple d'imbrication : supposons qu'on veuille que les opérations de retrait et de dépôt sur un compte se fassent nécessairement au guichet d'une banque. On va garder la classe compte inchangée, mais on va cacher l'objet compte à l'intérieur d'un objet représentant le guichet bancaire. Et chaque opération que l'on voudra faire sur un compte va être faite en utilisant un intermédiaire, une méthode du guichet.

```
public class Guichet{
    private CompteTitulaire[] lesComptes= new CompteTitulaire[100];
    private int nbComptes = 0;
    static private int numCompte = 1;
    // La méthode renvoie le numéro du compte créé
    public int creerCompte(Titulaire tit){
        CompteTitulaire compte = new CompteTitulaire(tit,numCompte);
        lesComptes[nbComptes] = compte;
        numCompte = numCompte + 1;
        nbComptes = nbComptes + 1;
        return compte.getNumero();
```

```
    }
    private CompteTitulaire findCompte(int num){
        int i=0;
        while (i<nbComptes && lesComptes[i].getNumero()!=num)
            i=i+1;
        if (i >= nbComptes)
            throw new IllegalArgumentException("Numero␣de␣compte␣invalide␣" +
                                                num);
        return lesComptes[i];
    }
    public void deposer(int num, double montant){
        findCompte(num).deposer(montant);
    }
    public void retirer(int num, double montant){
        findCompte(num).retirer(montant);
    }
    public double getSolde(int num){
        return findCompte(num).getSolde();
    }
    public String infosCompte(int num){
        return findCompte(num).toString();
    }
}
public class MainGuichet{
    public static void main(String[] args){
        Titulaire t1 = new Titulaire("Jean","Jean",new Date(),"Nice");
        Titulaire t2 = new Titulaire("Jeanne","Jeanne",new Date(),"Menton");
        Guichet guichet = new Guichet();
        int numero1, numero2;
        numero1 = guichet.creerCompte(t1);
        numero2 = guichet.creerCompte(t2);
        guichet.deposer(numero1,400);
        guichet.retirer(numero1,52.5);
        System.out.println(guichet.infosCompte(numero1));
    }
}
```

Quelques commentaires sur la classe `Guichet` : les méthodes des comptes (`retirer, déposer, getSolde`) se retrouvent dans la classe avec un paramètre en plus : le numéro du compte sur lequel il faut faire l'opération. Comme chacune de ces opérations commence par aller chercher le compte qui a ce numéro, plutôt que de répéter le même code dans les trois méthodes, on a créé une méthode `findCompte` qui fait cette opération de recherche. Grâce au mode `private`, cette méthode n'est utilisable qu'à l'intérieur de la classe.

Les comptes sont créés dans la classe, stockés dans la classe et n'en sortent jamais. Dans le `main`, je n'ai aucun moyen d'accéder aux comptes gérés par la guichet. Je n'ai aucune méthode ni aucune variable accessible qui me donnerait un des deux instances de `CompteTitulaire`.

Autre chose : pourquoi avoir deux variables pour le nombre de comptes et le numéro de compte : apparemment, c'est la même chose. La réponse est que ce n'est la même chose que s'il n'y a qu'un seul guichet. S'il y a plusieurs guichets, `nbComptes` est le nombres de comptes du guichet et `numCompte` est le numéro du prochain compte qui est égal au nombre total de comptes créés dans tous les guichets (ce nombre plus un, pour être précis).

4 Les collections

Il arrive extrêmement souvent qu'un objet comporte plusieurs objets du même type. Par exemple, une bibliothèque contient plusieurs livres. Chaque livre est un objet Java et la bibliothèque en est un autre qui contient tous les livres. Une armée contient des soldats. Une classe des élèves, un parking, des voitures, etc.

Il y a des opérations plus ou moins standards : ajouter un élément, supprimer un élément, retrouver un élément ou tous les éléments qui ont une certaine propriété. L'important dans ce cas de figure pour respecter les principes de la programmation objet, c'est de mettre les opérations dans la bonne classe. Les opérations concernant un élément de la collection doivent être dans la classe des éléments et celles qui concernent la gestion de la collection doivent être dans la classe de la collection.

Prenons le cas des vaches. Un éleveur peut représenter un bovin par un numéro d'immatriculation, un nom, son poids. Un troupeau sera un ensemble de bovins. De nouveaux bovins pourront venir par naissance ou par achat et des bovins quitteront le troupeau soit pour aller à la boucherie, soit pour aller à l'équarissage (en cas de mort prématurée), soit encore pour être vendu vivant.

Dans cet exemple comme dans beaucoup de cas, le numéro d'immatriculation permet d'identifier de manière unique un individu. Il s'agit ici d'un numéro du monde réel : en France, les bovins sont immatriculés par un nombre à 10 chiffres matérialisé par deux étiquettes, une dans chaque oreille de la bête. Ce nombre est unique tout au long de la vie du bovin. Dans certains cas, lorsqu'il n'existe pas de clé, on en crée une spécifique au programme que l'on écrit et qui permet d'identifier un individu. On peut également utiliser la référence de l'objet comme identifiant unique d'un objet.

Les opérations sur les bovins sont de créer un objet, changer son poids. Les opérations sur le troupeau sont d'enregistrer une arrivée, un départ, de retrouver un bovin à partir de son numéro ou de son nom. Enfin, il faut des `toString` dans les deux classes.

```
public class Bovin{
    private int immat;
    private String nom;
    private int poids;
    public Bovin(int im, String no, int po){
        immat = im;
        poids = po;
        nom = no;
    }
    public void setPoids(int po){
        poids = po;
    }
    public int getImmatriculation(){
        return immat;
    }
    public int getPoids(){
        return poids;
    }
    public String getNom(){
        return nom;
    }
    public String toString(){
        return nom + "␣numero:␣" + immat + "␣poids:␣" + poids;
    }
}
```

```
public class BovinNotFoundException extends RuntimeException{}
public class Troupeau{
    private Bovin[] troupe = new Bovin[100];
    private int nbBovins = 0;
    public Bovin rechercherBovinParNom(String nom){
        int ind = 0;
        while(ind<nbBovins && ! nom.equals(troupe[ind].getNom()))
            ind++;
        if (ind<nbBovins)
            return troupe[ind];
        else
            throw new BovinNotFoundException();
    }
    public Bovin rechercherBovinParNumero(int num){
        int ind = 0;
        while(ind<nbBovins && num != troupe[ind].getImmatriculation())
            ind++;
        if (ind<nbBovins)
            return troupe[ind];
        else
            throw new BovinNotFoundException();
    }
    public void ajouterBovin(Bovin bov){
        troupe[nbBovins] = bov;
        nbBovins++;
    }
    public void retirerBovin(int num){
        int ind = 0;
        while(ind<nbBovins && num != troupe[ind].getImmatriculation())
            ind++;
        if (ind<nbBovins){
            for (int i=ind; i<nbBovins; i++){
                troupe[i]=troupe[i+1];
            }
            nbBovins = nbBovins -1;
        }else{
            throw new BovinNotFoundException();
        }
    }
    public String toString(){
        String res = "";
        if (nbBovins == 0)
            return "Pas␣de␣bovin␣dans␣le␣troupeau!";
        else{
            for (int i=0; i<nbBovins; i++){
                res = res + troupe[i].toString() + "\n";
            }
            return res;
        }
    }
}
```

On a choisi d'utiliser un tableau pour stocker les objets de type `Bovin`. On aurait pu utiliser un

`ArrayList`. Dans le code, on utilise le fait que le nombre de bovins du troupeau est également l'indice de la première case vide du tableau. Et lorsqu'on enlève un bovin, on décale d'une case avec une boucle for tous les objets qui ont un indice supérieur à celui de l'objet à supprimer.

Si l'on veut modifier le poids d'un bovin, il faut d'abord utiliser une des deux méthodes de recherche de la classe `Troupeau` : recherche par nom ou par numéro, ce qui permet de récupérer l'objet Java représentant l'individu, puis sur celui-ci appeler la méthode `setPoids`.

```
public class TesteTroupeau{
    public static void main(String[] args){
        Bovin bov;
        Troupeau troupe = new Troupeau();
        System.out.println(troupe);
        bov = new Bovin("Marguerite",101,307);
        System.out.println(bov);
        troupe.ajouterBovin(bov);
        bov = new Bovin("Julie",504,57);
        troupe.ajouterBovin(bov);
        bov = new Bovin("Paquerette",582,650);
        troupe.ajouterBovin(bov);
        System.out.println("Troupeau:");
        System.out.println(troupe);
        troupe.rechercherBovinParNom("Julie").setPoids(82);
        System.out.println("Troupeau:");
        System.out.println(troupe);
        troupe.retirerBovin(101);
        System.out.println("Troupeau:");
        System.out.println(troupe);
    }
}
```