Séquence 18

Récursivité et héritage

1 Introduction

Pour qu'une méthode récursive fonctionne, il faut qu'il existe un ou plusieurs cas où l'on n'appelle pas récursivement la méthode. Cela permet au processus de s'arrêter faute de quoi la méthode est appelée indéfiniment et finit par saturer la pile.

Dans les exemples vus jusqu'ici, cette différence de cas s'exprime par un `if` dans la méthode. Si une certaine condition est vraie, la méthode s'arrête et le cas échéant, renvoie un résultat. Sinon, une invocation récursive de la même méthode est faite.

Par exemple, pour le parcours complet d'une structure récursive, la condition sera : si l'attribut qui a pour type la classe elle-même, si cet attribut a pour valeur `null`, alors le parcours est terminé. Sinon, appeler la méthode récursivement sur l'objet contenu dans cet attribut.

Nous allons voir ici qu'en combinant récursivité et héritage on peut ne pas avoir de `if` dans la méthode mais un fonctionnement équivalent en redéfinissant la méthode dans une sous-classe.

2 Point de départ : les militaires

Nous reprenons ici l'exemple des militaires qui connaissent chacun leur supérieur direct au moyen d'un attribut. Nous rappelons ici cet exemple qui est présenté en détail dans la séquence sur les structures récursives, avec deux des méthodes récursives présentées dans cette séquence : `afficheChaineHierachiqueCroissante` et `rechercheParNomBool`.

```
class Militaire{
    String nom;
    String grade;
    Militaire superieur;
    Militaire(String n, String g, Militaire s){
        nom=n;
        grade=g;
        superieur=s;
    }
    public String getNom(){
        return nom;
    }
    public String getGrade(){
        return grade;
```

```
    }
    public Militaire getSuperieur(){
        return superieur;
    }
    public Militaire setSuperieur(Militaire sup){
        superieur = sup;
    }
    public String toString(){
        return nom + "␣" + grade;
    }
    public void afficheChaineHierachiqueCroissante(){
        System.out.println(this.toString());
        if (superieur != null){
            superieur.afficheChaineHierachiqueCroissante();
        }
    }
    public boolean rechercheParNomBool(String achercher){
        if (nom.equals(achercher)){
                return true;
        }else{
            if (superieur == null){
                return false;
            }else{
                return superieur.rechercheParNomBool(achercher);
            }
        }
    }
}
```

3 Utilisation de l'héritage

On peut changer la représentation des objets pour prendre en compte le fait qu'un et un seul membre de la structure a un statut différent. Un et un seul n'a pas de chef. Au lieu de représenter ce statut spécial par la valeur `null` dans l'attribut `superieur`, on peut utiliser deux classes différentes pour représenter les militaires avec ou sans supérieur et une relation d'héritage permettant de mettre des instances de l'une et l'autre classe dans cet attribut.

Nous allons définir trois classes :

— une super-classe appelée `AbstractMilitaire`. Nous la disons abstraite (abstract) parce que le but de cette classe n'est pas d'être instanciée pour créer des objets mais de donner un type commun aux militaires avec ou sans supérieurs.

— une sous-classe `Generalissime` qui représente un militaire sans supérieur.

— une sous-classe `Subordonne` qui représente un militaire avec supérieur.

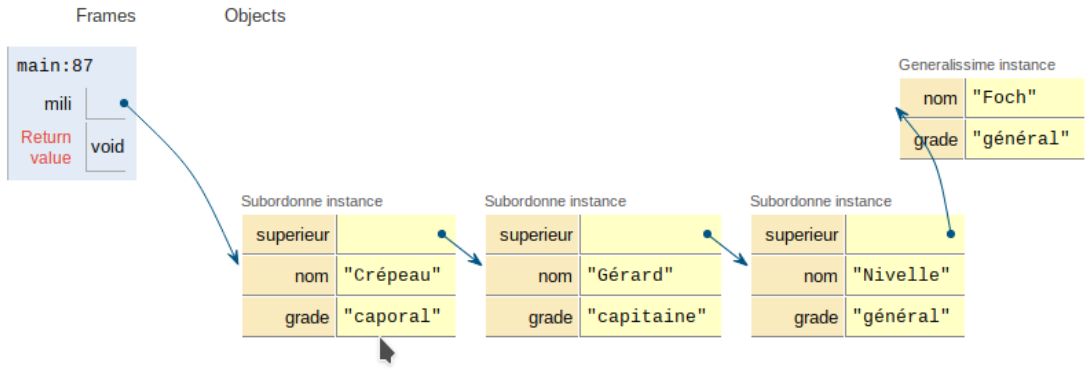
```
public class AbstractMilitaire{
    protected String nom;
    protected String grade;
    public AbstractMilitaire(String n, String g){
        nom = n;
        grade = g;
    }
```

```
    public String getNom(){
        return nom;
    }
    public String getGrade(){
        return grade;
    }
    public AbstractMilitaire getSuperieur(){
        throw new IllegalStateException("Pas␣de␣supérieur");
    }
    public void setSuperieur(AbstractMilitaire sup){
        throw new IllegalStateException("Pas␣de␣supérieur");
    }
    public String toString(){
        return nom + "␣" + grade;
    }
    public void afficheChaineHierachiqueCroissante(){
        throw new IllegalStateException("Operation␣impossible");
    }
    public boolean rechercheParNomBool(String achercher){
        throw new IllegalStateException("Operation␣impossible");
    }
}
public class Generalissime extends AbstractMilitaire{
    public Generalissime(String n, String g){
        super(n,g);
    }
    public String toString(){
        return nom + "␣" + grade;
    }
    public void afficheChaineHierachiqueCroissante(){
        System.out.println(this.toString());
    }
    public boolean rechercheParNomBool(String achercher){
        if (nom.equals(achercher)){
            return true;
        }else{
            return false;
        }
    }
}
public class Subordonne extends AbstractMilitaire{
    private AbstractMilitaire superieur;
    public Subordonne(String nom, String grade, AbstractMilitaire sup){
        super(nom,grade);
        if (sup == null){
            throw new IllegalArgumentException("Il␣faut␣un␣supérieur");
        }
        superieur = sup;
    }
    public AbstractMilitaire getSuperieur(){
        return superieur;
    }
    public void setSuperieur(AbstractMilitaire sup){
```

```
            if (sup == null){
                throw new IllegalArgumentException("Il␣faut␣un␣supérieur");
            }
            superieur = sup;
        }
        public void afficheChaineHierachiqueCroissante(){
            System.out.println(this.toString());
            superieur.afficheChaineHierachiqueCroissante();
        }
        public boolean rechercheParNomBool(String achercher){
            if (nom.equals(achercher)){
                return true;
            }else{
                return superieur.rechercheParNomBool(achercher);
            }
        }
}
public class Armee{
    public static void main(String[] args){
        AbstractMilitaire mili = new Generalissime("Foch","général");
        mili = new Subordonne("Nivelle","général",mili);
        mili = new Subordonne("Gérard","capitaine",mili);
        mili = new Subordonne("Crépeau","caporal",mili);
        mili.afficheChaineHierachiqueCroissante();
        System.out.println(mili.rechercheParNomBool("Nivelle"));
        System.out.println(mili.rechercheParNomBool("Mangin"));
    }
}
```

4 Dessin de la mémoire

Voici la mémoire à la fin de l'exécution de la méthode `main` dessinée par Pythontutor.



5 Étude de la méthode afficheChaineHierachiqueCroissante

Dans la version d'origine, le code de la méthode était le suivant :

```
    public void afficheChaineHierachiqueCroissante(){
        System.out.println(this.toString());
        if (superieur != null){
            superieur.afficheChaineHierachiqueCroissante();
        }
    }
```

Ce code est remplacé par deux méthodes différentes dans chacune des deux classes :

```
// Classe Generalissime
    public void afficheChaineHierachiqueCroissante(){
        System.out.println(this.toString());
    }
// Classe Subordone
    public void afficheChaineHierachiqueCroissante(){
        System.out.println(this.toString());
        superieur.afficheChaineHierachiqueCroissante();
    }
```

Il n'y a plus de `if`. Il y a quand même un fonctionnement conditionnel : la méthode appelée dépend du type de l'objet : si c'est la méthode de `Generalissime` qui est appelée, il n'y a pas d'appel récursif. S'il est de classe `Subordonne`, il y aura un appel à une méthode `afficheChaineHierachiqueCroissante`, une des deux.