Chapitre 8

Structures de données récursives

8.1 Introduction

La récursivité ne concerne pas seulement les traitements (les méthodes) mais également la représentation des données (classes). Dans une méthode récursive, il y a un appel à cet méthode. Dans une classe récursive, il y a une variable (ou plusieurs) dont le type est la classe elle-même.

8.2 Classe récursive

On appelle classe récursive une classe dans laquelle une ou plusieurs variables d'instance ont pour type le nom de la classe. C'est-à-dire qu'un objet instance de cette classe a des variables qui contiennent d'autres instances de la classe.

Prenons un premier exemple : celui d'une structure hiérarchique de type militaire où chaque individu a un supérieur hiérarchique unique.

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

On voit que chaque objet de la classe Militaire contient une variable de type Militaire et que le constructeur permettant de créer un objet instance prend en paramètre un autre objet représentant son supérieur.

Deux questions se posent : comment créer le premier objet de la classe, alors qu'il n'existe aucun autre objet pour être le supérieur ; à la création de l'objet, comment new peut réserver une place en mémoire assez grande pour le militaire et son chef et le chef du chef, etc ?

La réponse à la première question, c'est qu'on peut créer un objet en donnant la valeur **null** pour représenter le supérieur. Cela permet de créer non seulement le premier objet, mais autant d'objets qu'on veut et qui n'ont pas de supérieur (à leur création).

La réponse à la seconde question, c'est que le new ne réserve de place en mémoire que pour le nouvel objet. Son supérieur (si ce n'est pas null) existe déjà dans la mémoire au moment de la création de l'objet. Il n'y a pas à lui réserver de place. De même pour le chef du chef.

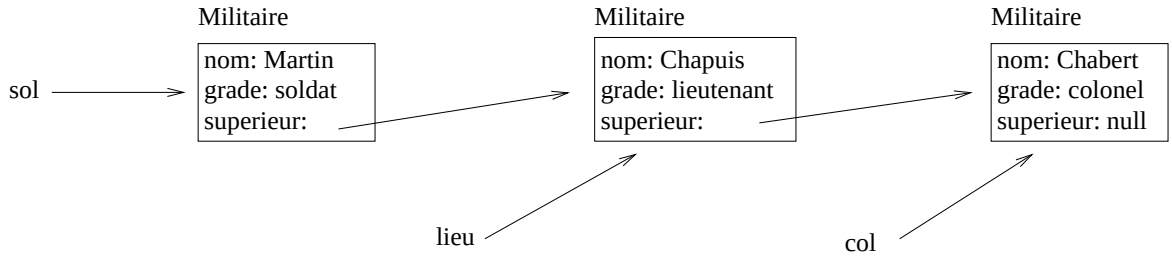
Tout le secret de la structure récursive tient dans le mécanisme des références. La variable `superieur` ne contient pas directement le supérieur mais une référence, l'adresse en mémoire du supérieur. Cette adresse peut être null, pour représenter le cas où le militaire n'a pas de chef ou le cas où cette information sur son supérieur n'a pas encore été mise à jour.

Une structure récursive contient une ou plusieurs variables du même type qu'elle-même, comme une méthode récursive contient un ou plusieurs appels à elle-même.

Voyons un programme qui utilise cette classe pour construire effectivement une structure.

```
class TestMili{
    public static void main(String[] args){
        Militaire sol, lieu, col;
        col=new Militaire("Chabert","Colonel",null);
        lieu=new Militaire("Chapuis","lieutenant",col);
        sol=new Militaire("Martin","soldat",lieu);
    }
}
```

Ce pogramme construit une structure comportant trois objets de type `Militaire` reliés entre eux. On peut la représenter comme suit (pour simplifier le dessin, on ne sépare pas la pile du tas).



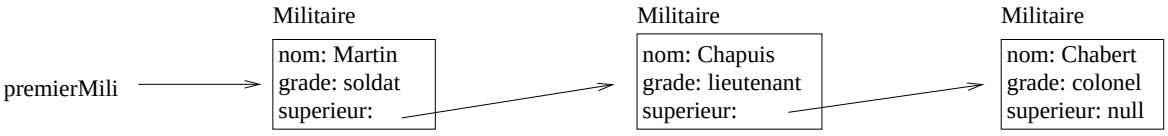
8.3 Parcours d'une structure récursive

Dans notre premier exemple de structure, il y a une variable par objet créé. Mais généralement, ce n'est pas la cas, on a seulement une variable sur le premier élément de la structure et on peut atteindre les autres éléments via un chemin d'accès traversant plusieurs objets.

Voyons un exemple concret de programme et de structure où il y a une seule variable qui contient le premier objet de la structure.

```
class TestMili2{
    public static void main(String[] args){
        Militaire premierMili;
        premierMili=new Militaire("Chabert","Colonel",null);
        premierMili=new Militaire("Chapuis","lieutenant",premierMili);
        premierMili=new Militaire("Martin","soldat",premierMili);
    }
}
```

La structure construite peut se représenter comme suit.



Dans cette structure, on peut accéder au premier objet par le chemin `premierMili`, au second objet par `premierMili.superieur` et au troisième objet par le chemin `premierMili.superieur.superi`

Une structure récursive contient indirectement, via des chaînes de références, plusieurs objets du même type. Beaucoup d'opérations courantes nécessitent de parcourir la structure pour accéder à certains ou à tous les objets de la structure. Ces opérations doivent fonctionner quel que soit le nombre d'objet dans la structure.

Voici quelques exemples classiques :

- compter le nombre d'objets dans la structure (accès à tous les objets). Par exemple : compter le nombre de supérieurs au-dessus d'un militaire donné.
- rechercher si un objet appartient à la structure. Par exemple : untel est-il le chef direct ou indirect d'un militaire donné.
- appeler une méthode sur tous les objets de la structure.

La caractéristique du parcours dans la structure est qu'on ne peut pas accéder directement à chaque élément : il faut passer par des éléments intermédiaires. Par exemple, on ne peut pas accéder directement au chef du chef d'un soldat : on est obligé de passer par le chef du soldat et de regarder quel est son chef.

Pour illustrer les opérations de parcours, nous allons prendre un autre exemple : celui d'un train qui passe dans plusieurs gares. A chaque gare, il y a une heure de passage du train et une référence à la gare suivante. On va utiliser deux classes : une pour le train et une pour les arrêts.

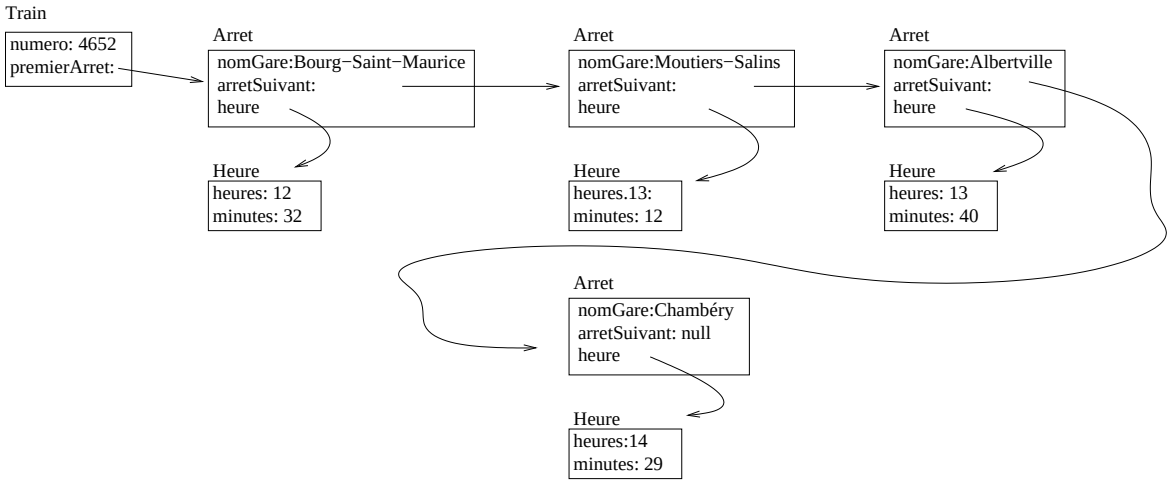
```
class Heure{
    private int minutes, heures;
    public Heure(int h, int m) throws PasHeure{
        if (h<0 || h>23 || m<0 || m>59)
            throw new PasHeure();
        heures=h; minutes=m;
    }
    public String toString(){
        return ""+heures+":"+minutes;
    }

}
class Arret{
    private String nomGare;
    private Heure heure;
    private Arret arretSuivant;
    public Arret(String n, Heure h, Arret as){
        nomGare=n;
        heure=h;
        arretSuivant=as;
    }
    public String toString(){
        return nomGare + "␣" + heure.toString();
    }
    public Arret getArretSuivant(){
        return arretSuivant;
    }
```

```
    public String getNomGare(){
        return nomGare;
    }
    public Heure getHeure(){
        return heure;
    }
}
public class Train{
    private int numero;
    private Arret premierArret;
    public Train(int n, Arret a){
        numero=n;
        premierArret=a;
    }
    public static void main(String[] args) throws PasHeure{
        Arret a1, a2, a3, a4;
        Train t;
        a1=new Arret("Chambery",new Heure(14,29),null);
        a2=new Arret("Albertville",new Heure(13,40),a1);
        a3=new Arret("Moutiers-Salins", new Heure(13,12),a2);
        a4=new Arret("Bourg-Saint-Maurice",new Heure(12,32),a3);
        t=new Train(4652,a4);
    }
}
```

Le train créé dans la méthode main est représenté dans la mémoire de l'ordinateur par cinq objets : un de type train et quatre de type Arret, un pour chaque arrêt du train. Chaque arrêt contient dans la variable arretSuivant une référence à la gare suivante du parcours.

On peut représenter de la façon suivante les objets en mémoire.



Comme exemple de parcours de cette structure, prenons une méthode d'affichage des informations sur le train. Cette méthode doit parcourir tous les arrêts successif et pour chaque arrêt, afficher le nom de la gare et l'heure. Voici comment cela va s'écrire sous la forme d'une méthode de la classe Train.

```
    void afficher(){
        Terminal.ecrireStringln("Train numero " + numero);
        Arret a = premierArret;
        while (a!=null){
```

```
            Terminal.ecrireStringln(a.toString());
            a=a.getArretSuivant();
        }
    }
```

Au premier tour de boucle, la variable a contient l'adresse du premier arrêt. Puis, dans la boucle, on lui affecte la valeur du suivant, c'est-à-dire l'adresse du deuxième arrêt. Et ainsi de suite, jusqu'à la fin. La fin, c'est quand l'arrêt suivant que l'on a affecté à la variable `a` vaut `null`.

Si l'on ajoute à la fin de la méthode main l'instruction `t.afficher();`, cela produit l'affichage suivant :

```
> java Train
Train numero 4652
Bourg-Saint-Maurice 12:32
Moutiers-Salins 13:12
Albertville 13:40
Chambery 14:29
```

Prenons un autre exemple de parcours de tous les éléments de type `Arret` : la méthode qui calcule le nombre de gares où passe un train. C'est encore une méthode de la classe Train.

```
public int nombreDeGares(){
    int res=0;
    Arret a = premierArret;
    while (a!=null){
        res++;
        a=a.getArretSuivant();
    }
    return res;
}
```

Voyons un troisième exemple qui renvoie l'heure de passage d'un train à une gare donnée. C'est encore une méthode de la classe Train. Cette fois, le parcours de la structure s'arrête dès que l'on a trouvé l'objet de type `Arret` qui correspond à la gare recherchée. Le parcours est le même, mais la condition d'arrêt est différente.

```
public Heure heureDePassage(String gare) throws PassePas{
    Arret a = premierArret;
    while (a!=null && !gare.equals(a.getNomGare())){
        a=a.getArretSuivant();
    }
    if (a==null)
        throw new PassePas();
    else
        return a.getHeure();
}
```

Un autre type de parcours consiste à modifier chaque élément de la structure et non seulement à les consulter comme on l'a fait jusqu'ici. Nous vous proposons une méthode qui applique un retard donné à toutes les gares du parcours. Pour cela, nous ajoutons une méthode ajouterDelai à la classe Heure et nous écrivons la classe enregistrerRetard dans la classe Train.

```
// classe Heure
    public void ajouterDelai(int heuresenplus, int minutesenplus){
        heures = (heures + heuresenplus+((minutes+minutesenplus)/60)) %24;
        minutes = (minutes + minutesenplus) % 60; // modulo 60
    }
// classe Train
    public void enregistrerRetard(int heuresenplus, int minutesenplus){
        Arret a = premierArret;
        while (a!=null){
            a.getHeure().ajouterDelai(heuresenplus,minutesenplus);
            a=a.getArretSuivant();
        }
    }
```

8.4 Opérations de création de structures récursives

Dans notre exemple, nous avons construit la structure au moyen du constructeur de la classe `Arret` et d'affectation à des variables. Il est souvent plus pratique de construire la structure au moyen d'une ou plusieurs méthodes. Dans cet exemple, on va mettre ces méthodes dans la classe `Train` et elles créeront de nouvelles instances de `Arret` qu'elle relieront de façon cohérente avec les arrêts du train. Ce qui diffère entre ces méthodes, c'est l'endroit où elles insèrent le nouvel arrêt.

La première méthode insère un arrêt en début de parcours. La seconde en fin de parcours. La dernière, la plus sophisitquée, l'insère à la seule place qui respecte l'ordre des heures de passage.

Pour insérer un nouvel arrêt en début de trajet, il faut créer un nouvel arrêt dont l'arrêt suivant sera le premier des arrêt existants auparavant. Puis modifier le premier arrêt du train pour qu'il réfère à ce nouvel élément. Les paramètres à donner à la méthode sont le nom de la gare et l'heure de passage.

```
    // dans la classe Train
    public void ajouterArretDebut(String ng, Heure h){
        Arret nouveau = new Arret(ng,h,premierArret);
        premierArret = nouveau;
    }
```

Pour construire le train de Bourg-Saint-Maurice, il faut écrire le code suivant :

```
        Train t1 = new Train(4652,null);
        t1.ajouterArretDebut("Chambery",new Heure(14,29));
        t1.ajouterArretDebut("Albertville",new Heure(13,40));
        t1.ajouterArretDebut("Moutiers-Salins", new Heure(13,12));
        t1.ajouterArretDebut("Bourg-Saint-Maurice",new Heure(12,32));
```

Ajouter un élément à la fin du trajet est plus difficile pour deux raisons. D'abord, il faut traiter à part le cas d'un train qui n'a aucun arrêt : dans ce cas, le dernier est aussi le premier et il faut modifier la valeur de la variable `premierArret` d'un objet de type `Train`. Dans les autres cas, il faut parcourir tous les objets du type `Arret`, du premier jusqu'au dernier. C'est celui-là dont il faut modifier la variable `arretSuivant`. Pour ce faire, nous allons ajouter une méthode `setArretSuivant` dans la classe `Arret`.

```
// dans la classe Arret
    public void setArretSuivant(Arret arr){
```

```
        arretSuivant=arr;
    }
// dans la classe Train
    public void ajouterALaFin(String ng, Heure h){
        Arret nouveau = new Arret(ng,h,null);
        if (premierArret==null)
            premierArret = nouveau;
        else{
            Arret arr=premierArret;
            while (arr.getArretSuivant()!=null)
                arr=arr.getArretSuivant();
            arr.setArretSuivant(nouveau);
        }
    }
```

8.5 Opérations de mise à jour d'une structure récursive

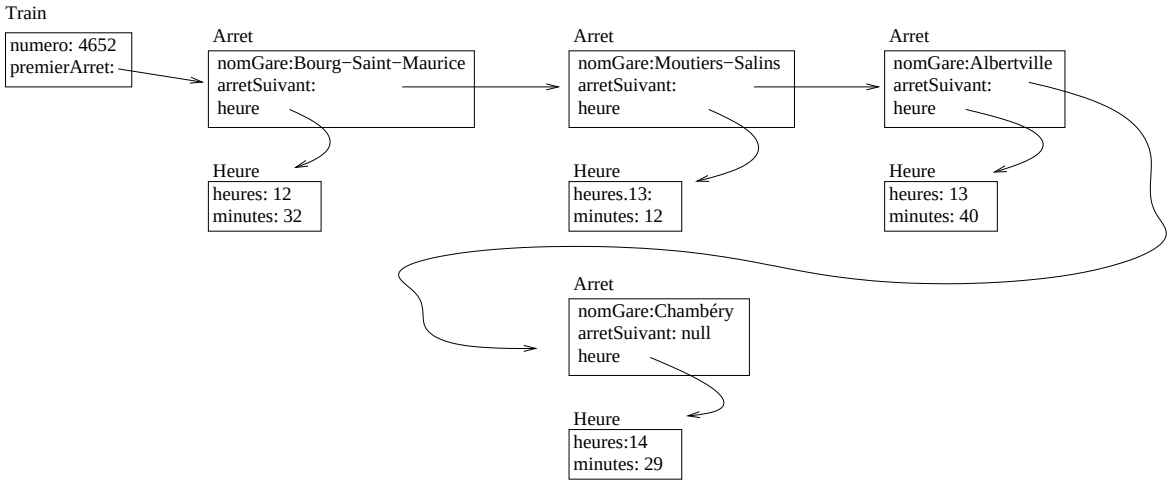
A la section précédente, nous avons vu comment ajouter des éléments à une structure récursive. Cela permet de créer la structure mais également de la faire évoluer au fil des changements (par exemple, s'il y a un nouvel arrêt ajouté à un train existant).

D'autres opérations peuvent servir à maintenir à jour une stucture : la suppression d'un élément, la modification du contenu d'un ou plusieurs éléments, la modification de l'ordre des éléments. Dans notre exemple, supprimer une gare sur un train, changer l'heure d'arrêt ou le nom d'une gare. En revanche, changer l'ordre des arrêts n'a a priori pas de sens pour les trains.

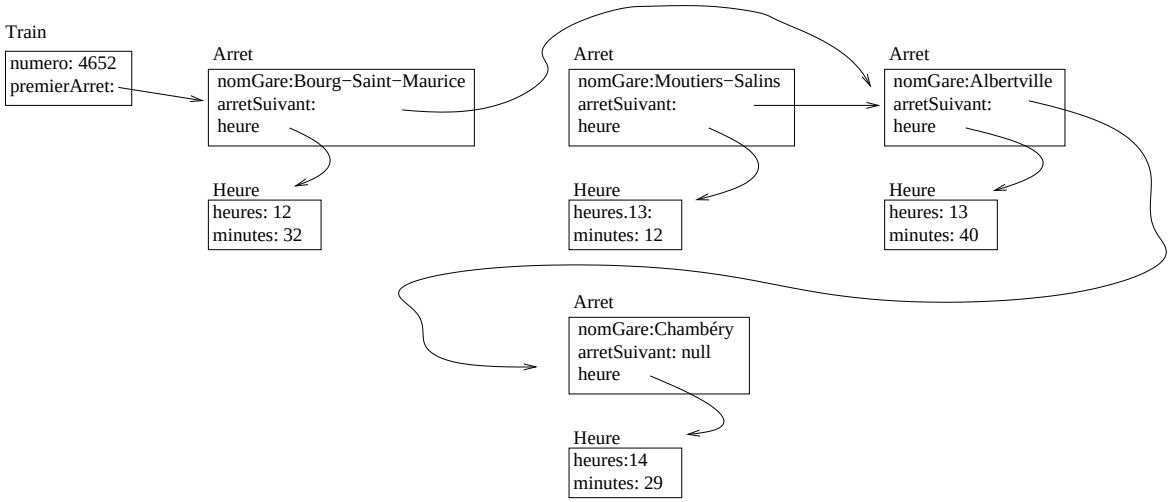
Voyons en détail l'opération de suppression d'un élément. Cette opération est relativement complexe, parce que la suppression d'un élément suppose de changer **l'élément qui le précède**. En effet, c'est la variable arretSuivant qui est à changer.

Prenons un exemple complet : pour supprimer l'arrêt à Moutiers-Salins, il faut modifier l'arrêt Bourg-Saint-Maurice pour que sa variable `arretSuivant` contienne une référence à l'arrêt Albertville à la place d ela référence à l'arrêt supprimé.

La situation avant la suppression est la suivante :



Après la suppression, les objets sont comme ceci :



On voit que cette opération enlève un objet de type `Arret` de la structure (il n'y a plus de chemin d'accès à l'objet supprimé), mais cela ne supprime pas immédiatement l'objet de la mémoire. En java, ce n'est pas le programme qui supprime les objets, c'est l'environnement d'exécution (runtime, machine virtuelle java) qui le fait automatiquement.

Voyons maintenant comment programmer l'opération de suppression. Comme souvent, il faut gérer à part le cas où l'élément à enlever est le premier de la structure, car dans ce cas, c'est la variable `premierArret` d'un objet de type `Train` qui est à changer. Dans les autres cas, c'est la variable `arretSuivant` d'un objet de type `Arret`.

Ensuite, il faut faire un parcours où l'on cherche l'élément précédant celui qui a une certaine caractéristique. Ici, on va utiliser le nom de la gare comment identification de l'élément à supprimer.

```
// dans la classe Train
    public void supprimerArret(String ng) throws PassePas{
        if (premierArret == null)
            throw new PassePas();
        if (premierArret.getNomGare().equals(ng))
            premierArret = premierArret.getArretSuivant();
        else{
            Arret precedent = premierArret;
            while (precedent.getArretSuivant()!=null &&
              !precedent.getArretSuivant().getNomGare().equals(ng)){
                precedent = precedent.getArretSuivant();
            }
            if (precedent.getArretSuivant()==null)
                throw new PassePas();
            precedent.setArretSuivant(
                        precedent.getArretSuivant().getArretSuivant());
        }
    }
\end{lslisting}


\section{Récursivité des structures de données et des méthodes}

Les méthodes récursives sont particulièrement adaptées au parcours des
structures récursives. Jusqu'ici, nous avons fait des parcours au moyen d'une
```

boucle **while**. On peut remplacer cette boucle par un appel récursif équivalent.

Nous allons illustrer cela en écrivant des méthodes récursives pour la classe
\texttt{Arret}.

```
\begin{lstlisting}
    public void afficher(){
        Terminal.ecrireStringln(this.toString());
        if (arretSuivant != null)
            arretSuivant.afficher();
    }
    public int nombreDeGares(){
        if (arretSuivant == null)
            return 1;
        else
            return 1+arretSuivant.nombreDeGares();
    }
```

Ces méthodes sont à comparer avec les méthodes itératives que nous avions écrites dans la classe `Train` :

```
    void afficher(){
        Terminal.ecrireStringln("Train numero " + numero);
        Arret a = premierArret;
        while (a!=null){
            Terminal.ecrireStringln(a.toString());
            a=a.getArretSuivant();
        }
    }
    public int nombreDeGares(){
        int res=0;
        Arret a = premierArret;
        while (a!=null){
            res++;
            a=a.getArretSuivant();
        }
        return res;
    }
```

8.6 Récursivité et héritage

Dans les exemples que nous avons vus, tous les objets de la structure récursive sont du même type. En utilisant l'héritage ou les interfaces, on peut créer des structures comportant des objets de différents types mais ayant en commun la variable d'instance référençant l'élément suivant et une méthode de parcours de la structure (cf. méthode `getArretSuivant` dans l'exemple des trains).

Nous allons reprendre l'exemple du train en utilisant l'héritage pour distinguer deux arrêts particuliers : le premier arrêt, la gare de départ, qui contiendra également les informations que nous mettions jusqu'ici dans la classe `Train` et le terminus qui n'a pas d'arrêt suivant. On va raffiner l'exemple en distinguant heure d'arrivée et heure de départ.

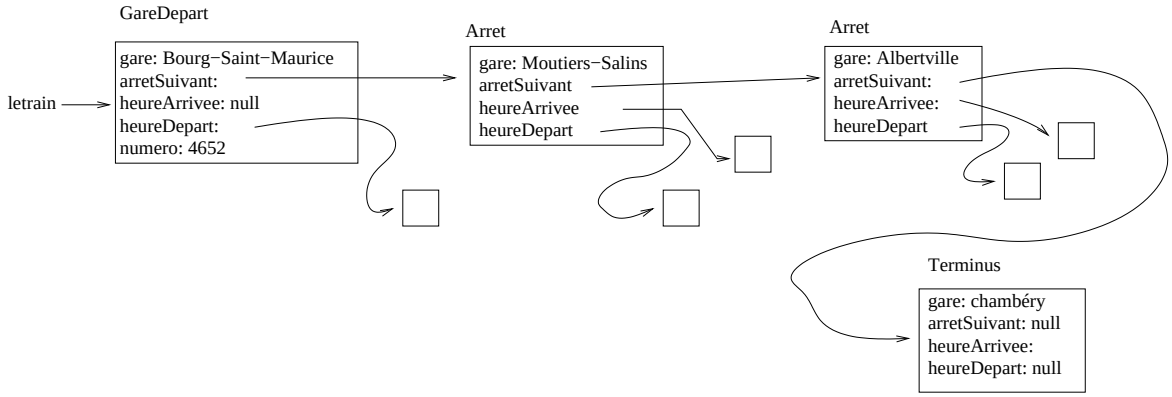
```
class Arret{
```

```
    protected String nomGare;
    protected Heure heureArrivee;
    protected Heure heureDepart;
    protected Arret arretSuivant;
    Arret(String n, Heure ha, Heure hd, Arret as){
        nomGare=n;
        heureArrivee=ha;
        heureDepart=hd;
        arretSuivant=as;
    }
    public String toString(){
        return "␣␣" + nomGare + "␣" + heureArrivee.toString() + "␣" +
            heureDepart.toString();
    }
    public Arret getArretSuivant(){
        return arretSuivant;
    }
    public String getNomGare(){
        return nomGare;
    }
    public Heure getHeureArrivee(){
        return heureArrivee;
    }
    public Heure getHeureDepart(){
        return heureDepart;
    }
    public void setArretSuivant(Arret arr){
        arretSuivant=arr;
    }
    public void setHeureArrivee(Heure h){
        heureArrivee=h;
    }
    public void setHeureDepart(Heure h){
        heureDepart=h;
    }
    public void afficher(){
        Terminal.ecrireStringln(this.toString());
        if (arretSuivant != null)
            arretSuivant.afficher();
    }
}
class GareDepart extends Arret{
    protected int numero;
    protected Arret premierArret;
    GareDepart(String s, int n, Heure heureDepart, Arret a){
        super(s, null, heureDepart,a);
        numero=n;
    }
    public String toString(){
        String res;
        res="Train␣numero␣" + numero;
        res=res + "␣" + nomGare + "␣" + heureDepart.toString();
        return res;
```

```
    }
    public int nombreDeGares(){
        int res=0;
        Arret a = this;
        while (a!=null){
            res++;
            a=a.getArretSuivant();
        }
        return res;
    }
    public Heure heureDePassage(String gare) throws PassePas{
        Arret a = this;
        while (a!=null && !gare.equals(a.getNomGare())){
            a=a.getArretSuivant();
        }
        if (a==null)
            throw new PassePas();
        else
            return a.getHeureArrivee();
    }
}
class Terminus extends Arret{
    Terminus(String n, Heure heureArr){
        super(n,heureArr,null,null);
    }
    public String toString(){
        return "  " + nomGare + " " + heureArrivee.toString();
    }
}
public class Train3{
    public static void main(String[] args) throws PasHeure, PassePas{
        Arret trainprov;
        GareDepart letrain;
        trainprov=new Terminus("Chambery",new Heure(14,29));
        trainprov=new Arret("Albertville",new Heure(13,40),
                            new Heure(13,42),trainprov);
        trainprov=new Arret("Moutiers-Salins", new Heure(13,12),
                            new Heure(13,15),trainprov);
        letrain=new GareDepart("Bourg-Saint-Maurice",4652,
                                new Heure(12,32),trainprov);
        letrain.afficher();
        Terminal.ecrireStringln(letrain.nombreDeGares()+" gares");
        Terminal.ecrireStringln(" arret a Moutiers: " +
                                letrain.heureDePassage("Moutiers-Salins"));
    }
}
class PassePas extends Exception{}
```

La structure construire comporte un objet `GareDepart`, deux objets `Arret` et un objet `Terminus`, tous reliés dans une même structure, à l'aide des variables `arretSuivant`.

Cette structure se représente comme suit (pour simplifier le dessin, les heures sont représentées par un carré vide).



8.7 Récursivité indirecte

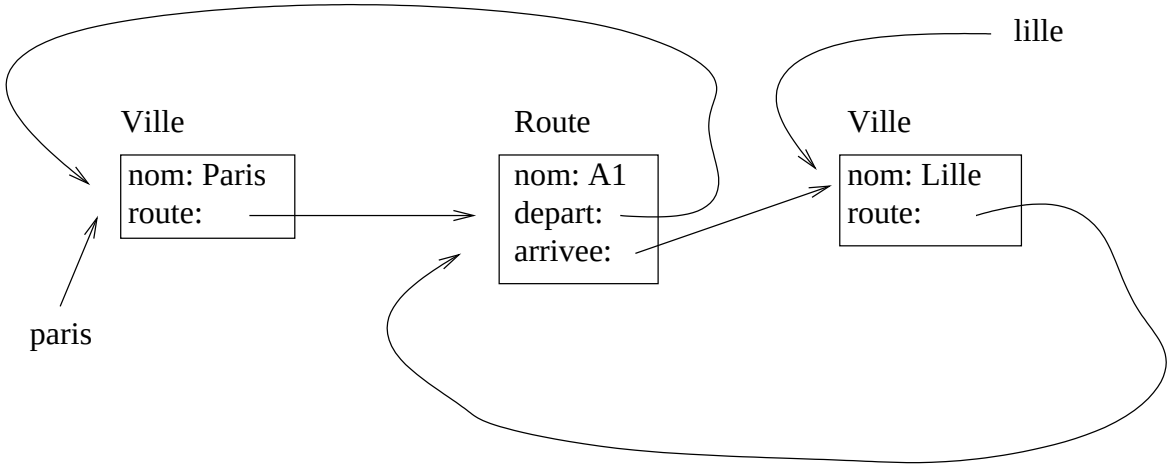
Il se peut que la récursivité ne se traduise pas directement par une variable ayant le type de la classe elle-même, mais qu'elle fasse intervenir un chemin incluant des objets intermédiaires appartenant à d'autres classes. On dit qu'il existe une récursivité indirecte si depuis un objet d'une certaine classe C, il existe un ou plusieurs chemins aboutissant à un autre objet de cette classe C.

Prenons un exemple où des villes sont reliées par des routes. Villes et routes sont des objets. Chaque ville a des routes pour entrer ou sortir. On peut aller d'une ville à une autre, mais seulement en passant par une route (ou en passant par des villes intermédiaires et plusieurs routes). Il n'y a pas de variable de type `Ville` dans la classe `Ville`, mais un chemin d'accès existe pour atteindre une autre ville.

```
public class Ville{
    String nom;
    Route route;
    Ville(String n){
        nom=n;
    }
    void setRoute(Route r){
        route=r;
    }
}
class Route{
    String nom;
    Ville depart, arrivee;
    Route(String n, Ville d, Ville a){
        nom=n;
        depart=d;
        arrivee=a;
        depart.setRoute(this);
        arrivee.setRoute(this);
    }
}
class Test{
    public static void main(String[] a){
        Ville paris, lille;
        Route a1;
        paris = new Ville("Paris");
```

```
        lille = new Ville("Lille");
        a1=new Route("A1",paris,lille);
    }
}
```

Dans cette ébauche, chaque ville n'a qu'une ville d'accès : ce n'est pas très satisfaisant. Mais nous cherchons l'illustration la plus simple possible de la récursivité indirecte. La méthode `Main` construit une structure de donnée que l'on peut représenter comme suit.



On voit qu'il existe un chemin d'accès d'un objet `Ville` à l'autre. Il en existe même deux dans chaque ville. Par exemple, pour l'objet référencé par la variable `paris`, les deux chemins sont `paris.route.depart` et `paris.route.arrivee`. Ces deux chemins mènent aux deux objets de type `Ville` qui ont été créés, c'est à dire que le premier revient sur `paris` et le second va vers `lille`.

Cet exemple peut être développé dans un sens plus réaliste avec plusieurs routes partant d'une ville et plusieurs villes sur une route. On peut, par exemple, avoir un ArrayList de routes dans la classe ville et un ArrayList de villes dans la classe route. Cela permet de représenter des réseaux routiers complexes dans lesquels il existe une multiplicité de chemins (au sens Java) entre objets de la classe `Ville`.