TP n° 4
Licence Informatique (L2)
« Programmation objet avancée »

F. BERTRAND

Année universitaire 2020–2021

Éléments de correction

1 Hiérarchie d'animaux

L'objectif de cet exercice est de montrer l'utilisation d'une hiérarchie d'interfaces représentant un sorte d'héritage multiple.

À partir de l'archive `zoo1.zip`, réaliser les étapes suivantes **en conservant, pour chaque étape, les classes et les interfaces développées** :

1. en utilisant des classes (pas d'interfaces), créer une hiérarchie permettant de stocker dans un tableau une instance de chacune des classes présentes dans l'archive. Le programme devra afficher :
   — le nombre de reptiles ;
   — le nombre d'animaux terrestres.

Éléments de correction

La hiérarchie idéale nécessiterait l'utilisation de l'héritage multiple (cf. Fig. 1) :

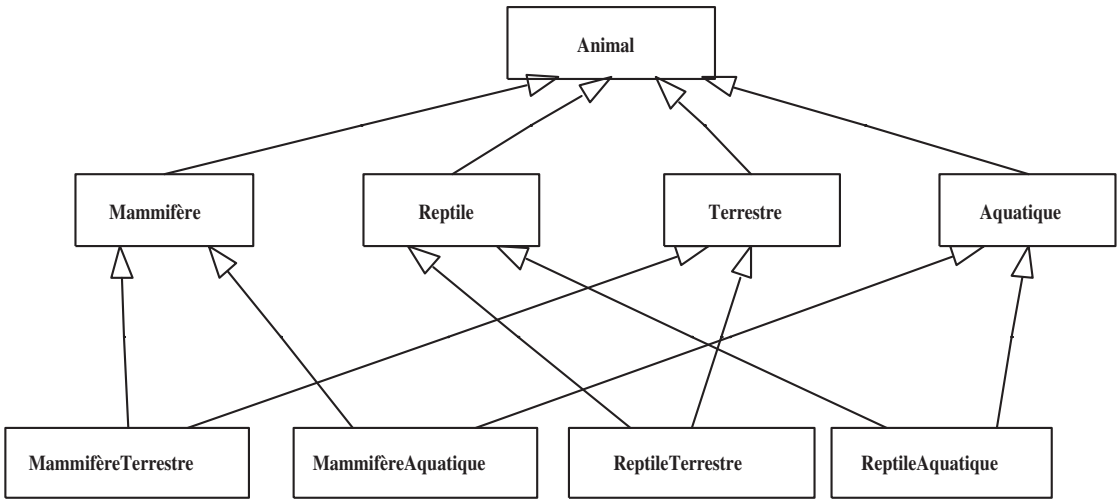

FIGURE 1 – Hiérarchie d'héritage multiple impossible à implémenter en Java

En Java, sans héritage multiple, deux solutions sont possibles. Soit on choisit comme premier critère pour le classification, la nature de l'animal (mammifère ou reptile) comme illustré sur la figure 2.

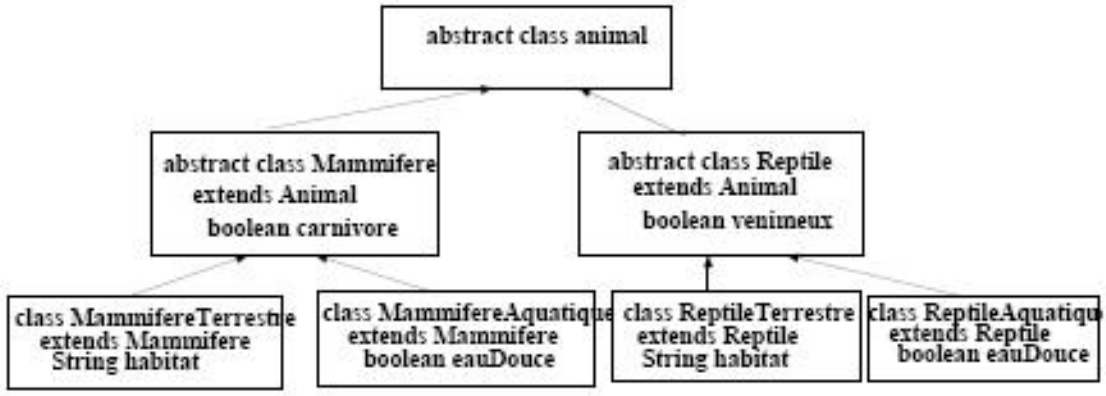

FIGURE 2 – Hiérarchie d'héritage simple avec `Mammifere` et `Reptile` comme super-classes

Soit on classe sur le milieu dans lequel évolue l'animal (terrestre ou aquatique) comme illustré sur la figure 3.

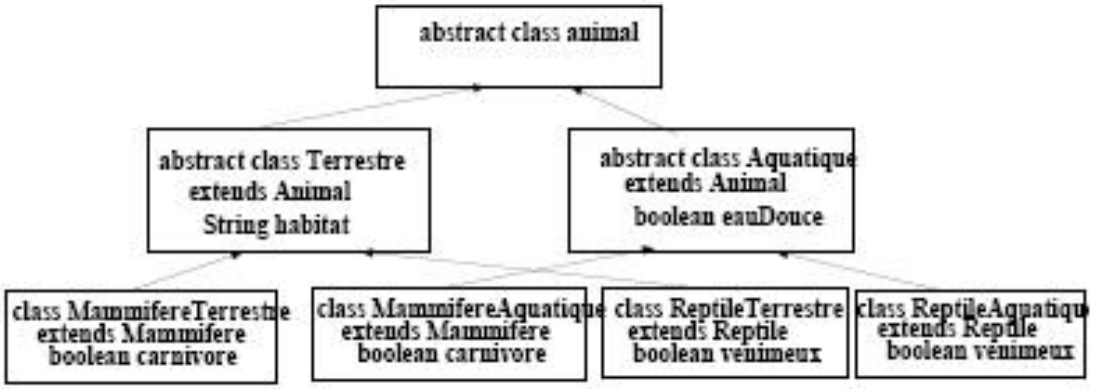

FIGURE 3 – Hiérarchie d'héritage simple avec `Aquatique` et `Terrestre` comme super-classes

La solution donnée ci-dessous s'appuie sur la figure 2.

```
1
2  public abstract class Animal
3  {
4     private String nomEspece;
5
6     public Animal(String nom)
7     {
8        this.nomEspece = nom;
9     }
10 }
```

```
1
2  public abstract class Mammifere extends Animal
3  {
4     private boolean carnivore;
5
6     public Mammifere(String nom, boolean carnivore)
7     {
8        super(nom);
9        this.carnivore = carnivore;
10    }
11 }
```

```
1  public abstract class Reptile extends Animal
2  {
3     private boolean venimeux;
4
5     public Reptile(String nom, boolean venimeux)
6     {
7        super(nom);
8        this.venimeux = venimeux;
9     }
10 }
```

```
1  public class MammifereAquatique extends Mammifere
2  {
3     private boolean eauDouce;
4
5     public MammifereAquatique(String nom, boolean carnivore, boolean eauDouce)
6     {
7        super(nom, carnivore);
8        this.eauDouce = eauDouce;
9     }
10 }
```

```
1  public class MammifereTerrestre extends Mammifere
2  {
3     private String habitat;
4
5     public MammifereTerrestre(String nom, boolean carnivore, String habitat)
6     {
7        super(nom, carnivore);
8        this.habitat = habitat;
9     }
10 }
```

```
1  public class ReptileAquatique extends Reptile
2  {
3     private boolean eauDouce;
4
5     public ReptileAquatique(String nom, boolean venimeux, boolean eauDouce)
6     {
7        super(nom, venimeux);
8        this.eauDouce = eauDouce;
9     }
10 }
```

```
1  public class ReptileTerrestre extends Reptile
2  {
3     private String habitat;
4
5     public ReptileTerrestre(String nom, boolean venimeux, String habitat)
6     {
7        super(nom, venimeux);
8        this.habitat = habitat;
9     }
10 }
```

```
1
2  public class Main
3  {
4     public static void main(String[] args)
5     {
6        Animal zoo[] = {new MammifereTerrestre("Lion", true, "savane"),
7           new MammifereAquatique("Baleine", false, false),
8           new ReptileTerrestre("vipere", true, "campagne"),
9           new ReptileAquatique("Crocodile", false, true)
10       };
11       // le comptage des animaux terrestres est facile car la hierarchie de
12       // classes est  bien adpatee...
13       int nbReptiles = 0;
14       for (Animal a : zoo)
15       {
16          if (a instanceof Reptile)
17          {
18             nbReptiles++;
19          }
20       }
21       System.out.println("Nombre de reptiles = " + nbReptiles);
22
23       // le comptage des animaux terrestres est moins facile car cela necessite
24       // d'effectuer un double test sur le type des instances...
25       int nbTerrestre = 0;
26       for (Animal a : zoo)
27       {
28          if ((a instanceof MammifereTerrestre)||(a instanceof ReptileTerrestre))
29          {
30             nbTerrestre++;
31          }
32       }
33       System.out.println("Nombre d'animaux terrestres = " + nbTerrestre);
34    }
35 }
```

2. Ajouter sous forme d'interface les classes que vous n'avez pas pu définir sous forme de classes abstraites et modifiez vos classes en conséquence.

Éléments de correction

Cela donne la hiérarchie suivante (cf. Fig. 4) :

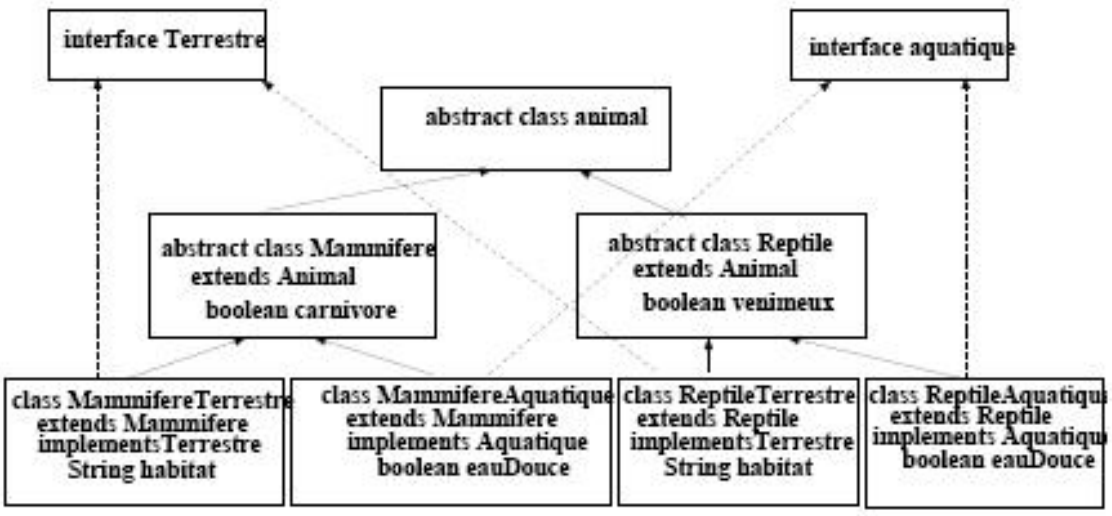

FIGURE 4 – Hiérarchie d'héritage avec des interfaces

et les classes suivantes modifiées par rapport à la question précédente :

```

public interface Aquatique
{
}
```

```

public interface Terrestre
{
}
```

```

public class MammifereAquatique
  extends Mammifere
  implements Aquatique
{
   private boolean eauDouce;

   public MammifereAquatique(String nom, boolean carnivore, boolean eauDouce)
   {
      super(nom, carnivore);
      this.eauDouce = eauDouce;
   }
}
```

```
public class MammifereTerrestre
  extends Mammifere
  implements Terrestre
{
   private String habitat;

   public MammifereTerrestre(String nom, boolean carnivore, String habitat)
   {
      super(nom, carnivore);
      this.habitat = habitat;
   }
}
```

```
public class ReptileAquatique
  extends Reptile
  implements Aquatique
{
   private boolean eauDouce;

   public ReptileAquatique(String nom, boolean venimeux, boolean eauDouce)
   {
      super(nom, venimeux);
      this.eauDouce = eauDouce;
   }
}
```

```
public class ReptileTerrestre
  extends Reptile
  implements Terrestre
{
   private String habitat;

   public ReptileTerrestre(String nom, boolean venimeux, String habitat)
   {
      super(nom, venimeux);
      this.habitat = habitat;
   }
}
```

```

public class Main
{
   public static void main(String[] args)
   {
      Animal zoo[] = {new MammifereTerrestre("Lion", true, "savane"),
         new MammifereAquatique("Baleine", false, false),
         new ReptileTerrestre("vipere", true, "campagne"),
         new ReptileAquatique("Crocodile", false, true)
      };
      // le comptage des animaux terrestres est inchange par rapport
      // a la version precedente
      int nbReptiles = 0;
      for (Animal a : zoo)
```

```
15          {
16              if (a instanceof Reptile)
17              {
18                  nbReptiles++;
19              }
20          }
21          System.out.println("Nombre de reptiles = " + nbReptiles);
22
23          // le comptage des animaux terrestres est plus facile avec la
24          // presence de l'interface Terrestre...
25          int nbTerrestre = 0;
26          for (Animal a : zoo)
27          {
28              if (a instanceof Terrestre)
29              {
30                  nbTerrestre++;
31              }
32          }
33          System.out.println("Nombre d'animaux terrestres = " + nbTerrestre);
34      }
35  }
```

3. Les deux interfaces créées dans la question précédente n'apportent aucune contrainte aux classes qui les implémentent. D'autre part, nous avons privilégié certains types qui demeurent des classes (ex. `Mammifere`) et d'autres qui ont été créés sous forme d'interfaces (ex. `Aquatique`).

   Dans un souci d'unification proposer une solution où les types `Animal`, `Mammifere`, `Reptile`, `Aquatique` et `Terrestre` seront des interfaces et déclareront une méthode de test sur la propriété qu'il possède (ex. `Animal` imposera qu'on définisse une méthode `String donneNom()`, `Mammifere` une méthode `boolean estCarnivore()`.

Éléments de correction

Cela donne la hiérarchie suivante (cf. 4) :

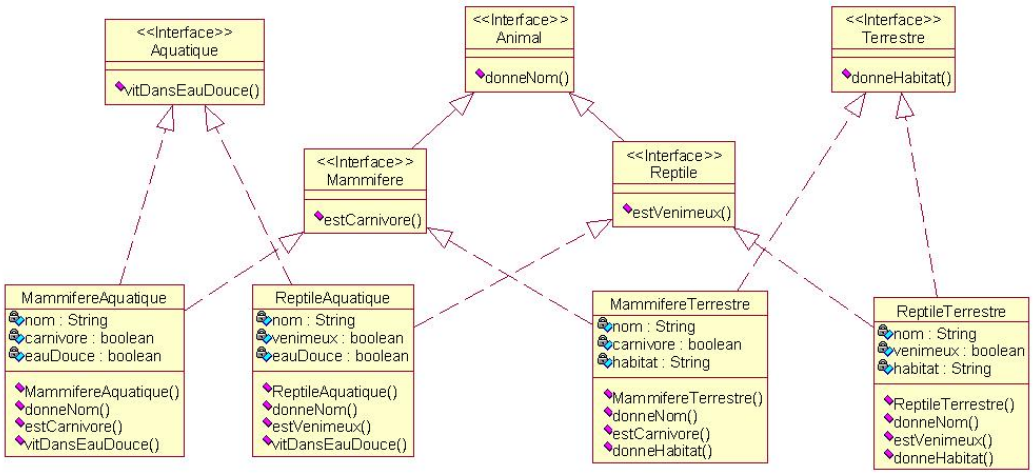

FIGURE 5 – Hiérarchie d'héritage avec des interfaces

et les classes suivantes modifiées (la classe `Main` demeure inchangée) :

```
1
2   public interface Animal
3   {
4       public String donneNom();
5   }
```

```
1
2  public interface Aquatique
3  {
4     public boolean vitDansEauDouce();
5  }
```

```
1
2  public interface Terrestre
3  {
4     public String donneHabitat();
5  }
```

```
1
2  public interface Mammifere extends Animal
3  {
4     public boolean estCarnivore();
5  }
```

```
1  public interface Reptile extends Animal
2  {
3     public boolean estVenimeux();
4  }
```

```
1
2  public class MammifereAquatique implements Mammifere, Aquatique
3  {
4     private String nom;
5     private boolean carnivore;
6     private boolean eauDouce;
7
8     public MammifereAquatique(String nom, boolean carnivore, boolean eauDouce)
9     {
10       this.nom = nom;
11       this.carnivore = carnivore;
12       this.eauDouce = eauDouce;
13    }
14
15    public String donneNom()
16    {
17       return this.nom;
18    }
19
20    public boolean estCarnivore()
21    {
22       return this.carnivore;
23    }
24
25    public boolean vitDansEauDouce()
26    {
27       return this.eauDouce;
28    }
29 }
```

```
1  public class MammifereTerrestre implements Mammifere, Terrestre
2  {
3     private String nom;
4     private boolean carnivore;
5     private String habitat;
6
7     public MammifereTerrestre(String nom, boolean carnivore, String habitat)
8     {
9        this.nom = nom;
10       this.carnivore = carnivore;
11       this.habitat = habitat;
12    }
13
14    public String donneNom()
15    {
16       return this.nom;
17    }
18
19    public boolean estCarnivore()
20    {
21       return this.carnivore;
22    }
23
24    public String donneHabitat()
```

```
    {
        return this.habitat;
    }

}
```

```
public class ReptileAquatique implements Reptile, Aquatique
{
    private String nom;
    private boolean venimeux;
    private boolean eauDouce;

    public ReptileAquatique(String nom, boolean venimeux, boolean eauDouce)
    {
        this.nom = nom;
        this.venimeux = venimeux;
        this.eauDouce = eauDouce;
    }

    public String donneNom()
    {
        return this.nom;
    }

    public boolean estVenimeux()
    {
        return this.venimeux;
    }

    public boolean vitDansEauDouce()
    {
        return this.eauDouce;
    }

}
```

```
public class ReptileTerrestre implements Reptile, Terrestre
{
    private String nom;
    private boolean venimeux;
    private String habitat;

    public ReptileTerrestre(String nom, boolean venimeux, String habitat)
    {
        this.nom = nom;
        this.venimeux = venimeux;
        this.habitat = habitat;
    }

    public String donneNom()
    {
        return this.nom;
    }

    public boolean estVenimeux()
    {
        return this.venimeux;
    }

    public String donneHabitat()
    {
        return this.habitat;
    }

}
```

2 Utilisation d'une interface pour le tri d'éléments

On souhaite pouvoir trier une liste de comptes bancaires (cf. les deux classes fournies `CompteBancaireInitial` et `Client` sur Moodle) en utilisant la méthode `sort()` de la classe `java.util.Collections`. Cette méthode, pour réaliser son tri, s'appuie sur la comparaison d'élements deux par deux définie par une méthode `compareTo()`. Pour être certain que les objets à comparer appartiennent à une classe ayant implémenté cette méthode `compareTo()`, le paramètre de `sort()` est du type `Comparable`, interface déclarant une méthode `compareTo()`. Ainsi la classe des objets à trier doit implémenter cette interface et donc définir une méthode `compareTo()`.

Travail à réaliser :

1. Modifier la classe `CompteBancaire` pour qu'elle implémente l'interface `Comparable`. La méthode `compareTo()` qui sera définie utilisera le numéro de compte comme critère de comparaison. À l'aide d'un programme de test, créer plusieurs comptes et vérifier que la méthode `sort()` trie correctement une liste de comptes passée en paramètre.
2. Puis proposer une solution plus générique permettant l'utilisation de différents critères de tri (ex. nom du détenteur, montant du solde, ...) en définissant trois classes implémentant l'interface `Comparator<CompteBancaire>` :
   — `ComparateurNumeroCompte`;
   — `ComparateurNomDetenteur`;
   — `ComparateurMontantSolde`.
   (la méthode `equals` déclarée dans l'interface n'aura pas à être implémentée car nous bénéficions d'une version par défaut héritée de la classe `Object`).

Éléments de correction

Pour la première question, il faut implémenter l'interface `Comparable` et donc créer une méthode `compareTo` :

```
1
2  public class CompteBancaire implements Comparable<CompteBancaire>
3  {
4      private double solde;
5      private int numero;
6      private Client detenteur;
7
8      // constructeur permettant d'initialiser le solde
9      // du compte, le numero et le proprietaire du compte
10     public CompteBancaire(double soldeInitial, int numero, Client c)
11     {
12         this.solde = soldeInitial;
13         this.numero = numero;
14         this.detenteur = c;
15     }
16
17     public int donneNumero()
18     {
19         return this.numero;
20     }
21
22     public double donneSolde()
23     {
24         return this.solde;
25     }
26
27     public Client donneDetenteur()
28     {
29         return this.detenteur;
30     }
31
32     // credite un compte d'un montant donne
33     public void crediter(double montant)
34     {
```

```
        this.solde = this.solde + montant;
    }

    // debite un compte d'un montant donne
    public double debiter(double montant)
    {
        this.solde = this.solde - montant;
        return montant;
    }

    // donne le solde du compte
    public double consulter()
    {
        return this.solde;
    }

    // transfert un montant du courant courant vers le compte de destination
    public void transferer(CompteBancaire unCompte, int montantATransferer)
    {
        this.solde = this.solde - montantATransferer;
        unCompte.solde = unCompte.solde + montantATransferer;
    }

    // comparaison
    public int compareTo(CompteBancaire c)
    {
        if (this.numero > c.numero)
        {
            return 1;
        }
        else if (this.numero < c.numero)
        {
            return -1;
        }
        else
        {
            return 0;
        }
    }

    // affichage
    public String toString()
    {
        return this.numero + " " + this.solde + " " + this.detenteur;
    }
}
```

Pour la seconde question, un peu plus difficile conceptuellement mais pas algorithmiquement, on va créer et utiliser des objets-fonctions qui seront passés à la version surchargée de la méthode `sort` *:*

```
import java.util.Comparator;

public class ComparateurNumeroCompte extends Object
        implements Comparator<CompteBancaire>
{
    public int compare(CompteBancaire c1, CompteBancaire c2)
    {
        if (c1.donneNumero() > c2.donneNumero())
        {
            return 1;
        }
        else if (c1.donneNumero() < c2.donneNumero())
        {
            return -1;
        }
        else
        {
            return 0;
        }
    }
}
```

```
import java.util.Comparator;

public class ComparateurMontantSolde extends Object
        implements Comparator<CompteBancaire>
```

```
{
    public int compare(CompteBancaire c1, CompteBancaire c2)
    {
        if (c1.donneSolde() > c2.donneSolde())
        {
            return 1;
        }
        else if (c1.donneSolde() < c2.donneSolde())
        {
            return -1;
        }
        else
        {
            return 0;
        }
    }
}
```

Pour le dernier critère, le nom du détenteur du compte, on utilise le fait que la classe `String` implémente l'interface `Comparable` et possède ainsi une méthode `compareTo` :

```
import java.util.Comparator;

public class ComparateurNomDetenteur extends Object
        implements Comparator<CompteBancaire>
{
    public int compare(CompteBancaire c1, CompteBancaire c2)
    {
        return (c1.donneDetenteur().donneNom().compareTo(c2.donneDetenteur().donneNom()));
    }
}
```

La classe de test :

```

import java.util.ArrayList;
import java.util.Collections;
import java.util.List;

public class Main
{
    public static void main(String[] args)
    {
        List<CompteBancaire> liste = new ArrayList<CompteBancaire>();
        liste.add(new CompteBancaire(10, 4556, new Client("Durand", "Marcel")));
        liste.add(new CompteBancaire(200, 1112, new Client("Martin", "Paul")));
        liste.add(new CompteBancaire(450, 9980, new Client("Dupont", "Georges")));

        System.out.println(liste);
        // tri par defaut
        Collections.sort(liste);
        System.out.println(liste);
        // tri selon le numero de compte (par defaut)
        Collections.sort(liste, new ComparateurNumeroCompte());
        System.out.println(liste);
        // tri selon le solde
        Collections.sort(liste, new ComparateurMontantSolde());
        System.out.println(liste);
        // tri selon le nom du detenteur du compte
        Collections.sort(liste, new ComparateurNomDetenteur());
        System.out.println(liste);
    }
}
```

3 Clonage d'objets

Soient les deux classes `EnteteCommande` et `Commande` déclarées dans `Commande.java` (cf. code fourni) dont le code est présenté ci-dessous :

```
1  package clonage_pb;
2
3  public class EnteteCommande {
4      // nom du client qui a passe la commande
5      private String nomClient;
6      // numero de la commande
7      private int numeroCommande;
8
9      public EnteteCommande(String nomClient, int numero) {
10         this.nomClient = nomClient;
11         this.numeroCommande = numero;
12     }
13
14     public String donneNomClient() {
15         return this.nomClient;
16     }
17
18     public void changeNomClient(String nomClient) {
19         this.nomClient = nomClient;
20     }
21
22     public int donneNumeroCommande() {
23         return this.numeroCommande;
24     }
25
26     public void changeNumeroCommande(int numeroCommande) {
27         this.numeroCommande = numeroCommande;
28     }
29
30     public String toString() {
31         return ("numero = " + this.numeroCommande
32                 + " effectu'ee par : " + this.nomClient);
33     }
34 }
```

```
1  package clonage_pb;
2
3  /**
4   * Classe representant une commande
5   */
6  public class Commande {
7      // compteur du nombre de commandes creees
8      private static int nombreCommandes = 0;
9      private EnteteCommande entete;
10
11     // le constructeur est inaccessible pour obliger l'utilisateur
12     // a utiliser la methode creerCommande pour creer des commandes
13     private Commande(String nomClient, int numeroCommande) {
14         this.entete = new EnteteCommande(nomClient, numeroCommande);
15     }
16
17     // permet la creation de commande
18     public static Commande creerCommande(String nomClient) {
19         // utilisation de la variable de classe nombreCommandes
20         // pour attribuer un nouveau numéro à chaque commande
21         return new Commande(nomClient, ++nombreCommandes);
22     }
23
24     // permet d'obtenir le nom du client d'une commande
25     public String donneNomClient() {
26         return this.entete.donneNomClient();
27     }
28
29     // permet de modifier le client d'une commande
30     public void changeNomClient(String nomClient) {
31         this.entete.changeNomClient(nomClient);
32     }
33
34     // permet d'obtenir le numero d'une commande
35     public int donneNumeroCommande() {
36         return this.entete.donneNumeroCommande();
37     }
```

```
38
39      // permet la copie de commandes
40      public Object clone() throws CloneNotSupportedException {
41          return super.clone();
42      }
43
44      public String toString() {
45          return ("Commande " + this.entete);
46      }
47  }
```

Et la classe de test `TestClonageCommande` :

```
1   package clonage_pb;
2
3   public class TestClonageCommandeV1 {
4       public static void main(String[] args) throws Exception {
5           // création de deux commandes
6           Commande cmd1 = Commande.creerCommande("Martin");
7           Commande cmd2 = Commande.creerCommande("Durand");
8
9           System.out.println(cmd1);
10          System.out.println(cmd2);
11          // creation de la commande cmd3 par clonage de la commande cmd2
12          Commande cmd3 = (Commande) cmd2.clone();
13          // changement du client initial de cette commande
14          cmd3.changeNomClient("Dupont");
15
16          System.out.println("Apres la copie de la commande");
17          System.out.println(cmd2);
18          System.out.println(cmd3);
19      }
20  }
```

Travail à réaliser :

1. Exécutez la méthode `main` de la classe `TestClonageCommandeV1` et constatez le problème. Indiquer pour quelle raison l'exception `CloneNotSupportedException` a été lancée par la machine virtuelle. Modifier le programme pour que celle-ci n'apparaisse plus.
2. Maintenant observer l'état de la commande n° 2 avant et après la modification de la commande n° 3 obtenue par clonage de la n° 2 en identifiant le problème. La figure 6 est là pour vous aider à comprendre le problème :

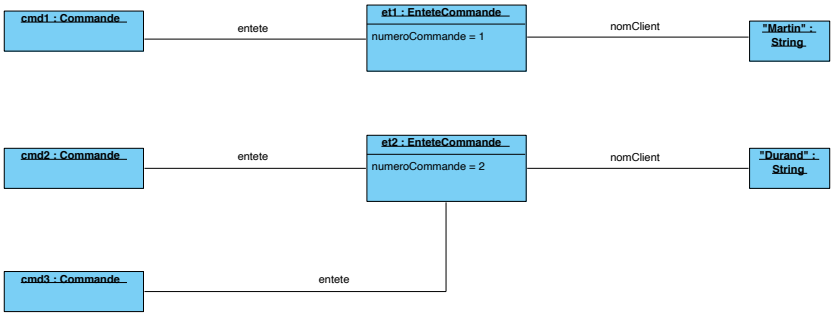

FIGURE 6 – Les différentes instances de `Commande` après clonage

Pour rappel, la méthode `clone`, par défaut, se contente de recopier les valeurs des attributs de l'objet qu'elle duplique et si ces attributs sont des références alors elle recopie les adresses des objets référencés mais ne duplique pas ceux-ci...

Résoudre ce problème en modifiant la méthode `clone()` de la classe `Commande` et en ajoutant une méthode `clone` dans la classe `EnteteCommande`.

Tester votre solution en exécutant le `main` de la classe `TestClonageCommandeV2`.

3. À ce stade, vous devez avoir résolu le problème précédent. Néanmoins votre solution doit avoir pris comme hypothèse que la méthode `changeNomClient` de la classe `EnteteCommande` change l'attribut `nomClient` en lui affectant une nouvelle instance de `String` et elle ne modifie donc pas l'objet référencé par `nomClient`. Elle ne le pourrait d'ailleurs pas car la classe `String` ne crée que des instances **immutables** c'est-à-dire qu'une fois créées, ces instances ne peuvent plus être modifiées.

   Pour vérifier cette hypothèse, donner à `nomClient` le type `StringBuffer` (classe permettant la modification de ses instances) et modifier le constructeur et la méthode `changeNomClient` en conséquence :

```
1  public EnteteCommande(String nomClient, int numero)
2  {
3    this.nomClient = new StringBuffer(nomClient);
4    this.numeroCommande = numero;
5  }
6
7  void changeNomClient(String nomClient)
8  {
9    this.nomClient.replace(0,this.nomClient.length(),nomClient);
10 }
```

   Exécuter la méthode `main` de la classe `TestClonageCommandeV3` et vérifier que votre solution fonctionne toujours. Si ce n'est pas le cas, modifier votre code (principalement la méthode `clone` de la classe `EnteteCommande`) en conséquence.

4. Maintenant il ne doit subsister qu'un seul problème : comment obtenir un nouveau numéro de commande lorsqu'on clone une commande existante ? Modifier votre code pour résoudre ce problème.

Éléments de correction

1. *Le problème ici est qu'on ne peut cloner un objet appartenant à une classe que si celle-ci l'autorise. Cette autorisation est matérialisée par le fait que la classe déclare implémenter l'interface* `Cloneable`*. Donc la déclaration de la classe* `Commande` *doit être de la forme suivante :*

```
public class Commande implements Cloneable {
  ...
}
```

2. *La méthode* `clone()`*, par défaut, se contente de recopier les références. Cela crée un nouvel objet par copie dite « superficielle » (swallow copy) mais qui possède alors les mêmes références que l'original, créant ainsi un phénomène d'alias. Dans le cas présent, la commande n° 3 partage la même entête avec la commande n° 2. Il est nécessaire de définir une méthode* `clone` *dans* `Commande` *qui fasse une copie dite « profonde » (deep copy).*

```
package clonage_ok;

// Point 1 : la classe doit déclarer qu'elle implémente l'interface Cloneable
public class Commande implements Cloneable {
    private static int nombreCommandes = 0;
    private EnteteCommande entete;

    // le constructeur est inaccessible pour obliger l'utilisateur
    // a utiliser la methode creerCommande pour creer des commandes
    private Commande(String nomClient, int numeroCommande) {
        this.entete = new EnteteCommande(nomClient, numeroCommande);
    }

    // permet la creation de commande
    public static Commande creerCommande(String nomClient) {
        return new Commande(nomClient, ++nombreCommandes);
    }

    // permet de modifier le client d'une commande
    public String donneNomClient() {
        return this.entete.donneNomClient();
    }

    // permet de modifier le client d'une commande
    public void changeNomClient(String nomClient) {
        this.entete.changeNomClient(nomClient);
    }

    // permet d'obtenir le numero d'une commande
    public int donneNumeroCommande() {
        return this.entete.donneNumeroCommande();
    }

    public Object clone() throws CloneNotSupportedException {
        // point 2 : on duplique l'entête pour éviter que celle-ci soit
        // partagée après duplication entre l'original et la copie
        Commande c = (Commande) super.clone();
        c.entete = (EnteteCommande) c.entete.clone();

        // point 4 : on genère une nouveau numéro de commande car
        // le clonage est une création d'objet de la même manière
        // que le constructeur...
        c.entete.changeNumeroCommande(++nombreCommandes);

        return c;
    }

    public String toString() {
        return ("Commande " + entete);
    }
}
```

3. *Pour le problème soulevé par l'utilisation de* `StringBuffer`*, il faut également cloner l'attribut* `nomClient` *et donc définir également une méthode* `clone` *dans la classe* `EnteteCommande` *:*

```
1  package clonage_ok;
2
3  public class EnteteCommande implements Cloneable {
4      private StringBuffer nomClient;
5      private int numeroCommande;
6
7      EnteteCommande(String nomClient, int numero) {
8          this.nomClient = new StringBuffer(nomClient);
9          numeroCommande = numero;
10     }
11
12     public String donneNomClient() {
13         return this.nomClient.toString();
14     }
15
16     void changeNomClient(String nomClient) {
17         this.nomClient.replace(0,this.nomClient.length(),nomClient);
18     }
19
20     public int donneNumeroCommande() {
21         return this.numeroCommande;
22     }
23
24     void changeNumeroCommande(int numeroCommande) {
25         this.numeroCommande = numeroCommande;
26     }
27
28     // Point 3 : clonage specifique de l'entete
29     public Object clone() throws CloneNotSupportedException {
30
31         EnteteCommande nouvelEntete = (EnteteCommande) super.clone();
32         nouvelEntete.nomClient = new StringBuffer(this.nomClient);
33         nouvelEntete.numeroCommande = this.numeroCommande;
34         return nouvelEntete;
35     }
36
37     public String toString() {
38         return ("numero = " + numeroCommande + " effectuée par : " + nomClient);
39     }
40 }
```

4. *Ne pas oublier que lorsqu'on clone une instance de* Commande*, on crée un objet, comme lors de l'appel d'un constructeur, et donc il faut incrémenter le compteur du nombre d'instances créées (cf. code donné au point 2).*