TD n° 3
Licence Informatique (L2)
« Programmation objet avancée »

F. BERTRAND

Année universitaire 2020–2021

Éléments de correction

Concepts abordés :
- Polymorphisme et liaison dynamique
- Affectations dans une hiérarchie de classes
- Classes et méthodes abstraites

1 Retour sur la méthode `equals()`

Sachant qu'une méthode `equals` est présente dans la classe `Object` avec la signature suivante :

```
1  public boolean equals(Object o);
```

et la classe `Produit` suivante :

```
1  class Produit {
2      private String categorie;
3      private String nom;
4
5      public Produit(String categorie, String nom)  {
6          this.categorie = categorie;
7          this.nom = nom;
8      }
9
10     public boolean equals(Produit p) {
11         if (p == this)
12            return true;
13         if (this.categorie.equals(p.categorie) && this.nom.equals(p.nom))
14            return true;
15         else
16            return false;
17     }
18
19     public static void main(String[] args) {
20         Produit a = new Produit("A", "1");
21         Object  b = new Produit("A", "1");
22         System.out.println("Resultat = " + a.equals(b));
23     }
24 }
```

Expliquer les points suivants :

1. Quel test effectue la méthode `equals` présente dans `Object` : identité ou égalité (des objets) ?...

2. Pourquoi le code ci-dessus produit l'affichage « `false` »?

Éléments de correction

Cet exercice a pour objectif de montrer que le choix d'une méthode (ici `equals`) s'effectue en 2 étapes :

1. *Détermination de l'existence de la méthode en se basant sur le type statique de l'objet appelé et en recherchant, à partir de cette classe (correspondant au type statique), la définition de la méthode appelée (dans cette classe ou dans ses super-classes). Cette première étape a pour but de s'assurer qu'au moins une version de la méthode appelée a été définie.*
2. *Puis, une fois cette vérification effectuée, le compilateur génère un test qui sera réalisé lors de l'exécution et qui permettra de déterminer le type dynamique de l'objet appelé et de rechercher, en fonction de cette classe (type dynamique), la version « la plus proche » de la méthode appelée si elle a été redéfinie.*

Pour la première question, la méthode `equals` de la classe `Object` se comporte comme l'opérateur == et effectue un test d'identité : elle vérifie que les adresses en mémoire des objets sont identiques et donc que `a` et `b` référencent le même objet, ce qui n'est pas le cas ici.

Concernant la deuxième question, différentes étapes :

1. *détermination du type statique du paramètre de la méthode appelée : le type statique de `b` est `Object` donc la méthode qui sera recherchée par le compilateur aura donc la signature : `equals(Object)` ;*
2. *puis le compilateur détermine le type statique de la référence (variable) `a` sur laquelle la méthode `equals` est appelée : le type statique de `a` est `Produit` ;*
3. *le compilateur vérifie que la classe `Produit` possède (définition locale ou héritage) une méthode `equals(Object)` : c'est le cas car la classe `Produit` hérite de la classe `Object` et donc de cette méthode.*
4. *puis, dans le cas général, le compilateur génèrerait un test qui sera exécuté à l'exécution du programme, permettant de déterminer la type dynamique de l'objet sur lequel la méthode `equals` est appelée). Ici, dans notre cas précis, ce test n'est pas généré car une référence de type `Produit` ne peut référencer qu'une instance de `Produit`. En revanche si le code exécuté était `b.equals(a)` le test serait généré car une référence de type `Object` (ici `b`) peut référencer soit une instance d'`Object`, soit une instance de `Produit`.*
5. *à l'exécution, détermination du type dynamique de `a` qui est `Produit` donc recherche de la méthode `equals(Object)` dans cette classe. Si la méthode n'est pas trouvée dans `Produit` alors l'interpréteur Java regardera dans la super-classe et, s'il ne la trouve pas, continuera dans la super-super-classe, et ainsi de suite... Si le code a pu se compiler cela garantit l'existence d'au moins une version de la méthode appelée.*

   La méthode exécutée est donc la méthode `equals(Object)` présente dans `Object` (super-classe de `Produit`) qui se comporte comme l'opérateur « == » (teste l'identité) et, comme `a` et `b` ne référencent pas la même instance, le test retourne faux.

   *Dans la classe `Produit`, la méthode `equals` a été **surchargée** (changement du type de son paramètre) et non **redéfinie** et elle ne sera pas appelée dans le code fourni.*

   Elle pourrait être appelée que si on lui passait un paramètre qui aurait pour type statique `Produit` et que le type statique de l'objet sur lequel elle serait appelée était également `Produit`.

Maintenant on décide de modifier la méthode `main` de la manière suivante :

```
public static void main(String[] args) {
   Produit a = new Produit("A", "1");
   Produit b = new Produit("B", "1");
   ArrayList<Produit> liste = new ArrayList<Produit>();
   liste.add(a);
   liste.add(b);
   System.out.println("Resultat 'contains a' = " + liste.contains(a));
   System.out.println("Resultat 'contains new Produit' = "
                                      + liste.contains(new Produit("A", "1")));
}
```

Sachant que la méthode `contains` d'`ArrayList` appelle la méthode `equals(Object o)`, expliquer les points suivants :

1. Pourquoi, à l'exécution, le premier résultat (`liste.contains(a)`) retourne vrai alors que le second retourne faux ?
2. Comment procéder pour que les deux tests retournent vrai ?

Éléments de correction

1. *Comme il a été dit précédemment, si la méthode `equals(Object o)` n'a pas été redéfinie alors sa sémantique est celle de l'opérateur « == ». Donc pour la comparaison avec l'objet référencé par la référence « `a` » cela fonctionne (c'est le même objet). Par contre, pour le second test, nous avons à faire à un objet qui n'est pas dans la liste donc cela ne fonctionne pas.*
2. *La solution est de redéfinir la méthode `equals(Object o)` car c'est celle qui sera appelée par `contains`. En effet, `contains(Object o)` n'est pas une méthode ayant comme paramètre un type générique donc elle ne peut appeler que la méthode `equals(Object o)`. Dans le code de la méthode `equals` présenté ci-dessous :*

```
    public boolean equals(Object o) {
        if (o == this) // auto-test
           return true;
        if (o.getClass() != Produit.class) // o doit référencer un Produit
           return false;
        else {
             Produit p = (Produit) o;
             if (this.categorie.equals(p.categorie) && this.nom.equals(p.nom))
                return true;
             else
                return false;
        }
   }
```

quelques éléments à noter :

— *le premier test correspond à un auto-test qui est un cas (rare) mais qui peut se produire par exemple :*

```
   Produit p1 = new Produit("A", "1");
   Produit p2 = p1;
   // on suppose un certain nombre de lignes de code entre les lignes
   // précédentes et le ligne suivante sinon le test paraît aberrant...
   if (p1.equals(p2))
      System.out.println("Un objet est forcément égal à lui-même !...");
```

— *le deuxième test s'assure que l'objet fourni en paramètre est bien une instance de `Produit` car, comme la méthode `equals` prend un `Object` en paramètre on peut lui passer n'importe quel type d'objet !...*

— *une fois qu'on est sûr (à l'exécution) que l'objet passé en paramètre est du type* `Produit`*, on peut procéder au transtypage notamment pour accéder aux attributs (*`categorie` *et* `nom`*) de* `Produit`*;*
— *le dernier test est le test d'égalité de deux produits : ils sont égaux s'ils appartiennent à la même catégorie et qu'ils possèdent le même nom. Généralement c'est le concepteur de la classe qui définit le critère d'égalité...*

On décide maintenant de créer une classe `ProduiteSoldé` qui hérite de `Produit` (munie maintenant de la version redéfinie d'`equals(Object)`) :

```
1  class ProduitSoldé extends Produit {
2      private float remise;
3
4      public ProduitSoldé(String categorie, String nom, float remise)  {
5          super(categorie,nom);
6          this.remise = remise;
7      }
8
9      public static void main(String[] args) {
10         Produit a = new Produit("A", "1");
11         ProduitSoldé  b = new ProduitSoldé("A", "1", 0.5);
12         System.out.println("Resultat = " + a.equals(b));
13     }
14 }
```

Pourquoi l'exécution du `main` affiche `false`?...

Éléments de correction

Le `main` *affiche* `false` *car la méthode* `equals` *appelée est celle de* `Produit` *or l'appel* `o.getClass()` *retourne la classe* `ProduitSoldé` *qui est différent de la classe* `Produit.class`*.*
Si on souhaite (et si cela a un sens) comparer une instance d'une sous-classe avec une instance de sa super-classe alors, dans la méthode `equals(Object)` *de sa super-classe, il faut utiliser l'opérateur* `instanceof` *à la place de* `getClass` *car celui-ci s'assurera que l'objet passé en paramètre possède le type* `Produit` *(et non, comme dans la version avec* `getClass()`*, que l'objet* ***est une instance*** *de* `Produit`*) :*

```
1     // nouvelle version de equals définie dans la classe Produit
2     public boolean equals(Object o) {
3         if (o == this) // auto-test
4            return true;
5         if (! (o instanceof Produit)) // ce test retournera vrai si o représente
6                                       // une instance de ProduitSoldé
7            return false;
8         else {
9              Produit p = (Produit) o;
10             if (this.categorie.equals(p.categorie) && this.nom.equals(p.nom))
11                return true;
12             else
13                return false;
14        }
15    }
```

2 Classes abstraites

On considère un jeu constitué de différentes pièces (ex. un jeu d'échec). Chaque type de pièce possède un nom, une couleur et un déplacement spécifique (méthode `deplacer`).

Le déplacement pourra s'exprimer soit en fournissant des coordonnées, soit, en adoptant une structuration objet et en définissant une classe `Case`, solution que nous adopterons.

1. Définir une classe `Case` en veillant à ne pouvoir créer que des cases valides, c'est-à-dire dont les coordonnées sont dans le plateau de jeu (dans le cas des échecs ce plateau fait 8×8 cases).
2. Définir une classe abstraite `Piece` en indiquant pour quelle raison cette classe doit être déclarée abstraite ;
3. Définir deux sous-classes `Tour` (permettant des déplacements horizontaux et verticaux) et `Fou` (permettant uniquement des déplacements diagonaux).

Éléments de correction

L'idée générale est de spécifier que chaque pièce peut se déplacer mais qu'au niveau de la classe `Piece` on est incapable de définir cette méthode. Donc on se contente de la spécifier (déclarer) abstraite.

Un autre point concerne la création d'instances valides de `Case`. Ici il faut souligner l'intérêt d'une méthode de classe qui contrôle l'accès au constructeur et qui ne fait appel à `new` que lorsque les paramètres ont été vérifiés. Pour que cela fonctionne il est nécessaire de spécifier le constructeur `private` sinon son appel direct est toujours possible.

Le code :

```
1  /* Représente une case d'un plateau de jeu (ici un échiquier) */
2  public class Case
3  {
4      private int noLigne;
5      private int noCol;
6      // taille du plateau
7      private static final int MAX_LIGNE = 8;
8      private static final int MAX_COL = 8;
9
10     // le constructeur est qualifié privé.
11     // De cette manière l'utilisateur de cette classe devra
12     // employer la méthode de classe creerCase.
13     private Case(int l, int c) {
14         this.noLigne = l;
15         this.noCol = c;
16     }
17
18     // l'objectif de cette méthode est de contrôler la validité des paramètres
19     // AVANT l'appel du constructeur.
20     public static Case creerCase(int l, int c) {
21         if (0 <= l && l < MAX_LIGNE && 0 <= c && c < MAX_COL)
22         {
23             return new Case(l, c);
24         }
25         else //  si coordonnées invalides, on ne retourne rien...
26         {
27             return null;
28         }
29     }
30
31     public int donneNoLigne()
32     {
33         return this.noLigne;
34     }
35
36     public void changeNoLigne(int noLigne)
37     {
38         this.noLigne = noLigne;
39     }
40
41     public int donneNoCol()
42     {
43         return this.noCol;
44     }
```

```
45
46      public void changeNoCol(int noCol)
47      {
48          this.noCol = noCol;
49      }
50  }
```

```
1   import java.awt.Color;
2
3   /* Cette classe est abstraite car elle possède une méthode abstraite */
4   public abstract class Piece
5   {
6       protected String identifiant;
7       protected Color couleur;
8       protected Case position;
9
10      public Piece(String nom, Color c, Case position)
11      {
12          this.identifiant = nom;
13          this.couleur = c;
14          this.position = position;
15      }
16
17      /* La définition de cette méthode ici signifie que :
18       * - toutes les sous-classes (pour devenir instanciables)
19       *   devront définir cette méthode.
20       * - mais qu'à ce stade on ne peut pas définir cette méthode
21       */
22      public abstract boolean deplacer(Case destination);
23  }
```

```
1   import java.awt.Color;
2
3   /* Exemple de sous-classe concrète (instanciable) de Piece */
4   public class Tour extends Piece
5   {
6       public Tour(String id, Color c, Case position)
7       {
8           super(id, c, position);
9       }
10
11      /* Une tour se déplace sur une ligne ou une colonne de l'échiquier */
12      public boolean deplacer(Case destination)
13      {
14          // si elle est déjà sur la ligne de destination, on ne change que la colonne...
15          if (this.position.donneNoLigne() == destination.donneNoLigne())
16          {
17              this.position.changeNoCol(destination.donneNoCol());
18              return true;
19          }
20          // si elle est déjà sur la colonne de destination, on ne change que la ligne...
21          else if (this.position.donneNoCol() == destination.donneNoCol())
22          {
23              this.position.changeNoLigne(destination.donneNoLigne());
24              return true;
25          }
26          else
27          {
28              return false;
29          }
30      }
31  }
```

```
1   import java.awt.Color;
2
3   /* Exemple de sous-classe concrète (instanciable) de Piece */
4   public class Fou extends Piece
5   {
6       public Fou(String id, Color c, Case position)
7       {
8           super(id, c, position);
9       }
10
11      /* Un fou se déplace se déplace en diagonale sur l'échiquier */
12      public boolean deplacer(Case destination)
13      {
14          // le déplacement en diagonale est fondé sur le fait que l'écart entre
15          // les lignes (depart/arrivée) et les colonnes doit être égal...
```

```
16          if (Math.abs(this.position.donneNoLigne() - destination.donneNoLigne())
17                  == Math.abs(this.position.donneNoCol() - destination.donneNoCol()))
18          {
19              this.position.changeNoLigne(destination.donneNoLigne());
20              this.position.changeNoCol(destination.donneNoCol());
21              return true;
22          }
23          else
24          {
25              return false;
26          }
27      }
28  }
```