Égalité entre points et point nommés
Corrigé

Exercice 1 : Comprendre la surcharge et la redéfinition

L'objectif de cet exercice est de comprendre les notions de surcharge et de redéfinition en s'appuyant sur les classes Point (listing 1) et PointNommé (listing 2).

On s'intéresse à l'égalité logique de deux points. Contrairement à l'égalité physique qui est réalisée par la comparaison des poignées (il y a égalité physique de deux poignées si elles référencent le même objet), il s'agit de vérifier si les deux points ont les mêmes valeurs d'attributs [1].

1.1. Définir une méthode `isEqual` [2] dans la classe `Point`.

Solution :

```
1  public boolean isEqual(Point autre) { // Méthode 1
2     return this == autre
3        || (autre != null && autre.x == this.x && autre.y == this.y);
4  }
```

On donne le numéro 1 à cette méthode.

La première sous-expression `this == autre` a deux objectifs :

1. garantir que la relation d'égalité est réflexive,
2. améliorer les performances, car si les deux poignées référencent le même objet, il n'est pas utile de comparer tous les attributs qui sont forcément égaux.

La sous-expression `autre != null` est importante pour que la méthode soit toujours définie.

1.2. On considère les déclarations suivantes :

```
1  PointNomme pn1 = new PointNomme("A", 1, 2);
2  PointNomme pn2 = new PointNomme("A", 1, 2);
3  PointNomme pn3 = new PointNomme("B", 1, 2);
4  PointNomme pn4 = new PointNomme("A", 1, 1);
5  PointNomme pn5 = new PointNomme("B", 1, 1);
6  Point p1 = new Point(1, 1);
7  Point p2 = new Point(1, 1);
8  Point p3 = p2;
9  Point q1 = pn4;
10 Point q2 = pn5;
11 Point q3 = new PointNomme("A", 1, 1);
```

1.2.1. Indiquer, pour les expressions suivantes, les méthodes exécutées et les résultats obtenus.

1. L'égalité logique est en fait plus difficile à définir et dépend de la classe considérée.
2. La méthode qui correspond à l'égalité logique en Java est la méthode `equals` de la classe `Object`. Tout ce qui sera dit ici sur `isEqual` s'appliquera à `equals`.

```
p1 == p2
p3 == p2
p1.isEqual(p2)
p1.isEqual(pn4)
pn1.isEqual(pn3)
```

Solution :

1. `p1 == p2` s'évalue à faux car p1 et p2 référencent deux points différents même s'ils ont mêmes abscisses et mêmes ordonnées.
2. `p3 == p2` s'évalue à vrai car p3 et p2 référencent le même point.
3. `p1.isEqual(p2)`
   (a) Liaison statique (ou résolution de la surcharge) : Il existe une méthode `isEqual` dans `Point`, type de p1. C'est la seule. `p2` de type `Point` est compatible avec son paramètre formel `Point`. C'est donc cette méthode (`isEqual(Point)`) qui est retenue par le compilateur.
   (b) Laison dynamique : La classe de l'objet attaché à `p1` est un Point, la méthode exécutée (laison dynamique) est donc la méthode isEqual(Point) de la classe Point (méthode 1).
   (c) L'évaluation donne vrai.
4. `p1.isEqual(pn4)` est acceptée en suivant le même raisonnement et s'appuyant sur le fait que le paramètre effectif de type `PointNomme` est compatible avec le paramètre formel `Point`.
   L'évaluation donne vrai.
5. `pn1.isEqual(pn3)`
   (a) Liaison statique (ou résolution de la surcharge) : Le type de `pn1` est `PointNomme`. Dans la classe `PointNomme`, il n'y qu'une méthode `isEqual`. Elle a pour paramètre formel un `Point` et donc accepte `pn3` du type `PointNomme` (car sous-type de `Point`).
   (b) Liaison dynamique : C'est la méthode `isEqual(Point)` de la classe `PointNomme` qui est exécutée. Ici, c'est celle héritée de `Point` (méthode 1).
   (c) L'évaluation donne vrai.

1.2.2. Dans le dernier cas, indiquer comment faire pour que le résultat de l'expression soit « faux » et modifier en conséquence les classes Point et PointNommé.

Solution : L'appel `pn1.isEqual(pn3)` nous conduit à définir une nouvelle méthode `isEqual` dans `PointNomme`, type du récepteur `pn1` avec un paramètre de type `PointNomme`, type de `pn3`.

Un point nommé est égal à un autre point nommé s'ils sont égaux en temps que Point et s'ils ont mêmes noms. Notons l'utilisation de la méthode `equals` pour l'égalité des noms : c'est bien l'égalité logique qu'il faut utiliser et non l'égalité physique (==).

```
// Surcharge du isEqual(Point) de Point
public boolean isEqual(PointNomme autre) { // Méthode 2
   return super.isEqual(autre) && nom.equals(autre.nom);
}
```

On numérote 2 cette méthode.

Attention : On aurait aussi pu définir la méthode précédente de la manière suivante :

```
// Surcharge du isEqual(Point) de Point
public boolean isEqual(PointNomme autre) { // Méthode 2faux
   return this.isEqual((Point) autre) && nom.equals(autre.nom);
}
```

Si ce code fonctionne pour l'instant, il deviendra bientôt faux ! Ce qu'il faut faire, c'est bien utiliser `super.isEquals(autre)` car on veut comparer les deux objets en tant que `Point`.

Reprenons l'évaluation de `pn1.isEqual(pn3)` :

1. Liaison statique (ou résolution de la surcharge) : Le type de `pn1` est `PointNomme`. Dans la classe `PointNomme`, il y a maintenant deux méthodes `isEqual`. La première prend en paramètre en `Point`, la seconde un `PointNomme`. L'appel se faisant avec `pn3` du type `PointNomme`, les deux signatures sont possibles mais la première nécessite d'utiliser le sous-typage (distance de 1 entre signature et appel) et l'autre correspond exactement (distance de 0). C'est dont `isEqual(PointNomme)` qui est la signature (l'opération) retenue.
2. Liaison dynamique : C'est la méthode `isEqual(PointNomme)` de la classe `PointNomme` qui est exécutée, celle qu'on vient de définir (méthode 2).
3. La méthode appelée est d'abord 2, qui appelle ensuite 1 (qui répond vrai) puis qui compare les noms (la réponse est faux).

   L'évaluation donne donc faux.

1.2.3. Indiquer, pour les expressions suivantes, les méthodes exécutées et les résultats obtenus.

```
q1.isEqual(p1)
q3.isEqual(pn4)
pn4.isEqual(p1)
```

Solution :

`q1.isEqual(p1)`

1. Le type de `q1` est `Point`. La seule méthode candiate est `isEqual(Point)`. Le paramètre effectif est compatible. L'appel est donc accepté.
2. La liaison dynamique fait que c'est l'implantation de `isEqual(Point)` de la classe `PointNomme`, classe de l'objet attaché à `q1` qui sera exécutée (méthode 1).
3. Le résultat est donc vrai.

`q3.isEqual(pn4)`

— Le même raisonnement conduit au fait que c'est l'opération de signature `isEqual(Point)` qui est appelée et la liaison dynamique conduit à exécuter la méthode 1.
Remarque : On ne peut pas envisager la méthode `isEqual(PointNomme)` de la classe `PointNomme` car le type de `q3` est `Point`. Cette méthode n'est donc pas candidate !
— Le résultat est donc vrai.

`pn4.isEqual(p1)`

1. Le type de `pn4` est `PointNomme`. On a donc deux opérations `isEqual` candidates : `isEqual(Point)` et `isEqual(PointNomme)`. Seule la première accepte un paramètre effectif de type `Point`. C'est donc elle qui est retenue.
2. La liaison dynamique conduit à exécuter la méthode 1.
3. Le résultat est donc vrai.

1.2.4. Indiquer comment faire pour que le résultat de la dernière expression soit « faux » et modifier en conséquence les classes Point et PointNommé.

Solution : Il s'agit de redéfinir le `isEqual(Point)` de `Point` dans `PointNommé` :

```
1 // Redéfinition du isEqual(Point) de Point
2 @Override public boolean isEqual(Point autre) { // Méthode 3
3    return false;
4 }
```

Notons qu'en définissant cette nouvelle méthode dans la classe PointNomme, on n'ajoute pas de nouvelle opération (cette signature `isEqual(Point)` était déjà présente sur `PointNomme`). On a juste changé l'implantation de la méthode qui était déjà présente dans `PointNomme` car héritée de `Point`. C'est bien une redéfinition d'où l'utilisation de `@Override`.

Remarque : On a adopté une approche type *Extreme Programming* où on fait juste ce qu'il faut pour répondre au problème posé.

Mais est-ce vrai qu'un point nommé ne peut jamais être égal à un point ?

Nous allons voir dans les questions suivantes que la réponse est non !

`pn4.isEqual(p1)` :

1. Liaison statique : Toujours qu'une seule signature possible.
2. Liaison dynamique : le type réel du récepteur (classe de l'objet attaché au récepteur) est `PointNomme`, la méthode exécutée est donc `isEquals(Point)` de la classe `PointNomme`, donc la méthode que l'on vient de définir, méthode 3.
3. L'évaluation donne faux !

1.2.5. Indiquer alors, pour l'expression suivante, les méthodes exécutées et les résultats obtenus.

```
1 pn4.isEqual(q3)
```

Solution :

1. Liaison statique : Le type de `pn4` étant `PointNomme`, on a deux méthodes `isEqual` candidates : `isEqual(Point)` et `isEqual(PointNomme)`. Seule la première accepte un paramètre effectif de type `Point`. C'est donc elle qui est retenue.
2. La liaison dynamique conduit à exécuter la méthode `isEqual(Point)` qui est dans la classe de l'objet associé à `pn4` (type réel), donc `PointNommé`. C'est donc la méthode (3).
3. Le résultat est donc faux.

1.2.6. On souhaite que l'expression précédente s'évalue à « vrai ». Indiquer les éventuelles modifications à apporter.

Solution : En renvoyant `false` tout à l'heure on a répondu à la question mais on se rend compte que ce n'est pas satisfaisant. En effet, l'égalité d'un point nommé est d'un point peut être vraie si derrière le point, il y a en fait un point nommé !

Voici la nouvelle version de la méthode (3) :

```
1 // Redéfinition du isEqual(Point) de Point
2 @Override public boolean isEqual(Point autre) { // Méthode 3
3    return autre instanceof PointNomme && this.isEqual((PointNomme) autre);
4 }
```

`pn4.isEqual(q3)` :

- Liaisons statique et dynamique ne changent pas. On appelle toujours la méthode 3.
- Le type réel de `q3` est bien `PointNomme`, il y a donc appelle de la méthode 2 qui à son tour appelle la méthode 1.

Attention : Si on avait gardé la méthode 2faux, on aurait eu `StackOverFlowError` à cause d'appels récursifs infinis : 3 appelle 2faux qui appelle 3 qui appelle 2faux... Car on a depuis redéfini la méthode `isEquals(Object)` dans `PointNomme` et donc ce n'est pas l'égalité des points qui est appelée (`Point.isEqual(Point)` mais l'égalité des points nommés `PointNomme.isEqual(Point)`) !

1.2.7. Indiquer alors, pour les expressions suivantes, les méthodes exécutées et les résultats obtenus.

```
1 pn4.isEqual(pn5)
2 pn4.isEqual(q2)
3 q1.isEqual(pn5)
4 q1.isEqual(q2)
```

Solution :

`pn4.isEqual(pn5)`

1. Liaison statique : deux méthodes `isEqual` candidatent dans `PointNomme`, type apparent de `pn4` : `isEqual(Point)` et `isEqual(PointNomme)`.

   Le paramètre effectif `pn5` étant de type `PointNomme`, les deux fonctionnent la première avec une distance de 1 (un recours au sous-typage) et l'autre avec une distance de 0. C'est donc la deuxième, `isEqual(PointNomme)`, qui est sélectionnée.
2. Liaison dynamique : la méthode associée à l'opération `isEqual(PointNomme)` dans `PointNomme`, type réel de `pn4`, est la méthode 2.
3. L'évaluation donne faux car les noms des points sont différents (2 appelle 1 puis compare les noms).

`pn4.isEqual(q2)`

1. Liaison statique : deux méthodes `isEqual` candidatent dans `PointNomme`, type apparent de `pn4` : `isEqual(Point)` et `isEqual(PointNomme)`.

Seule la première accepte un paramètre de type `Point`, le type apparent de `q2`. C'est donc la méthode `isEqual(Point)` qui est sélectionnée.

2. Liaison dynamique : Le type réel du récepteur `pn4` est `PointNomme`. La méthode correspondant à l'opération `isEqual(Point)` dans `PointNomme` est la méthode 3.
3. L'évaluation donne faux car les noms des points sont différents (3 appelle 2 qui appelle 1 puis compare les noms).

`q1.isEqual(pn5)`

1. Liaison statique : une seule candidate `isEqual(Point)` car le type apparent de `q1` est `Point`. Elle convient car le type apparent de `pn5` est `PointNomme`, sous-type de `Point`. C'est donc elle.
2. Liaison dynamique : c'est la méthode `isEqual(Point)` dans `PointNomme`, type réel de `q1`. C'est donc la méthode 3.
3. L'évaluation donne faux : 3 appelle 2 qui appelle 1 (puis compare les noms qui sont différents).

`q1.isEqual(q2)`

1. Liaison statique : une seule candidate `isEqual(Point)` car le type apparent de `q1` est `Point`. Elle convient car le type apparent de `q2` est `Point`. C'est donc elle.
2. Liaison dynamique : c'est la méthode `isEqual(Point)` dans `PointNomme`, type réel de `q1`. C'est donc la méthode 3.
3. L'évaluation donne faux : 3 appelle 2 qui appelle 1 (puis compare les noms qui sont différents).

Remarque : Dans cette dernière expression, il y a toute l'ambigüité d'un langage à objet typé statiquement :

— Quand on regarde les types des poignées (types apparents), on peut penser que c'est l'égalité des points.
— Ce qui est exécutée, c'est la comparaison entre objets de type `PointNomme`.

1.2.8. Indiquer et commenter le résultat des deux expressions suivantes :

```
1 p1.isEqual(pn4)
2 pn4.isEqual(p1)
```

Solution : La première s'évalue à vrai (exécution de la méthode 1).

La seconde s'évalue à faux (exécution de 3).

On constate que l'égalité que nous venons de définir n'est pas symétrique !

Important : Quand on définit l'égalité, il faut s'assurer qu'elle est réflexive, symétrique et transitive ! Elle doit aussi être consistante : plusieurs appels avec les mêmes opérandes doivent donner le même résultat. Enfin, l'égalité d'un objet avec null doit renvoyer faux (voir la documentation de la méthode `equals` de `Object`).

Ici, on pourrait commencer par vérifier que les deux objets sont de mêmes types en utilisant la méthode `getClass()` de `Object` qui retourne la classe de l'objet.

La méthode `equals` dans `Point` :

```java
@Override public boolean equals(Object obj) {
   if (this == obj) { // mêmes objets
      return true;
   } else if (obj != null && obj.getClass().equals(this.getClass())) {
      // obj est aussi un Point.
      Point pt = (Point) obj;
      return this.x == pt.x && this.y == pt.y;

   } else {
      return false;
}  }

@Override public int hashCode() {
   return java.util.Arrays.hashCode(new double[] {this.x, this.y});
}
```

La méthode `equals` dans `PointNomme` :

```java
@Override public boolean equals(Object obj) {
   if (this == obj) { // mêmes objets
      return true;
   } else if (obj != null && obj.getClass().equals(this.getClass())) {
      PointNomme pn = (PointNomme) obj;
      return super.equals(pn) && this.nom.equals(pn.nom);

   } else {
      return false;
}  }

@Override public int hashCode() {
   return java.util.Objects.hash(this.getX(), this.getY(), this.nom);
}
```

Remarque : L'égalité logique correspond à la méthode `equals` de `Object`. Comme toute classe en Java hérite forcément de `Object`, quand on définit une nouvelle classe C, il faut donc se demander s'il y a des méthodes à redéfinir, en particulier `equals`. Ce qui vient d'être fait ici pour `PointNomme` vis à vis de `Point` sera à faire pour `C` vis à vis de `Object`.

Cependant, en général on n'écrit que la redéfinition `C.equals(Object)` et pas la méthode surchargée `C.equals(C)`, comme ceci a été fait pour `Point` et `PointNomme` ci-dessus.

La documentation de la méthode `equals` de `Object` indique que si on redéfinit `equals`, il faut aussi redéfinir `hashCode` car deux objects égaux doivent avoir la même valeur de hachage.

Compléments Voici le résultat de l'évaluation des différentes expressions avec les classes telles qu'écrites à l'issue de la question 1.2.6 (les numéros affichés sont les numéros des différentes méthodes `isEqual` appelées).

```
p1 == p2 --> false
p3 == p2 --> true
p1.isEqual(p2) --> 1 > true
p1.isEqual(pn4) --> 1 > true
pn1.isEqual(pn3) --> 2 > 1 > false

q1.isEqual(p1) --> 3 > false
q3.isEqual(pn4) --> 3 > 2 > 1 > true
pn4.isEqual(p1) --> 3 > false

pn4.isEqual(q3) --> 3 > 2 > 1 > true
```

```
pn4.isEqual(pn5) --> 2 > 1 > false
pn4.isEqual(q2) --> 3 > 2 > 1 > false
q1.isEqual(pn5) --> 3 > 2 > 1 > false
q1.isEqual(q2) --> 3 > 2 > 1 > false

p1.isEqual(pn4) --> 1 > true
pn4.isEqual(p1) --> 3 > false
```

Listing 1 – La classe Point

```java
/** Définition d'un point avec ses coordonnées cartésiennes. */
public class Point {
   private double x;  // abscisse
   private double y;  // ordonnée

   /** Construire un point à partir de son abscisse et de son ordonnée.
     * @param x abscisse
     * @param y ordonnée */
   public Point(double x, double y) {
      this.x = x;
      this.y = y;
   }

   /** Abscisse du point */
   public double getX() {
      return x;
   }

   /** Ordonnée du point */
   public double getY() {
      return y;
   }

   /** Changer l'abscisse du point
     * @param x la nouvelle abscisse */
   public void setX(double x) {
      this.x = x;
   }

   /** Changer l'ordonnée du point
     * @param y la nouvelle ordonnée */
   public void setY(double y) {
      this.y = y;
   }

   @Override public String toString() {
      return "(" + x + "," + y + ")";
   }

   /** Distance par rapport à un autre point */
   public double distance(Point autre) {
      double dx2 = Math.pow(autre.x - x, 2);
      double dy2 = Math.pow(autre.y - y, 2);
      return Math.sqrt(dx2 + dy2);
   }

  /** Translater le point.
    * @param dx déplacement suivant l'axe des X
    * @param dy déplacement suivant l'axe des Y */
   public void translater(double dx, double dy) {
      x += dx;
      y += dy;
}  }
```

Listing 2 – La classe PointNommé

```
1  /** Un point nommé est un point avec un nom. */
2  public class PointNomme extends Point {
3     private String nom;
4
5     /** Construire un point nommé. */
6     public PointNomme(String nom, double x, double y) {
7        super(x, y);
8        this.nom = nom;
9     }
10
11    /** Nom du point nommé */
12    public String getNom() {
13       return nom;
14    }
15
16    /** Changer le nom du point nommé
17     * @param nom le nouveau nom */
18    public void setNom(String nom) {
19       this.nom = nom;
20    }
21
22    @Override public String toString() {
23       return nom + ":" + super.toString();
24 }  }
```