Héritage, masquage de méthode

On considère des rectangles donc les côtés ne sont plus nécessairement parallèles aux axes. Un tel rectangle est vu comme un rectangle aux côtés parallèles qui aurait été incliné d'un certain angle par rapport à l'horizontale.

Exercice n° 1

> **ÉNONCÉ**
>
> En utilisant l'héritage, définir une classe SlantedRectangle permettant de manipuler de tels objets. Définir des constructeurs appropriés.

> **SOLUTION**
>
> *La résolution de l'exercice cet est pratique (voir SlantedRectangle.java)*

Exercice n° 2

> **ÉNONCÉ**
>
> Définir une méthode rotate dans l'esprit de la méthode translate.

> **SOLUTION**
>
> *La résolution de l'exercice cet est pratique*

Exercice n° 3

ÉNONCÉ

De quelles méthodes hérite la classe SlantedRectangle ? Redéfinir celles qui le nécessitent.

SOLUTION

La classe SlantedRectangle hérite des méthodes de la classe Rectangle.

Le reste de l'exercice est pratique

Exercice n° 4

ÉNONCÉ

Pour chacun des appels de méthode ci-dessous, dire s'il va être compilé correctement et auquel cas, quelle méthode est appelée effectivement à l'exécution ?

```
 Point p = new Point(1,2);
Rectangle r = new Rectangle(p, 2, 3);
Rectangle t = new SlantedRectangle(p, 2, 3);
SlantedRectangle s = new SlantedRectangle(p, 2, 3);
System.out.println(r.surface());
r.rotate(2);
System.out.println(r.contains(p));
System.out.println(t.surface()); t.rotate(2);
System.out.println(t.contains(p));
System.out.println(s.surface()); s.rotate(2);
System.out.println(s.contains(p));
```

SOLUTION

LIGNE PAR LIGNE

```
Point p = new Point(1,2);
```

✓ Compilation OK

```
Rectangle r = new Rectangle(p, 2, 3);
```

✓ Appelle le constructeur

```
Rectangle(Point, double, double)
```

```
Rectangle t = new SlantedRectangle(p, 2, 3);
```

✗ Ne compile pas ; car malgrès le fait que le polymorphisme soit bien défini, SlantedRectangle n'a aucun constructeur qui prend en paramètre des éléments tels que définis ici.

```
SlantedRectangle s = new SlantedRectangle(p, 2, 3);
```

✗ Ne compile pas ; car malgrès le fait que le polymorphisme soit bien défini, SlantedRectangle n'a aucun constructeur qui prend en paramètre des éléments tels que définis ici.

Méthodes appelées

```
System.out.println(r.surface());
```

✓ Compilation OK

```
r.rotate(2);
```

✗ Erreur de compilation. La méthode rotate() n'existe pas dans Rectangle et r est un Rectangle

```
System.out.println(r.contains(p));
```

✓ Compilation OK

```
t.rotate(2);
```

❌ Même erreur : t est typé Rectangle → ne voit pas rotate()

```
System.out.println(t.contains(p));
```

✓ OK → SlantedRectangle.contains(Point) est appelé (Grâce au polymorphisme même si t est typé Rectangle)

```
s.rotate(2);
```

✓ Compilation OK

```
System.out.println(s.contains(p));
```

✓ Compilation OK

En resumé, ce code comporte des erreur que ferrons qu'il ne compile pas. Il s'agit de :

- La mauvaise définition des SlantedRectangle : Les paramètres passés ne correspondent pas aux paramètres atttendus par les constructeurs de SlantedRectangle. (Point, double, double) ne correspond ni a (Point, Point, double), ni a (Point, double, double, double), ni à (double, double, double, double, double). **Cette erreur a 2 occurrences.**
- La méthode rotate() est une méthode propre aux SlantedRectangle et ne peut être appliquée au type rectangle. **Cette erreur a 2 occurrences. Avec le rectangle r et le rectangle t.**

Exercice n° 5

ÉNONCÉ

Est-ce que la classe Dessin définie précédemment peut contenir des rectangle inclinés ? Est ce que les méthodes surface, contains et hull de la classe Dessin fonctionnent encore correctement ?

SOLUTION

1. Dessin peut-il contenir des SlantedRectangle ?

Oui, car la classe Dessin est définie avec l'objet Rectangle, et SlantedRectangle hérite de Rectangle. Donc la classe dessin peut manipuler les SlantedRectagles.

Il a été dit que, Dessin gère un tableau de Rectangle, donc un SlantedRectangle peut être ajouté puisque SlantedRectangle hérite de Rectangle. Cependant, certaines méthodes doivent être adaptées pour bien gérer les rectangles inclinés.

2. La méthode `surface()` fonctionne-t-elle ?

✅ **Oui, car `SlantedRectangle` redéfinit `surface()`. D'ailleurs, le calcul de surface dans notre exercice et en général ne tient pas compte de l'inclinaison.**

3. La méthode `contains(Point p)` fonctionne-t-elle ?

❌ **Non, car `contains(Point)` dans Rectangle teste l'inclusion sans considérer l'angle. Cependant, SlantedRectangle contient déjà une version corrigée qui tient compte de l'inclinaison. Donc le problème peut être résolue en contextualisant le `contains()` de SlantedRectangle.**

4. La méthode `hull()` fonctionne-t-elle ?

❌ **Non, elle ne prend pas en compte les inclinaisons. Actuellement, hull() récupère simplement les min/max des coordonnées. Mais pour SlantedRectangle, il faut calculer l'enveloppe convexe du rectangle après inclinaison.**

Exercice n° 6

ÉNONCÉ

Définir une méthode String toString() dans la classe Rectangle ? Est-ce en fait une définition ou une redéfinition ? Est-il nécessaire de la redéfinir dans la classe SlantedRectangle ?

SOLUTION

1. Définition de la méthode toString()

> *La résolution est pratique*

2. Définition ou Redéfinition ?

C'est une redéfinition et non une nouvelle définition car, la méthode toString() existe déjà dans la classe Object, que toutes les classes Java héritent implicitement. Rectangle hérite de Object et redéfinit toString() pour fournir une version personnalisée.

3. Faut-il redéfinir toString() dans SlantedRectangle ?

Oui, si on veut ajouter l'angle à l'affichage.

Exercice n° 7

ÉNONCÉ

Redéfinir la méthode equals dans les classes Rectangle et SlantedRectangle.

On considère les définitions de classes suivantes

```
class A {
    void f(A o) { System.out.println("void f(A o) dans A");
                }
}

class B extends A {
    void f(A o) {
        System.out.println("void f(A o) dans B");
    }
}
```

SOLUTION

La résolution de l'exercice cet est pratique

Exercice n° 8

ÉNONCÉ

Qu'affiche le fragment de programme suivant ?

```
 A a = new A();

A ab = new B();

B b = new B();

a.f(a);

a.f(ab);

a.f(b);

ab.f(a);

ab.f(ab);

ab.f(b);

b.f(a);

b.f(ab);

b.f(b);
```

SOLUTION

Dans cet exercice, nous avons deux classes:

- Une classe `A` avec une méthode `f(A o)`
- Une classe `B` qui étend `A` et redéfinit la méthode `f(A o)`

Le programme crée différentes instances et appelle la méthode `f` avec différents paramètres. Pour déterminer ce qui s'affiche, il faut suivre les règles du polymorphisme en Java.

Analysons ligne par ligne l'exécution du fragment de code:

```
 A a = new A();      // Crée un objet de type A, référencé par
'a'
A ab = new B();     // Crée un objet de type B, référencé par
'ab' de type A
B b = new B();      // Crée un objet de type B, référencé par 'b'

a.f(a);             // L'objet 'a' appelle la méthode f avec 'a'
comme paramètre
a.f(ab);            // L'objet 'a' appelle la méthode f avec 'ab'
comme paramètre
a.f(b);             // L'objet 'a' appelle la méthode f avec 'b'
comme paramètre
ab.f(a);            // L'objet 'ab' (de type B) appelle la
méthode f avec 'a' comme paramètre
ab.f(ab);           // L'objet 'ab' (de type B) appelle la
méthode f avec 'ab' comme paramètre
ab.f(b);            // L'objet 'ab' (de type B) appelle la
méthode f avec 'b' comme paramètre
b.f(a);             // L'objet 'b' appelle la méthode f avec 'a'
comme paramètre
b.f(ab);            // L'objet 'b' appelle la méthode f avec 'ab'
comme paramètre
b.f(b);             // L'objet 'b' appelle la méthode f avec 'b'
comme paramètre
```

Voici ce qui sera affiché:

1. `a.f(a)` → "void f(A o) dans A"

2. `a.f(ab)` → "void f(A o) dans A" (car le type statique de 'ab' est A)
3. `a.f(b)` → "void f(A o) dans A" (car B est un sous-type de A)
4. `ab.f(a)` → "void f(A o) dans B" (car l'objet réel est de type B)
5. `ab.f(ab)` → "void f(A o) dans B" (car l'objet réel est de type B)
6. `ab.f(b)` → "void f(A o) dans B" (car l'objet réel est de type B)
7. `b.f(a)` → "void f(A o) dans B"
8. `b.f(ab)` → "void f(A o) dans B"
9. `b.f(b)` → "void f(A o) dans B"

Exercice n° 9

ÉNONCÉ

On ajoute maintenant à la classe B la méthode suivante

```
void f(B o) {
    System.out.println("void f(B o) dans B");
}
```

Est-ce une redéfinition ou une surcharge ? Qu'affiche alors le fragment de programme de l'exercice 8 ?

SOLUTION

C'est une **surcharge** de méthode dans la classe B, car la signature est différente de la méthode héritée `f(A o)`. La méthode `f(B o)` accepte un paramètre de type B, tandis que la méthode existante `f(A o)` accepte un paramètre de type A.

Le fragment de l'exercice 8 affichera maintenant:

1. `a.f(a)` → "void f(A o) dans A"
2. `a.f(ab)` → "void f(A o) dans A" (car le type statique de 'ab' est A)
3. `a.f(b)` → "void f(A o) dans A" (car B est un sous-type de A)
4. `ab.f(a)` → "void f(A o) dans B" (car l'objet réel est de type B)
5. `ab.f(ab)` → "void f(A o) dans B" (car le type statique de 'ab' est A)
6. `ab.f(b)` → "void f(A o) dans B" (même si l'objet est de type B, le type statique détermine la signature)
7. `b.f(a)` → "void f(A o) dans B"
8. `b.f(ab)` → "void f(A o) dans B" (car le type statique de 'ab' est A)
9. `b.f(b)` → **"void f(B o) dans B"** (ici la surcharge est utilisée car le type statique et dynamique est B)

Exercice n° 10

ÉNONCÉ

On ajoute finalement à la classe A la méthode suivante

```
void f(B o) {
    System.out.println("void f(B o) dans A");
}
```

Est-ce une redéfinition ou une surcharge ? Qu'affiche alors le fragment de programme de l'exercice 8 ?

SOLUTION

C'est une **surcharge** de méthode dans la classe A, car la signature `f(B o)` est différente de la méthode existante `f(A o)`.

Le fragment de l'exercice 8 affichera maintenant:

1. `a.f(a)` → "void f(A o) dans A"
2. `a.f(ab)` → "void f(A o) dans A" (car le type statique de 'ab' est A)
3. `a.f(b)` → **"void f(B o) dans A"** (ici la surcharge est utilisée)
4. `ab.f(a)` → "void f(A o) dans B" (redéfinition utilisée)
5. `ab.f(ab)` → "void f(A o) dans B" (car le type statique de 'ab' est A)
6. `ab.f(b)` → **"void f(B o) dans B"** (surcharge dans B, car B redéfinit aussi f(B o))
7. `b.f(a)` → "void f(A o) dans B"
8. `b.f(ab)` → "void f(A o) dans B" (car le type statique de 'ab' est A)
9. `b.f(b)` → "void f(B o) dans B"

Exercice n° 11

ÉNONCÉ

Qu'affiche le fragment de programme suivant ?

```
 System.out.println(a instanceof A);

System.out.println(ab instanceof A);

System.out.println(b instanceof A);

System.out.println(a instanceof B);

System.out.println(ab instanceof B);

System.out.println(b instanceof B);
```

SOLUTION

L'opérateur `instanceof` vérifie si un objet est une instance d'une classe ou d'une de ses sous-classes.

1. `a instanceof A` → `true` (a est une instance de A)
2. `ab instanceof A` → `true` (ab est une instance de B, qui est une sous-classe de A)
3. `b instanceof A` → `true` (b est une instance de B, qui est une sous-classe de A)
4. `a instanceof B` → `false` (a est une instance de A, pas de B)
5. `ab instanceof B` → `true` (ab est une instance de B)
6. `b instanceof B` → `true` (b est une instance de B)

Exercice n° 12

ÉNONCÉ

Dans la classe Rectangle a été définie une méthode boolean contains(Rectangle). Cette méthode doit-elle être redéfinie dans la classe SlantedRectangle ? Quels cas ne sont pas couverts par cette redéfinition ? On ajoute alors une méthode boolean contains(SlantedRectangle) dans les classes Rectangle et SlantedRectangle. Quels cas ne sont toujours pas couverts par ces ajouts ?

SOLUTION

Analyse de l'Exercice n°12 : contains() pour Rectangle et SlantedRectangle

1. Redéfinition de contains(Rectangle) dans SlantedRectangle

Oui, cette méthode doit être redéfinie pour SlantedRectangle car la version existante dans Rectangle ne prend pas en compte l'inclinaison.

💡 Problème avec la méthode actuelle : La méthode contains(Rectangle) de Rectangle vérifie si les coordonnées des quatre coins du rectangle à tester sont contenues dans les limites du rectangle de référence.

Cette logique ne fonctionne pas pour SlantedRectangle car

- elle ne prend pas en compte l'angle du rectangle incliné.
- Un rectangle incliné pourrait avoir ses coins en dehors du rectangle de référence même s'il est inclus en rotation.

2. Cas non couverts par cette redéfinition

Même avec cette redéfinition, certains cas ne sont pas gérés:

❌ **La méthode ne vérifie pas l'inclusion d'un SlantedRectangle dans un autre SlantedRectangle.**

❌ **La forme du rectangle incliné peut dépasser les limites du rectangle hôte sans que ses sommets sortent.**

🔷 Solution : Ajouter une méthode contains(SlantedRectangle) dans Rectangle et SlantedRectangle.

3. Ajout de contains(SlantedRectangle) dans les deux classes

✅ **Dans Rectangle, la méthode devra appliquer une transformation inverse aux coordonnées du SlantedRectangle pour tester son inclusion dans le rectangle de base.**

✅ **Dans SlantedRectangle, il faut gérer l'angle de chaque rectangle, ce qui est plus complexe.**

🚨 cas qui ne seront toujours pas couverts ?

❌ **Les situations où un SlantedRectangle est inclus partiellement mais ses sommets ne permettent pas une détection évidente.**

> ❌ **Les cas où un rectangle est inclus uniquement sous certaines rotations précises.**
> 💡 **Dans ces cas, il faudrait une approche géométrique avancée utilisant l'enveloppe convexe.**

Exercice n° 13

ÉNONCÉ

On considère les définitions de classes suivantes

```
class C {
    char ch = 'C';
    char getCh() {
        return ch;
    }
}

class D extends C {
    char ch = 'D';
    char getCh() {
        return ch;
    }
}
```

Qu'affiche le fragment de programme suivant ?

```
C c = new C(); C cd = new D();

D d = new D();

System.out.println(c.ch);

System.out.println(c.getCh());

System.out.println(cd.ch);

System.out.println(cd.getCh());

System.out.println(d.ch);

System.out.println(d.getCh());
```

SOLUTION

1. **Attributs** : L'accès aux attributs dépend du type statique de la référence (type déclaré)
2. **Méthodes** : L'appel de méthode dépend du type dynamique de l'objet (type réel)

Analysons le programme ligne par ligne pour comprendre l'affichage

```
C c = new C();     // Type statique: C, Type dynamique: C
C cd = new D();    // Type statique: C, Type dynamique: D
D d = new D();     // Type statique: D, Type dynamique: D
```

Maintenant analysons chaque affichage:

1. `System.out.println(c.ch);`

- `c` est de type statique C
- Acc