Remarques suite à première évaluation de la classe Cercle.

Contents

Bien noter que de nombreuses recommandations (indentation, pertinence des variables locales, choix de noms significatifs, etc.) ne sont pas nouvelles : elles ont déjà été faites dans les modules déjà suivis (PIM, Langage C. . .).

1. Quelques conventions :
   - Même si le langage Java n'impose pas d'ordre, la convention est de mettre d'abord les attributs, puis les constructeurs, puis les méthodes. On conseille parfois de mettre les membres de classe (static) avant les membres d'instance.
   - Les variables locales, les paramètres, les attributs, les méthodes ont des noms qui commencent par une minuscule (centre, rayon, leCentre, translater, getCentre. . .) !
   - Les types que l'on définit (classe, interface, type énuméré) commencent par un majuscule (String, Math, Comparateur, Point, PointCartesien. . .).
   - Les constantes de classe s'écrivent toutes en majuscules (PI, MAX, TAILLE_MAX. . .).
   - On utilise une majuscule pour mettre en évidence les morceaux d'un identifiant composé de plusieurs mots (getCentre, nouvelleCouleur, lAgeDuCapitaine. . .) sauf pour les constantes où le souligné est utilisé (TAILLE_MAX. . .).
   - On met un espace avant et après les opérateurs binaires.
2. Attention à l'indentation !
3. Laisser au moins une ligne vide entre deux méthodes ou constructeurs.
4. Ne pas confondre variable locale et attribut (voire paramètre et attribut). Une information qui n'est utile qu'au sein d'une même méthode (ou constructeur) est une variable locale (ou un paramètre) et non un attribut. Un

attribut est une information qui stocke l'état d'un objet, au moins entre deux appels de méthodes, généralement sur la durée de vie d'un objet.

5. Utiliser des variables locales pour gagner en lisibilité :

```
    this.centre = new Point( (point1.getX() + point2.getX())/2, (point1.getY() + point2
==>
    double cx = (point1.getX() + point2.getX()) / 2;
    double cy = (point1.getY() + point2.getY()) / 2;
    this.centre = new Point(cx, cy);
```

6. Mais attention, TROP de variables locales peut nuir à la lisibilité !

```
    double x1 = c.getX();
    double x2 = d.getX();
    double y1 = c.getY();
    double y2 = d.getY();
    double x = (x1 + x2)/2;
    double y = (y1 + y2)/2;
    this.centre = new Point(x,y);
==>
    double x = (c.getX() + d.getX()) / 2;
    double y = (c.getY() + d.getY()) / 2;
    this.centre = new Point(x,y);
```

   Est-ce que x1, x2, y1 et y2 sont vraiment utiles ? Quel gain ?
   Au passage, est-ce que c et d sont des noms significatifs ?

7. Éviter les variables locales inutiles :

```
    {
        Point pt = new Point (this.centre.getX(), this.centre.getY());
        return pt;
    }
==>
    {
        return new Point(this.centre.getX(), this.centre.getY());
    }
```

   La variable locale pt ne sert à rien (elle permet de nommer le résultat, mais le nom de la méthode le fait déjà) ! Autant la supprimer !

8. Noms peu significatifs : point1, point2 ou p1, p2 en paramètre de creerCercle !

9. Un identifiant à une seule lettre (r, c, etc.) est une mauvaise idée. Il faut prendre des noms **significatifs**. Comme souvent, il y a des exceptions à la règle, en particulier quand une convention est bien admise. C'est le cas de x et y sur les points qui sont utilisés pour l'abscisse et l'ordonnée. Il faut le considérer comme un contre-exemple !

10. Est-ce que les noms choisis pour les paramètres de creerCercle sont significatifs ?

11. Si possible, ne déclarer une variable que quand on sait l'initialiser.

```
    double r;
    r = p1.distance(p2);
==>
    double r = p1.distance(p2);
```

12. Éviter les parenthèses inutiles :

```
return (this.rayon) * (this.rayon) * PI;
    ==> return this.rayon * this.rayon * PI;

this.rayon = (point1.distance(point2))/2;
    ==> this.rayon = point1.distance(point2) / 2;

(this.centre).getX()
    ==> this.centre.getX()
```

13. Éviter les if else inutiles (et gagner en lisiblité) :

```
    if (condition) {
        return true;
    } else {
        return false;
    }
==>
    return condition;
```

    La reformulation est équivalente... et plus lisible !

14. assert (comme return) est un opérateur du langage et non une méthode. Il n'est donc pas nécessaire de mettre des parenthèses.

```
    assert(p != null)
==>
    assert p != null;
```

15. La précondition "this != null" (et donc assert this != null;) est inutile car this ne peut pas avoir la valeur "null" dans une méthode d'instance ou un constructeur. C'est une précondition garantie par le langage Java :

    - si le récepteur d'une méthode vaut "null", on aura NullPointerException et la méthode ne sera pas appelée,
    - si new n'a pas réussi à allouer de la mémoire pour l'objet, on aura l'exception OutOfMemoryError et le constructeur ne sera pas exécuté.

16. L'appel explicite à toString() est généralement inutile :

```
    return "C" + this.rayon + "@" + this.centre.toString();
==>
    return "C" + this.rayon + "@" + this.centre;
```

Justification : c'est plus concis et ça marchera même si 'centre' est 'null'. La version initiale provoquera NullPointerException si 'centre' est 'null'.

17. Éviter la redondance de code entre constructeurs : Si un constructeur contient les mêmes instructions qu'un autre, on peut certainement (et on doit) utiliser this(...).
18. Utiliser les méthodes de Point et non réécrire leur code dans Cercle.
19. Plus généralement, il faut utiliser les méthodes existantes plutôt que de réécrire leur code.
20. Un moyen de factoriser du code (plusieurs instructions proches à des endroits différents) est de définir une nouvelle méthode.
21. Vérifier que tous les commentaires de documentation (javadoc) sont donnés. Il en faut au moins un pour la classe et un pour chaque méthode. Pour les méthodes, il faut utiliser @param, @return. Le commentaire doit être pertinent : expliquer la méthode, le paramètre, la valeur de retour...
22. Pour les tests unitaires, il faut faire une méthode de test par exigence à tester (ou par méthode à tester). On doit faire plusieurs méthodes de test pour une même exigence (même méthode), si plusieurs aspects sont à tester.
23. Les tests réalisés par les classes de test fournies ne sont pas exhaustifs. Vous avez donc certainement des tests supplémentaires à faire que vous devez mettre dans ComplementsCercleTest. Mais cette classe peut rester vide. Il n'y aura pas de pénalité. En revanche, si elle est définie elle doit réussir sur votre classe Cercle.
24. L'orthographe peut être prise en compte dans la notation.