

Author : Christophe Garion <garion@isae.fr>
Public : SUPAERO 2A
Date :



Résumé

Le but de ce TP est de manipuler les notions de classe et d'objet au travers de la classe Point vue en cours et de l'implantation de la classe Orbite.

1 Contenu

Ce corrigé succinct contient les réponses au TP numéro 1. Le corrigé de la partie implantation en Java est disponible sur <http://www.tofgarion.net/lectures/IN201>.

2 Manipulation de la classe Point

L'objectif de cette section est de manipuler la classe Point déjà vue en cours. On ne dispose que du *bytecode* de Point (dans le répertoire classes de votre TP) et de sa documentation Javadoc sur le site du cours. Le code source de la classe sera mis à votre disposition sur le site lors de la publication du corrigé de ce TP.

1. créer une classe TestPoint pouvant être interprétée. La compiler et vérifier qu'elle peut être interprétée.

Solution :

Il suffit de déclarer correctement la classe et de ne pas se tromper dans la signature de la méthode main :

```
class TestPoint {  
  
    public static void main(String[] args) {  
  
    }  
}
```

2. créer un nouveau point de coordonnées (2,3) affecté à une référence p et l'afficher.

Solution :

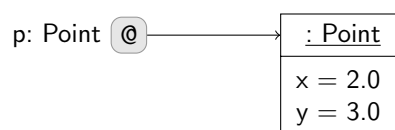
Rien de bien difficile ici, voici les instructions nécessaires :

```
Point p = new Point(2.0, 3.0);  
  
System.out.println("p apres creation : " + p);
```

Quelques remarques :

- ces lignes de code sont bien évidemment placées dans la méthode main de TestPoint.
- on peut tout à fait *déclarer* sur une ligne le point p et lui *affecter* sur une autre ligne l'objet Point de coordonnées (2,3).
- lorsque des paramètres ont un type réel (**float** ou **double**), j'utilise explicitement des constantes de ce type (d'où l'utilisation de 2.0 et 3.0 dans la construction du point). Vous pouvez bien sûr utiliser dans ce cas des constantes entières (donc faire un appel à `new Point(2, 3)`), la conversion étant sans perte et étant faite automatiquement par javac.
- pour afficher le point, il suffit de lire la documentation de la classe Point pour se rendre compte que l'existence d'une méthode particulière, `toString`, nous permet de passer directement en paramètre de `System.out.println` la référence p.

L'état de la mémoire est représentée par la figure suivante :



On y voit l'objet de type Point et sa référence p.

3. traduire le point référencé par p d'un vecteur $(-5, 3)$ et afficher le point.

Solution :

Voici les lignes correspondantes :

```
p.translater(-5.0, 3.0);
System.out.println("p apres translation : " + p);
```

Rien de bien particulier, l'objectif de la question était de vous faire appeler une méthode sur une référence. L'état de la mémoire est maintenant :

L'état de la mémoire est représentée par la figure suivante :



4. déclarer une nouvelle référence q de type `Point` et lui affecter la référence p . Traduire l'objet référencé par q d'un vecteur $(1, 1)$ et afficher les objets référencés par p et q . Que se passe-t-il ? Pourquoi ?

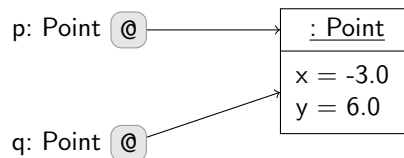
Solution :

Voici les lignes correspondantes :

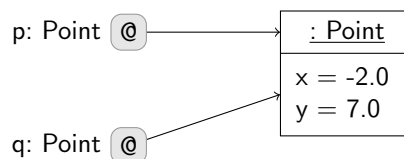
```
Point q = p;
q.translater(1.0, 1.0);
System.out.println("p apres translation de q : " + p);
System.out.println("q apres translation : " + p);
```

On remarque après exécution que le point référencé par p a été traduit alors que l'on n'a traduit « que » q . Tout cela est parfaitement normal :

- lors de la déclaration de q , on lui affecte p . Les deux références p et q référencent donc le même objet, l'état de la mémoire est donc :



- lorsque l'on appelle une méthode sur une référence, la méthode est appelée sur l'objet référencé. Après la translation de q , l'état mémoire est donc :



Lorsque l'on affiche le point référencé par p , on a donc bien les nouvelles coordonnées.



La seule façon de créer un objet en mémoire avec Java est d'appeler l'opérateur **new**. L'affectation directe de références ne crée pas de nouvel objet en mémoire.

5. utiliser maintenant une méthode disponible dans l'API¹ de `Point` pour copier un point et affecter le point copié à une

1. API : *Application Programming Interface*, les services pouvant être utilisés depuis l'extérieur d'un module, d'une classe etc. Dans le monde

référence `r`. Afficher les objets référencés par `p` et `r`. Translater l'objet référencé par `r` d'un vecteur (2,2) et afficher les objets référencés par `p` et `r`. Que se passe-t-il ? Pourquoi ?

Solution :

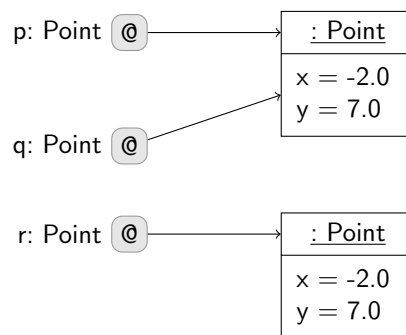
Il fallait utiliser la méthode `clone` disponible dans la classe `Point`, elle était facilement trouvable grâce à la documentation javadoc de la classe. Le code est le suivant :

```
Point r = p.clone();

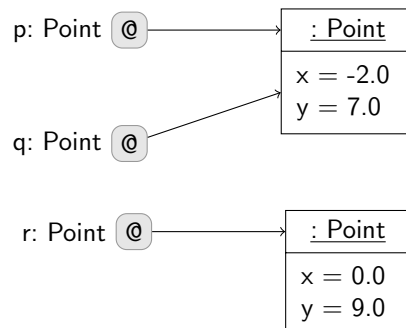
System.out.println("p apres copie : " + p);
System.out.println("r apres copie : " + r);

r.translater(2.0, 2.0);
System.out.println("p apres translation de r : " + p);
System.out.println("r apres translation de r : " + r);
```

`clone` renvoie un *nouvel* objet qui possède les mêmes coordonnées que l'objet initial. L'objet référencé par `r` est donc *différent* de celui référencé par `p` et `q`. L'état mémoire après l'appel à `clone` sera donc le suivant :



L'opération de translation ne s'applique qu'à l'objet référencé par `r`, l'état mémoire après translation de `r` sera donc :



6. copier de nouveau `p` dans la référence `r`. En utilisant l'opérateur `==`, afficher le résultat de la comparaison entre `p` et `q` et entre `p` et `r`. Que se passe-t-il ? Pourquoi ?

Solution :

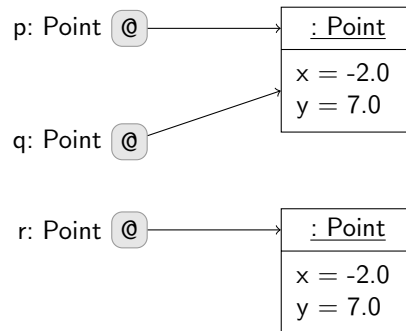
Le code est le suivant :

```
r = p.clone();

System.out.println("p == q ? " + (p == q));
System.out.println("p == r ? " + (p == r));
```

On remarquera tout d'abord que l'on peut réaffecter une variable de type poignée comme on pourrait le faire avec une variable contenant un entier. L'état mémoire après l'appel à `clone` est le suivant :

Java, l'API d'une classe est documentée via Javadoc.



Lorsqu'il est utilisé avec des poignées, l'opérateur `==` compare l'égalité *physique* des références sous-jacentes. Il ne renvoie **true** que si les deux poignées réfèrent le même objet en mémoire. Le résultat est donc :

- **true** pour p et q qui réfèrent le même objet
- **false** pour p et r qui réfèrent des objets différents, *même si ceux-ci représentent des points avec des coordonnées identiques*.

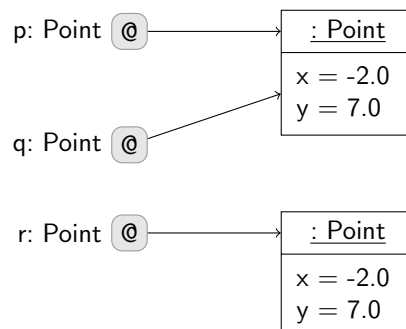
7. trouver dans l'API de `Point` une méthode permettant de comparer *logiquement* (i.e. en utilisant l'état des objets) deux points et refaire les comparaisons précédentes. Que se passe-t-il ? Pourquoi ?

Solution :

La méthode permettant de comparer deux points en utilisant leurs coordonnées et non pas une égalité physique est `equals`. Le code s'écrit donc :

```
System.out.println("p.equals(q) ? " + (p.equals(q)));
System.out.println("p.equals(r) ? " + (p.equals(r)));
```

L'état mémoire est toujours le même :



`equals` renverra **true** pour les deux comparaisons, car les objets comparés (le point référencé par p avec lui-même, le point référencé par p avec celui référencé par r) ont des coordonnées identiques. On peut facilement vérifier que si l'on translate r, `p.equals(r)` renverra **false**.

Nous verrons dans le cours sur l'héritage que `clone` et `equals` sont des méthodes importantes de la classe `Object`. Vous pourrez regarder la documentation Javadoc de la classe `Object` pour plus de détails.

3 Implantation d'une classe `Orbite` pour vol en formation

1. à partir du diagramme UML, écrire le squelette de `Orbite` et vérifier qu'il compile.

Solution : Rien de compliqué, il fallait juste « transformer » le diagramme UML en code Java. Il ne fallait pas oublier de mettre une instruction **return** lorsque les méthodes n'avaient pas **void** pour type de retour. On renvoie alors la valeur par défaut du type concerné (0, 0.0, **null**).

2. écrire les constructeurs de `Orbite`, ainsi que les accesseurs et modifieurs définis dans le diagramme UML. Que se passe-t-il lorsque l'on translate le point ayant servi à construire l'ellipse ? Est-ce correct ?

Solution :

Il fallait se demander s'il fallait copier le point passé en paramètre du constructeur. J'ai choisi de le copier (cf. listing 1) pour pouvoir respecter le principe d'encapsulation. En effet, si l'on affecte directement le point foyer passé

en paramètre du constructeur, on va pouvoir modifier le foyer de l'ellipse *directement depuis l'extérieur de la classe* via la référence passée au constructeur. De la même façon, j'ai écrit l'accessor et le modifieur de foyer en utilisant clone (cf. listing 2).

Listing 1– Orbite.java (constructeur)

```

36  /**
37   * Créer une nouvelle instance de <code>Orbite</code>.
38   *
39   * @param e valeur de l'excentricite de l'orbite. <b>Doit etre comprise
40   *         entre 0 et 1</b>
41   * @param a valeur du demi-grand axe de l'ellipse. <b>Doit etre positive</b>
42   * @param foyer un point representant le foyer de l'ellipse sur
43   *             lequel est situe la Terre. Attention, ce point n'est pas copie.
44   */
45  public Orbite(double e, double a, Point foyer) {
46      this.e = e;
47      this.a = a;
48      this.foyer = foyer.clone();
49      this.calculerB();
50  }

```

Listing 2– Orbite.java (accessor et modifieur de foyer)

```

128 /**
129  * Le point ou est situe la Terre.
130  *
131  * @return une reference vers le point representant la Terre. Attention,
132  *         le point n'est pas copie.
133  */
134  public Point getFoyer() {
135      return this.foyer.clone();
136  }
137
138  /**
139  * Changer le point representant la Terre.
140  *
141  * @param foyer le nouveau point representant la Terre
142  */
143  public void setFoyer(Point foyer) {
144      this.foyer = foyer.clone();
145  }

```

On pouvait se poser la question de la vérification des paramètres du constructeur. En effet, *a* doit être positif et *e* compris entre 0 et 1. Nous verrons plus tard dans le cours que l'on peut utiliser le mécanisme d'exception pour cela. Pour l'instant, je me suis contenté de le préciser dans la documentation Javadoc, cela devrait être suffisant (ou pas...). *calculerB* est une méthode privée, elle peut être appelée depuis la classe *Orbite* mais pas depuis l'extérieur. Elle est utilisée dans les constructeurs et dans les modifieurs de *a* et de *e*. On pouvait se demander comment la tester depuis *TestOrbite* puisqu'elle est privée. Comme *calculerB* ne renvoie rien et qu'elle ne sert qu'à mettre à jour l'attribut *b*, on peut la tester indirectement à travers l'accessor de *b*.

J'ai utilisé dans le programme de test représenté par la classe *TestOrbite* une méthode *statique*² *testEgalite* me permettant d'écrire plus rapidement le code (cf. listing 3). Vous remarquerez que j'utilise EPS pour comparer les valeurs réelles.

Listing 3– TestOrbite.java (méthode testEgalite)

```

65  private static void testEgalite(double attendu, double reel, String message) {
66      System.out.print(message + " : ");
67
68      if (Math.abs(attendu - reel) < Orbite.EPS) {

```

```

69     System.out.println("OK");
70     } else {
71         System.out.println("ERREUR! attendu : " + attendu + ", obtenu : " + reel);
72     }
73     }

```

Les tests que j'ai faits sur le constructeur et les accesseurs sont assez basiques, il en manque.

3. écrire les méthodes `clone`, `equals` et `toString`. Pour la méthode `toString`, on utilisera les paramètres de l'ellipse et les coordonnées de son foyer par exemple.

Solution :

Rien d'original, j'ai utilisé les mêmes principes que pour `Point`.

4. écrire la méthode permettant de calculer un point à partir de θ . On utilisera la documentation de la classe `Math` (cf. [2]) pour trouver les fonctions trigonométriques et autres nécessaires.

Solution :

La méthode n'était pas très compliqué à écrire, restait à la tester. Je propose un premier test simple basé sur la définition géométrique d'une ellipse (somme des distances au foyer constante), voir le listing 4 pour la vérification pour l'orbite GPS. On peut également vérifier que le périhélie et l'apogée des deux ellipses « test » sont correctes. Il faudrait également faire ces tests pour des orbites avec un foyer qui n'est pas situé à l'origine. On peut également vérifier que l'orbite GPS est un cercle.

Vous vous apercevez évidemment que la vérification manuelle du test est très fastidieuse, voire impossible dans mon cas. Nous verrons lors de la prochaine séance un *framework* de test pour Java, JUnit [1], qui permettra de tester plus facilement des méthodes.

Listing 4– TestOrbite.java (tests simples pour le calcul d'un point)

```

21     double c = gps.getC();
22
23     double distance = 2 * (gps.getA() - c) + 2 * c;
24
25     Point foyer1 = gps.getFoyer();
26     Point foyer2 = new Point(foyer1.getX() - 2 * c, foyer1.getY());
27
28     Point p = null;
29
30     for (double theta = 0; theta < 2 * Math.PI; theta += 10E-6) {
31         p = gps.calculerPointSurOrbite(theta);
32         testEgalite(distance, p.distance(foyer1) + p.distance(foyer2),
33             "Test orbite GPS avec angle " + theta);
34     }

```

5. écrire la méthode permettant de calculer un point à partir de v .

Solution :

La méthode `calculerPointOrbiteFoyer` n'était pas compliquée à écrire une fois que l'on avait écrit `calculerTheta` (cf. listing 5). Par contre, `calculerTheta` étant privée, on ne peut pas la tester dans `TestOrbite`³. On peut refaire les mêmes tests que pour `calculerPointOrbite`, cela devrait fonctionner (mais cela reste toujours aussi fastidieux...). J'ai utilisé la méthode `atan2` de la classe `Math` qui renvoie un angle compris entre $-\pi$ et π .

Listing 5– Orbite.java (méthode `calculerPointOrbiteFoyer`)

```

226     public Point calculerPointSurOrbiteFoyer(double v) {
227         return this.calculerPointSurOrbite(this.calculerTheta(v));
228     }

```

6. écrire la méthode permettant de trouver le vecteur tangent en un point de l'ellipse. On utilisera la classe `Point` pour représenter un vecteur. Pour simplifier les calculs, on se ramènera à une ellipse dont le centre est à l'origine. On fera attention au signe de l'angle à l'origine et aux cas limites.

Solution :

C'était une question difficile, à la fois du point de vue de l'implantation et du point de vue du test. La méthode est présentée sur le listing 6. Il fallait faire attention :

- aux cas limites;
- à bien se ramener à une ellipse centrée sur l'origine;
- à bien trouver le signe de la composante en x du vecteur, ce qui pouvait être compliqué, sauf si on utilisait `atan2`. J'ai utilisé l'opérateur ternaire `? :` de Java pour éviter une conditionnelle avec `if`, mais cela peut nuire à la lisibilité du code;
- à normaliser le vecteur pour éviter les problèmes d'arrondis dus aux nombres utilisés. J'ai introduit une constante privée `ORIGINE` pour représenter le point origine.

Je propose une vérification simple dans `TestOrbite` en utilisant le fait que l'orbite GPS est circulaire (cf. listing 7), mais cela ne suffit normalement pas...

Listing 6– Orbite.java (méthode calculerVecteurTangent)

```

238 public Point calculerVecteurTangent(double theta) {
239     // on s'occupe des cas aux limites
240     if (Math.abs(theta % (2.0 * Math.PI)) < EPS) {
241         return new Point(0.0, 1.0);
242     }
243
244     if (Math.abs(theta % Math.PI) < EPS) {
245         return new Point(0.0, -1.0);
246     }
247
248     // calcul dans les autres cas
249     Point centre = new Point(this.foyer.getX() - this.getC(),
250                             this.foyer.getY());
251
252     Point p = this.calculerPointSurOrbite(theta);
253
254     // on se ramene a un centre a l'origine
255     p.translater(-centre.getX(), -centre.getY());
256
257     // on choisit un vecteur avec une abscisse qui vaut celle du
258     // demi-grand axe. Comme on sait que theta via atan2 sera
259     // compris entre -PI et PI, c'est facile...
260     double x = ((theta < 0.0) ? 1.0 : -1.0) * this.a + p.getX();
261
262     double y = (1.0 - (x * p.getX()) / (this.a * this.a)) *
263               (this.b * this.b) / p.getY();
264
265     // on normalise le vecteur
266     Point vec = new Point(x - p.getX(), y - p.getY());
267
268     double lvec = vec.distance(Point.ORIGINE);
269
270     return new Point(vec.getX() / lvec, vec.getY() / lvec);
271 }

```

Listing 7– TestOrbite.java (tests simples pour le vecteur tangent)

```

49     Point vec = null;
50
51     for (double v = 0; v < 2 * Math.PI; v += 1E-5) {
52         p = gps.calculerPointSurOrbiteFoyer(v);
53         distance = p.distance(new Point(0.0, 0.0));
54

```

```

55     p.setX(p.getX() / distance);
56     p.setY(p.getY() / distance);
57
58     vec = gps.calculerVecteurTangent(v);
59
60     testEgalite(0.0, p.getX() * vec.getX() + p.getY() * vec.getY(),
61               "Test vecteur tangent orbite GPS avec angle " + v);
62 }

```

7. implanter la méthode `voitSatelliteSuivant`. Quelques indications :

- sur l'orbite GPS les satellites sont toujours visibles ;
- sur l'orbite EXOSAT, on peut utiliser le tableau suivant pour faire les tests

v	visibilité
[0.000; 1.246]	oui
[1.247; 3.169]	non
[3.170; 3.348]	oui
[3.349; 4.276]	non
[4.277; 2π]	oui

Solution :

Encore une fois, l'implantation de la méthode était un peu fastidieuse mais pas très compliquée (cf. listing 8). On se rend compte là encore que pour tester une telle méthode, il va falloir un *framework* adapté...

Listing 8– Orbite.java (méthode `voitSatelliteSuivant`)

```

281 public boolean voitSatelliteSuivant(double v) {
282     Point sat = this.calculerPointSurOrbiteFoyer(v);
283     Point satSuivant = this.calculerPointSurOrbiteFoyer(v + Math.PI * 2.0 / 3.0);
284
285     // calcul vecteur tangent
286     // on assimile les vecteurs a des points...
287     Point vTangent = this.calculerVecteurTangent(this.calculerTheta(v));
288     double lvTangent = vTangent.distance(Point.ORIGINE);
289
290     // vecteur vers prochain satellite (normalise)
291     Point vSat = new Point(satSuivant.getX() - sat.getX(),
292                           satSuivant.getY() - sat.getY());
293     double lvSat = vSat.distance(Point.ORIGINE);
294     vSat = new Point(vSat.getX() / lvSat, vSat.getY() / lvSat);
295     lvSat = vSat.distance(Point.ORIGINE);
296
297     double cosVSatTangent = (vTangent.getX() * vSat.getX() +
298                             vTangent.getY() * vSat.getY()) /
299                             (lvTangent * lvSat);
300
301     return ((cosVSatTangent >= Math.cos(ANGLE_VISEE + OUVERTURE / 2.0)) &&
302           (cosVSatTangent <= Math.cos(ANGLE_VISEE - OUVERTURE / 2.0)));
303 }

```

Références




- [1] JUNIT TEAM. *JUnit*. 2013. URL : <http://www.junit.org>.
- [2] ORACLE. *Java API specifications*. 2013. URL : <http://docs.oracle.com/javase/7/docs/api/index.html>.

License CC BY-NC-SA 3.0



This work is licensed under the Creative Commons Attribution-NonCommercial-ShareAlike 3.0 Unported license (CC BY-NC-SA 3.0)

You are free to Share (copy, distribute and transmute) and to Remix (adapt) this work under the following conditions:

-  **Attribution** – You must attribute the work in the manner specified by the author or licensor (but not in any way that suggests that they endorse you or your use of the work).
-  **Noncommercial** – You may not use this work for commercial purposes.
-  **Share Alike** – If you alter, transform, or build upon this work, you may distribute the resulting work only under the same or similar license to this one.

See <http://creativecommons.org/licenses/by-nc-sa/3.0/> for more details.