2I002 - Introduction à la programmation Objet
TME SOLO - Système Solaire

Thomas Robert Vincent Guigue

24 novembre 2016 - Durée : 1h45

Consignes

— Pas de communication, texto ou autre : téléphones éteints et rangés
— Pas de documents
— Dans toutes les classes créées : écrire votre nom en commentaire au début du fichier et en début de nom de classe (ex : pour la classe `Etoile` de l'élève `Durant`, la classe s'appelera `DurantEtoile`)
— Soumission par mail en fin de séance à : `vincent.guigue@lip6.fr` Objet : `[2i002] <NOM>`, les sorties consoles de votre programme seront copiées dans le corps du message

Diagramme UML

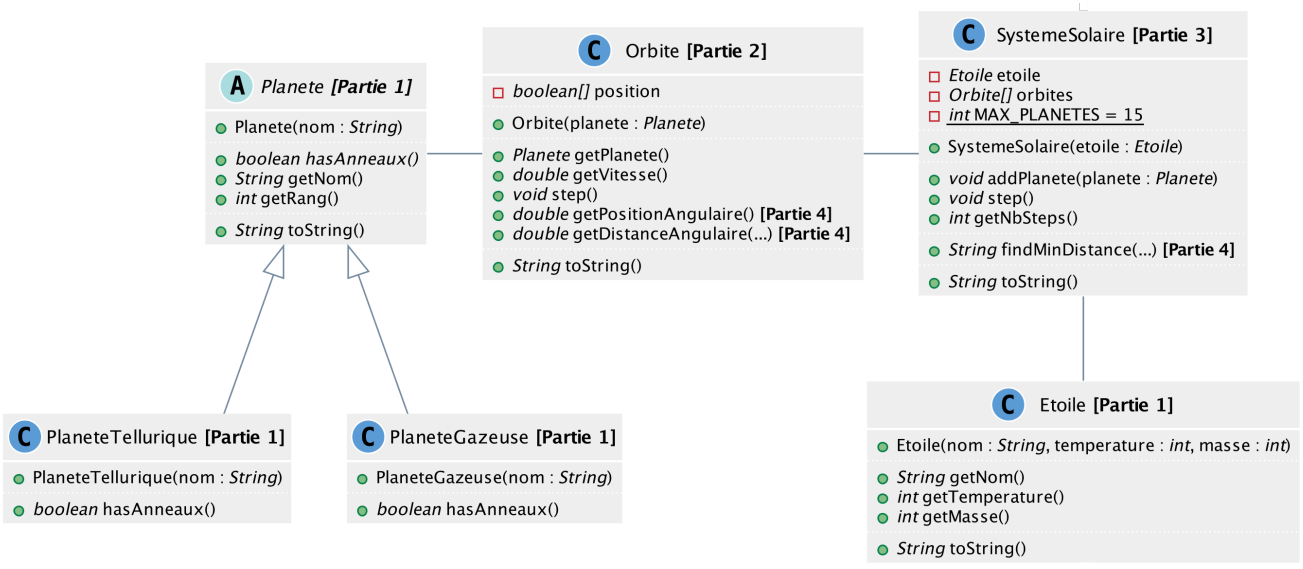

FIGURE 1 – Diagramme UML des classes à coder
Note : les méthodes publiques sont toutes indiquées, par contre seuls certains attributs privés sont indiqués, à vous de mettre dans votre code ceux dont vous avez besoin

Partie 1 – Etoile et planètes

Dans cette partie, nous allons développer les classes `Etoile`, `Planete`, `PlaneteTellurique` et `PlaneteGazeuse`. Le diagramme UML indique les méthodes publiques à développer pour chaque classe.

1. Développer puis tester la classe `Etoile`. Cette classe à :
   — un constructeur prenant les paramètres `nom`, `temperature` et `masse`
   — des *getters* pour accéder à ces variables
   — une méthode `toString()` affichera l'étoile au format `<nom>[T=<temperature>K M=<masse>]` (ex : `Soleil[T=5777K M=1]`)
2. Développer puis tester la classe **abstraite** `Planete`. Cette classe à :
   — un constructeur prenant le paramètre `nom` et le *getter* associé
   — une méthode `getRang()` permettant d'obtenir le rang de la planète. Ce rang dépend de l'ordre de création des objets de type `Planete` : le premier objet créé a le rang 1, le deuxième a le rang 2, etc.
   — la méthode `hasAnneaux()` est une méthode **abstraite**
   — `toString()` affichera la planète au format `<nom>[anneaux=<anneaux>, rang=<rang>]` (ex : `Terre[anneaux=false, rang=3]`)
3. Développer les classes `PlaneteTellurique` et `PlaneteGazeuse` qui héritent de `Planete` et qui respectivement n'ont pas et ont des anneaux.
4. Testez vos différentes classes dans une classe `TestPartie1`. Instanciez une étoile (Soleil), les planètes telluriques (Mercure, Venus, Terre, Mars) et les planètes gazeuses (Jupiter, Saturne, Uranus, Neptune).

Partie 2 – Orbite

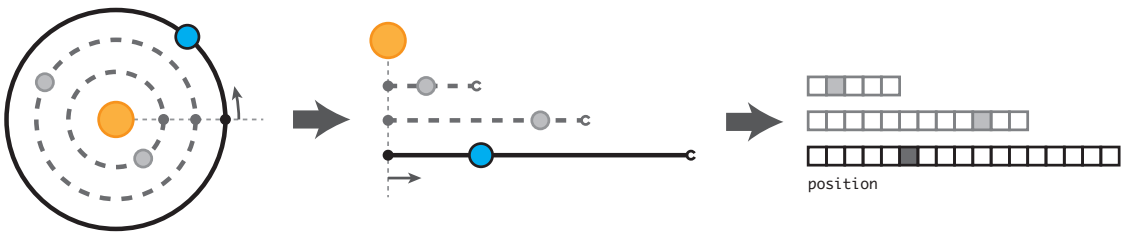

FIGURE 2 – Linéarisation et représentation des orbites

Une planète se déplace sur une orbite. Nous allons donc implémenter une classe `Orbite` pour représenter ce comportement.

Une orbite sera créée à partir d'une planète. **Le rang de la planète correspond au rayon de l'orbite.** L'orbite circulaire sera représentée par une orbite "linéarisée" (voir figure 2).

2.1 Tableau `position`

L'orbite linéarisée sera représentée par un tableau de booléen `position` indiquant où se trouve la planète. La longueur du tableau dépendra donc du périmètre de l'orbite, et donc de son rayon. Ainsi, on choisira un tableau de taille

$$n = 7 \times rayon\_orbite = 7 \times rang\_planete$$

Initialisation Toutes les cases de ce tableau seront mises à *false* à part une indiquant la position de la planète. Initialement, on placera la planète aléatoirement sur une des positions de son orbite.

2.2 Méthode `getVitesse()`

La méthode `getVitesse()` retournera la vitesse v à laquelle la planète se déplace sur son orbite. Elle dépend du rayon de l'orbite tel que :

$$v = \frac{2,8}{\sqrt{rayon\_orbite}}$$

2.3 Méthode `step()` : avancée de la planète

La méthode `step` permet de faire avancer la planète d'un pas de temps. Cette avancée correspondra à la vitesse v de la planète. Concrètement, à chaque appel à `step`, il faut donc modifier le tableau `position` afin de faire avancer la planète de v cases. On arrondira v à l'entier **le plus proche**.

Une fois arrivé à la fin du tableau, on reboucle bien sûr au début.

2.4 Méthode `getPlanete()`

Simple *getter* de la planète présente sur l'orbite.

2.5 Méthode `toString()`

La méthode `toString` affichera l'orbite sous l'une des formes suivantes (selon le cas) :

```
............o........ (planete: Terre[anneaux=false, rang=3] v=1.62)
............O..................... (planete: Jupiter[anneaux=true, rang=5] v=1.25)
```

On indiquera donc un point par case de `position` où la planète n'est pas. Par ailleurs, les planètes telluriques seront indiquées par un petit `o` alors que les planètes gazeuses seront indiquées par un grand `O`.

2.6 Testez

Testez votre code dans une classe `TestPartie2`. Reprenez votre code de `TestPartie1` instanciant les 8 planètes, puis créez des orbites pour la Terre et Jupiter. Faites avancer votre planète et affichez votre orbite un certain nombre de fois pour vérifier le bon comportement de votre classe.

Partie 3 – Système solaire

Un système solaire est constitué d'une étoile et d'un ensemble d'orbites stocké sous forme d'un tableau d'orbites `orbites`.

Il se construit à partir d'une étoile seule. Le tableaux d'orbite est donc initialisé à un tableau vide, de taille fixe définie par une **constante** `MAX_PLANETES` dont on fixera la valeur à 15. On ajoutera des orbites lors de l'appel à `addPlanete`.

3.1 Méthode `addPlanete(...)`

La méthode `addPlanete(Planete planete)` ajoutera une orbite correspond à cette planète dans la case du tableau correspondant au rang de la planète. Par exemple, on placera forcément l'orbite de la Terre dans la 3ème case du tableau.

3.2 Méthodes `step()` et `getNbSteps()`

La méthode `step` fera avancer l'intégralité des planètes du système d'un pas.

La méthode `getNbSteps` indiquera le nombre de fois où `step` a été appelée depuis l'instantiation du système.

3.3 Méthode `toString()`

La méthode `toString` affichera le numéro de l'itérations et l'étoile du système solaire suivi de ses orbites. Exemple :

```
Iteration 48  --  Soleil[T=5777K M=1MS]
....o.. (planete: Mercure[anneaux=false, rang=1] v=2.80)
...........o. (planete: Venus[anneaux=false, rang=2] v=1.98)
...........o........ (planete: Terre[anneaux=false, rang=3] v=1.62)
..................o....... (planete: Mars[anneaux=false, rang=4] v=1.40)
...........O..................... (planete: Jupiter[anneaux=true, rang=5] v=1.25)
......O................................. (planete: Saturne[anneaux=true, rang=6] v=1.14)
.............................................O (planete: Uranus[anneaux=true, rang=7] v=1.06)
.............................................O....... (planete: Neptune[anneaux=true, rang=8]
 ↪  v=0.99)
```

3.4 Testez

Testez votre code dans une classe `TestPartie3`. Reprenez votre code de `TestPartie1` instanciant les 8 planètes et le Soleil, puis créez notre système solaire. Faites avancer le système et affichez le un certain nombre de fois pour vérifier le bon comportement de votre classe.

Pour faire une pause de 0,2s entre deux itérations –afin de pouvoir observer l'évolution progressive du système– vous pouvez utiliser ce bout de code :

```
try {
    Thread.sleep(200);
}
catch(InterruptedException e) {
    e.printStackTrace();
 }
```

Partie 4 – Voyage vers Mars

Nous aimerions envoyer une fusée de la Terre vers Mars. Cependant, pour limiter la durée du voyage, nous voulons lancer notre fusée au moment où les planètes sont les plus proches. Comme nous ne sommes pas de bons très bons en maths, nous allons plutôt chercher ce moment par simulation numérique. Nous allons donc chercher à quelles itérations de notre simulation du système solaire les deux planètes sont les plus proches.

4.1 Méthode `getPositionAngulaire()` (classe `Orbite`)

La première méthode dont nous avons besoin est la méthode `getPositionAngulaire()` de la classe `Orbite`. Elle permettra de savoir quel est l'angle de la planète sur son orbite.

La position angulaire en degrés est :

$$\theta = \frac{indice\_position}{size(position)} \times 360$$

où $indice\_position$ est l'indice à vrai dans la variable `position` et $size(position)$ la longueur de `position`.

4.2 Méthode `getDistanceAngulaire(...)` (classe `Orbite`)

Dans la classe `Orbite`, ajoutez une méthode `getDistanceAngulaire` (signature exacte non fournie) qui retourne la distance angulaire entre l'orbite et une autre orbite fournie en entrée de la méthode. Notons que la distance entre deux orbites i et j est définie comme suit :

$$\begin{cases} |\theta_i - \theta_j| & \text{si } |\theta_i - \theta_j| \leq 180 \\ 360 - |\theta_i - \theta_j| & \text{sinon} \end{cases}$$

4.3 Méthode `findMinDistance(...)` (classe `SystemeSolaire`)

```
findMinDistance(int indOrbiteI, int indOrbiteJ, int nbSteps, double alpha)
```

Dans la classe `SystemeSolaire`, ajoutez une méthode `findMinDistance` qui va chercher, pendant un nombre d'itérations `nbSteps`, toutes les itérations où la distance angluaire entre les planètes des orbites i et j (entrées de la méthode) est inférieure à l'angle α donné en entrée.

La méthode retournera un rapport sous forme de chaîne de caractères. Le rapport listera toutes les itérations où la condition était satisfaite. Il aura ce format :

```
Recherche de proximité entre <planèteI> et <planèteJ>. État initial du système :

<affichage du système solaire>

Itérations intéressantes :

Itération <nIte1> : angle de <distance angulaire> degrés
Itération <nIte2> : angle de <distance angulaire> degrés
...
```

4.4 Testez

Dans une classe `TestPartie4`, en vous basant sur `TestPartie3`, affichez le rapport indiquant les meilleures itérations pour aller de la Terre à Mars. Testez le avec `nbSteps` $= 500$ et $\alpha = 3$ degrés.