L3 2024-2025

Programmation orientée objet - Java - Examen 21 Janvier 2025.

Pour cet examen, votre fond d'écran doit être vert clair. Vous devez absolument mettre tous les fichiers que vous souhaitez rendre dans le répertoire `EXAM`, qui est déjà présent sur votre compte à l'ouverture de la session. Tout ce qui ne se trouve pas dans ce dossier sera perdu lorsque vous vous déconnecterez. Ne créez pas vous-même de répertoire `EXAM`. Mettez tous vos fichiers dans celui qui existe.

Sous Eclipse, votre workspace doit être dans `EXAM`. Sinon configurez le workspace d'Eclipse (File > Switch WorkSpace) pour qu'il corresponde à un répertoire dans `EXAM`). Attention, les noms des classes, des champs et des méthodes que l'on vous demande doivent impérativement être respectés. De plus, le code doit être indenté correctement.

Commencez par vérifier la configuration d'Eclipse.

- Vérifiez que le JRE est /usr/local/apps/java23.
- Vérifez que le compilateur est java 23 avec enable preview.
- Créez ensuite votre Java project qui porte votre nom : `NOM_PRENOM`.

La javadoc 23 et les slides du cours (seuls documents autorisés) sont accessibles ici :
`http://igm.univ-mlv.fr/~juge/javadoc-23/`
`https://igm.univ-mlv.fr/~beal/JavaL3.html`

Les tests des exemples donnés dans chaque question doivent être faits (les mettre en commentaire s'ils ne marchent pas). Ces tests sont pris en compte dans la notation. Les deux exercices sont indépendants. **Dans toutes les questions, on utilisera quand c'est possible et adapté des streams et/ou des lambdas.**

Exercice 1.

Toutes les classes pour cette partie seront écrites dans un package `fr.uge.astronomy`. On écrira une méthode `main` dans une classe `Main` dans le package pour faire les tests.

On cherche à modéliser des systèmes planétaires tels que le système solaire. Un système planétaire est composé d'une étoile, de planètes, et de divers autres corps célestes gravitant autour d'une planète.

1. Écrire un record `Planet` pour représenter une planète qui gravite autour d'une étoile. Une planète contiendra trois champs : un champ `name` pour son nom (une chaîne de caractères), un champ `distance` pour sa distance à l'étoile, de type `double` en unité astronomique, et un champ booléen `habitable` qui indique si la planète est habitable. L'étoile n'est pas stockée dans la planète. Le contructeur devra tester que la `distance` est toujours positive ou nulle.

2. Ajouter une deuxième constructeur ne prenant en paramètre que le nom et la distance, et permettant de créer des planètes habitables.

3. Compléter le record de telle sorte que le code suivant fonctionne. L'affichage d'une planète se fera en indiquant son nom, suivi de "(H)" si elle est habitable, suivi de la distance à son étoile comme ci-dessous.

```
IO.println("test 1");
var earth = new Planet("Earth", 1.0);
var mars = new Planet("Mars", 1.52, false);
var neptune = new Planet("Neptune", 30.05, false);
IO.println(earth); // Earth (H) distance = 1.0 ua
IO.println(mars); // Mars distance = 1.52 ua
IO.println(neptune); // Neptune distance = 30.05 ua
```

4. Écrire un record `Satellite` pour représenter un satellite qui gravite autour d'une planète. Il aura un champ `planet` de type `Planet` pour représenter la planète autour de laquelle il gravite, un champ `name` pour un identifiant (une chaîne de caractères), un champ `distance`, de type `double`, qui est la distance en ua à sa planète.

5. Compléter le record de telle sorte que le code suivant fonctionne. L'affichage d'un satellite se fera en indiquant son identifiant, suivi de "satellite of" et du nom de sa planète, et de sa distance à sa planète comme ci-dessous.

```
IO.println("test 2");
var moon = new Satellite(earth, "Moon", 0.0025);
var triton = new Satellite(neptune, "Triton", 0.0025);
var larissa = new Satellite(neptune, "Larissa", 0.00049);
IO.println(moon); // Moon satellite of Earth distance = 0.0025 ua
IO.println(triton); // Triton satellite of Neptune distance = 0.0025 ua
```

6. Écrire une classe `PlanetarySystem` pour représenter un système planétaire. La classe contiendra un champ `star` (une chaîne de caractères) pour le nom de son étoile et un ensemble de corps célestes. Les corps celestes ne pourront être que de type `Planet` ou `Satellite`. On aura besoin d'une interface pour ces corps célestes de nom `Body`. Le champ pour l'ensemble des corps célestes sera `bodies` de type `LinkedHashSet<Body>`. Un `LinkedHashSet` est comme un `HashSet` mais il permet de garder l'ordre d'insertion pour l'affichage.

7. Ajouter une méthode `add()` qui permet d'ajouter un corps céleste au système planétaire.

8. Compléter la classe de telle sorte que le code suivant fonctionne. L'affichage d'un système planétaire donnera le nom de son étoile sur une ligne et ensuite chaque corps céleste sur une ligne.

```
IO.println("test 3");
var solarSystem = new PlanetarySystem("Sun");
solarSystem.add(earth);
solarSystem.add(mars);
solarSystem.add(neptune);
solarSystem.add(triton);
solarSystem.add(larissa);
solarSystem.add(moon);
IO.println(solarSystem);
// Sun
// Earth (H)  distance = 1.0 ua
// Mars distance = 1.52 ua
// Neptune distance = 30.05 ua
// Triton satellite of Neptune distance = 0.0025 ua
// Larissa satellite of Neptune distance = 4.9E-4 ua
// Moon satellite of Earth distance = 0.0025 ua
```

9. Écrire une méthode `private boolean isValidBody(Body body)` qui renvoie vrai si le corps celeste en argument est une planète, ou bien, si c'est un satellite, la planète autour de laquelle il gravite est bien présente dans le système planétaire. La méthode renvoie faux sinon. On utilisera le pattern matching de type de Java. Modifier la méthode `add` pour qu'elle lève une exception si on tente d'ajouter un satellite dont la planète n'est pas déjà dans le système planétaire. Sautez cette question si vous ne savez pas la faire et ne modifiez pas le `add`.

10. Écrire une méthode `List<Planet> planets()` qui renvoie la liste non modifiable des planètes du système planétaire (sans les satellites donc). La liste renvoyée devra bien être `List<Planet>`. On utilisera le pattern matching de types de Java. On pourra écrire une méthode `private static Stream<Planet> bodyAsPlanet(Body body)` qui renvoie un stream null si body est un satellite, et un stream contenant la planète si c'est une planète

(obtenu avec `Stream.of()`). Si vous ne savez pas comment renvoyer une `List<Planet>`, écrivez une méthode qui fait la même chose mais avec un type de retour `List<Body>`.

Sur l'exemple dessous, les sorties sont données en commentaire sur plusieurs lignes uniquement pour la présentation du sujet.

```
IO.println("test 4");
IO.println(solarSystem.planets());
//[Earth (H) distance = 1.0 ua, Mars distance = 1.52 ua,
//  Neptune distance = 30.05 ua]
```

11. Écrire une méthode `Planet closestToTheStar()` qui renvoie la planète la plus proche de l'étoile. La méthode lèvera une exception si le système planétaire est vide.

```
IO.println("test 5");
IO.println(solarSystem.closestToTheStar());
// Earth (H) distance = 1.0 ua
```

12. Écrire une méthode `Map<Planet, List<Satellite>> satellites()` qui renvoie une map associant à chaque planète du système solaire qui a des satellites gravitant autour d'elle, la liste des satellites qui gravitent autour de cette planète. Attention, une planète qui n'a aucun satellite gravitant autour d'elle ne sera pas une entrée de la map. Sur l'exemple ci-dessous, Mars ne figurera pas car elle n'a aucun satellite entré dans le système solaire de l'exemple (même si Mars a des lunes en réalité). On devra calculer la map en parcourant la liste `bodies` une seule fois. Sur l'exemple dessous, les sorties sont données en commentaire sur plusieurs lignes uniquement pour la présentation du sujet.

```
IO.println("test 6");
IO.println(solarSystem.satellites());
//{Neptune distance = 30.05 ua =
//   [Triton satellite of Neptune distance = 0.0025 ua,
//    Larissa satellite of Neptune distance = 4.9E-4 ua],
// Earth (H) distance = 1.0 ua=
//   [Moon satellite of Earth distance = 0.0025 ua]}
```

Exercice 2.

Faites une copier-coller du package `fr.uge.astronomy` pour créer un nouveau package `fr.uge.astronomy2`.

1. On conserve dans `fr.uge.astronomy2` les interfaces ou record `Body`, `Planete`, `Satellite`. On va changer les classes `PlanetarySystem` et `Main`.

   On souhaite changer l'implémentation des systèmes planétaires pour avoir un accès en temps quasi constant aux satellites d'une planète. Créer une classe `PlanetarySystem` avec deux champs : un champ `star` (une chaîne de caractères) pour le nom de son étoile, et un champ `satellites` de type `HashMap<Planet, Set<Satellite>>` qui indique, pour chaque planète, son ensemble de satellites. Lorsqu'une planète du système n'a pas de satellites, elle figure comme entrée de la map et son ensemble de satellite associé est vide.

2. Ajouter une méthode `add()` qui permet d'ajouter un corps céleste au système planétaire. Si un satellite est ajouté sans que sa planète soit déjà présente dans le système planétaire, le `add` ne provoque pas d'exception comme dans la partie précédente, sa planète est ajoutée à la map.

3. Ajouter une méthode `toString()` pour afficher le système planétaire. On affiche d'abord le nom de l'étoile sur une ligne, puis la map `satellites` avec une entrée de la map par ligne comme ci-dessous (ici une entrée de la map est sur plusieurs lignes uniquement pour le sujet). L'ordre d'affichage des planètes et des satellites peut être quelconque.

```
IO.println("Part 2. Test 1");
var earth = new Planet("Earth", 1.0);
var mars = new Planet("Mars", 1.52, false);
var neptune = new Planet("Neptune", 30.05, false);
var moon = new Satellite(earth, "Moon", 0.0025);
var triton = new Satellite(neptune, "Triton", 0.0025);
var larissa = new Satellite(neptune, "Larissa", 0.00049);
var solarSystem = new PlanetarySystem("Sun");
solarSystem.add(earth);
solarSystem.add(mars);
solarSystem.add(neptune);
solarSystem.add(triton);
solarSystem.add(larissa);
solarSystem.add(moon);
IO.println(solarSystem);
```

donne en sortie (sans les retours à la ligne pour l'affichage des satellites qui sont juste pour la présentation du sujet).

```
Sun
Neptune distance = 30.05 ua:
    [Larissa satellite of Neptune distance = 4.9E-4 ua,
     Triton satellite of Neptune distance = 0.0025 ua]
Mars distance = 1.52 ua: []
Earth (H) distance = 1.0 ua:
    [Moon satellite of Earth distance = 0.0025 ua]
```

Vous remarquerez que Mars figure dans la map même si elle n'a pas de satellite. Son ensemble de satellites est vide.

4. Ajouter une méthode `Set<Satellite> getSatellites(Planet planet)` qui renvoie l'ensemble des satellites d'une planète. Cet ensemble devra être non modifiable. L'ensemble renvoyé est l'ensemble vide si cette planète n'a pas de satellites et on lèvera une exception si la planète passée en argument n'est pas dans le système.

```
IO.println("Part 2. Test 2");
IO.println(solarSystem.getSatellites(neptune));
IO.println(solarSystem.getSatellites(mars));
```

donne en sortie (sans les retours à la ligne qui sont juste pour la présentation du sujet).

```
[Larissa satellite of Neptune distance = 4.9E-4 ua,
  Triton satellite of Neptune distance = 0.0025 ua]
[]
```

5. Ajouter une méthode `Set<Planet> planets()` qui renvoie l'ensemble des planètes du système planétaire.

```
IO.println("Part 2. Test 3");
IO.println(solarSystem.planets());
```

donne en sortie

```
Part 2. Test 3
[Neptune distance = 30.05 ua,
 Mars distance = 1.52 ua,
 Earth (H) distance = 1.0 ua]
```

6. Ajouter une méthode `Satellite closestToItsPlanet()` qui renvoie, parmi tous les satellites, le satellite qui est le plus proche de la planète autour de laquelle il gravite. La méthode lèvera une exception si aucune planète du système n'a de satellites.

```
IO.println("Part 2. Test 4");
IO.println(solarSystem.closestToItsPlanet());
```

donne en sortie

```
Part 2. Test 4
Larissa satellite of Neptune distance = 4.9E-4 ua
```