TP 1 : Objet, visibilité, constructeurs, associations de classes, introduction aux collections.

Indications Techniques : Le code de chaque classe doit être écrit dans un fichier dont le nom correspond à celui de la classe (e.g. le code de la classe `MyClass` sera écrit dans le fichier `MyClass.java`). Attention : le langage Java est sensible à la casse. On compile une classe par la commande `javac MyClass.java`, ce qui produit un ficher byte-code `MyClass.class`. Le programme sera exécuté avec l'interpréteur de byte-code, par la commande `java MyClass`. Si vous le souhaitez, vous pouvez utiliser une plateforme de développement comme Eclipse ou NetBeans.

Exercice 1 :
Nous allons définir une classe `Bike` représentant un vélo, défini par :

- un attribut brand décrivant la marque du vélo,
- un attribut color définissant sa couleur (on utilisera la classe `Color` du package `java.awt.Color`),
- un attribut `speed` définissant la vitessse à laquelle il roule,
- un attribut `size`, définissant sa taille en pouces.

À la création d'un vélo, sa marque et sa taille sont définis une fois pour toute, ces deux attributs ne pourront plus être modifiés.

La classe minimale...

- Écrivez une classe `Bike` (dans un fichier `Bike.java`) qui comprend tous les attributs décrits ci-dessus. Assurez-vous que les attributs `brand` et `size` ne pourront pas être modifiés. Pour le moment, ne définissez aucun constructeur. N'oubliez pas de bien définir la visibilité des différents attributs afin de respecter le principe d'abstraction.
- Définissez des constantes `DEFAULT_BRAND`, `DEFAULT_COLOR` et `DEFAULT_SIZE` et utilisez-les de manière à ce que votre code compile, toujours sans définir aucun constructeur. Ces constantes ont-elles besoin d'être stockées dans chaque instance ?
- Ajoutez la méthode `print` permettant d'afficher un vélo de la manière suivante :
  `[Electra - 26" - java.awt.Color[r=255,g=0,b=0] - (0km/h)]`
- Ajoutez une méthode `accelerate`, et une méthode `brake` permettant d'accélérer et de ralentir d'1 km/h.
- Ajoutez également une méthode `repaint` qui permet de repeindre un vélo mais uniquement de la couleur par défaut.
- Toujours dans la classe `Bike`, ajoutez une méthode `main` dans laquelle vous instancierez deux vélos `b1` et `b2`.
- Quel(s) constructeur(s) pouvez-vous utiliser ? Vérifiez les valeurs des attributs de `b1` et `b2` en utilisant la méthode `print`.
- Testez les méthodes permettant d'accélérer et de freiner, et vérifiez également que `b1` et `b2` référencent bien deux instances différentes. Pouvez-vous modifier les valeurs des attributs de vos vélos directement (i.e. sans passer par les méthodes que vous avez définies) dans la méthode main ? Vérifiez que vos attributs ont bien une visibilité private, pouvez-vous toujours modifier directement vos attributs ?
- Créez à présent une classe
- Main.java et déplacez-y la méthode

— main de la classe `Bike` et refaites les tests de la question précédente. Que constatez-vous ?
— Assurez-vous que votre vélo ne pourra jamais avoir une vitesse négative, et qu'aucun vélo ne puisse dépasser la vitesse de 60 km/h. Définissez éventuellement de nouvelles constantes.
— Dans la méthode `main`, effectuez l'instruction `b1=b2` puis affichez à nouveau `b1` et `b2`. Modifiez la vitesse de l'objet référencé par `b1` et affichez à nouveau l'objet référencé par `b1` et celui référencé par `b2`. Que s'est-il passé ? Qu'est devenu l'objet initialement référencé par `b1` ?

De nouveaux constructeurs

— Définissez un constructeur dans votre classe permettant de créer un vélo quelconque (i.e. en choisissant sa marque, sa couleur, et sa taille). Pouvez-vous toujours construire des vélos en utilisant le constructeur par défaut ?
— Ajoutez à présent un constructeur par défaut permettant de créer un vélo de la même manière que vous le faisiez dans la partie 1). Attention, une constante doit être initialisée soit à sa déclaration, soit dans les constructeurs.
— Ajoutez un troisième constructeur permettant de copier un vélo. Testez ce nouveau constructeur en vous assurant qu'un nouvel objet est bien créé à chaque appel.

On ajoute des roues

— Ajoutez une classe `Wheel`. Une roue est définie par une marque, une taille et une pression. Par défaut, la pression d'une roue est de 3 bars. Une roue peut être gonflée et dégonflée de 0.1 bar via les méthodes inflate et deflate. Ajoutez une méthode `print` permettant d'afficher une roue sous la forme :
`[Schalbe - 24" - 3 bar]`
— Enrichissez votre classe vélo en ajoutant un attribut de type tableau de roues. Par convention, la première case du tableau désignera la roue avant, et la seconde, la roue arrière. Dans cette modélisation, on supposera que chaque roue est créée par le vélo auquel elle appartient et si jamais le vélo est détruit, on veut être certain que les roues seront également détruites.
À sa création, un vélo créera sa roue avant avec une taille de 1 pouce de plus que lui, et sa roue arrière avec une taille de 1 pouce de moins. Les deux roues ayant la même marque, donnée en paramètre, et sont gonflées à la pression par défaut. Modifiez les constructeurs de la classe vélo en conséquence.
— La spécification vous permet-elle d'ajouter des accesseurs sur les roues ?
— Ajoutez une méthode permettant de supprimer une roue.
— Ajoutez une méthode permettant de remplacer une roue par une nouvelle. Pour cela on indiquera sa marque, sa taille et s'il s'agit de la roue avant ou arrière. Si la taille de la nouvelle roue a plus de 2 pouces d'écart avec la taille du vélo, alors on ne remplacera pas la roue. Sinon, l'ancienne roue est détruite et remplacée par la nouvelle.
— Ajoutez une méthode permettant de permuter les deux roues.
— Modifiez la méthode d'affichage d'un vélo de manière à avoir un affichage de la forme :

```
Bike :[Electra - 26" - java.awt.Color[r=255,g=0,b=0] - (0km/h)]
 - Front Wheel:[Schalbe - 24" - 3 bar]
 - Rear Wheel:[Schalbe - 27" - 4 bar]
```

— Ajoutez une méthode permettant de savoir combien le vélo possède de roues.
— Testez le constructeur par copie de la classe `Bike`. Assurez-vous que deux vélos ne peuvent

pas partager de roue.

Des collections de vélos

L'objectif de cette partie est de vous initier à la manipulation de collections.

— Dans le `main`, déclarez une liste de roues de la manière suivante :
`List<Wheel> wlist = new ArrayList<>();`
— Ajoutez plusieurs roues.
— Ajoutez une nouvelle roue à une position donnée.
— Pouvez-vous ajouter deux fois la même roue ?
— Affichez le nombre d'éléments que contient la liste.
— Affichez la wlist.
— En vous inspirant de la méthode `print()`, redéfinissez la méthode `toString` de la classe `Wheel` comme vu en cours. Affichez à nouveau `wlist`, que constatez-vous ?
— Ajouter à présent un ensemble de Roues de la manière suivante :
`Set<Wheel> wset = new HashSet<>();`
— Vérifiez que vous ne pouvez pas ajouter deux fois la même roue

Exercice 2 :

L'objectif est d'implanter une version de l'exercice sur les robots qui évoluent dans un monde, dont la conception a été discutée en TD.

Un monde (plutôt hostile) est un espace 2D à coordonnées entières (i.e. chaque position du monde est définie de manière unique par un couple d'entiers (x, y)). Bien entendu, un monde est un espace fini, on supposera pour simplifier qu'ils ont tous la même taillée carrée. (qui sera donc stockée dans une constante static). Chaque monde connait l'ensemble des robots qu'il contient et sait où ils sont positionnés. deux robots ne peuvent pas se trouver sur la même position.

Un robot sera uniquement caractérisée par son nom.

— Définissez la classe `Location` contenant deux attributs constants publics `X` et `Y` permettant de représenter une position (X, Y).
— Définissez la classe `Robot` permettant de créer un `Robot` avec un nom choisi.
— Redéfinissez la méthode `toString` de la classe Robot permettant d'afficher son nom.
— Définissez à présent la classe `World` représentant un monde dans lequel évolue un ensemble de robots. Un monde stocke l'ensemble de ses robots et les positions à laquelle ils se trouvent dans l'attribut `robots`. Pour cela on utilisera un dictionnaire (i.e. un tableau associatif permettant de manipuler des couples `(clef, valeur)` où chaque clef est unique). La déclaration d'un dictionnaire ayant pour clefs des objets de type TKEY et pour valeurs des objets de type TVAL se fait ainsi : `Map<TKEY, TVAL> dict = new HashMap<>();`. Les opérations de manipulation des dictionnaires sont décrites sur l'API Java disponible en ligne. On utilisera un dictionnaire dont les clés sont des `Location`, et les valeurs des `Robot`.
Un `Robot` ne connait donc ni le monde dans lequel il se trouve, ni sa position, mais le monde connait la liste des `Robots` qu'il contient et sait où chacun se trouve.
— Ajouter une constante static `ALL_LOCS` permettant de stocker la liste de toutes les positions. Pour cela, vous utiliserez un bloc `static`(cf. le cours) afin de créer toutes les Location et les ajouter à `ALL_LOCS`.

— Ajoutez un attribut `FREE_LOCS` permettant de stocker l'ensemble des positions libres du monde. A la construction d'un monde, cet attribut contient l'ensemble des locations de `ALL_LOCS`. Afin de ne pas dupliquer les positions (mais seulement les références), on utilisera le constructeur par copie des ArrayList (en copiant `ALL_LOCS`).
— Ajoutez la méthode `static Location pickLocation()` permettant de choisir une position aléatoire dans le monde. Pour cela, on utilisera la méthode `Collections.shuffle` permettant de mélanger une liste. Il suffira ensuite de simplement récupérer le premier élément de la liste.
— Ajoutez de la même manière les méthodes `pickFreeLocation()` et `pickRobotLocation()` permettant de choisir respectivement une position aléatoire libre, et une position aléatoire occupée dans le monde. Pour la seconde, vous utiliserez la méthode permettant de récupérer l'ensemble des clés d'un dictionnaire (à vous de trouver sur l'API Java).
— Ajoutez la méthode `boolean hasRobotAtLocation(Location)` permettant de savoir si une position est libre ou non.
— Ajoutez la méthode `shootAt(Location l)` permettant au monde de détruire l'emplacement désigné par l. Attention, cette position n'existe plus dans le monde en question, s'il y avait un robot, dommage pour lui...
— Ajoutez la méthode `moveRobot(Location from, Location dest)`. Si un robot se trouve en position `from`, alors il est déplacé en position `dest` (sauf si la position n'est plus valide), si un autre robot s'y trouvait, dommage pour lui, il est détruit...
— Ajoutez la fonction `display()` affichant chaque robots d'un monde et sa position associée.
— Testez votre monde en le faisant évoluer avec quelques Robots. Vous pouvez ajouter dans la méthode précédente l'affichage de l'ensemble des positions libres.