Université Lille 1 – Licence 2 Informatique 2016-2017

UE Programmation Orientée Objet

Devoir Surveillé

lundi 15 mai 2017 – 8h–10h

Copie des diapositives de cours annotées

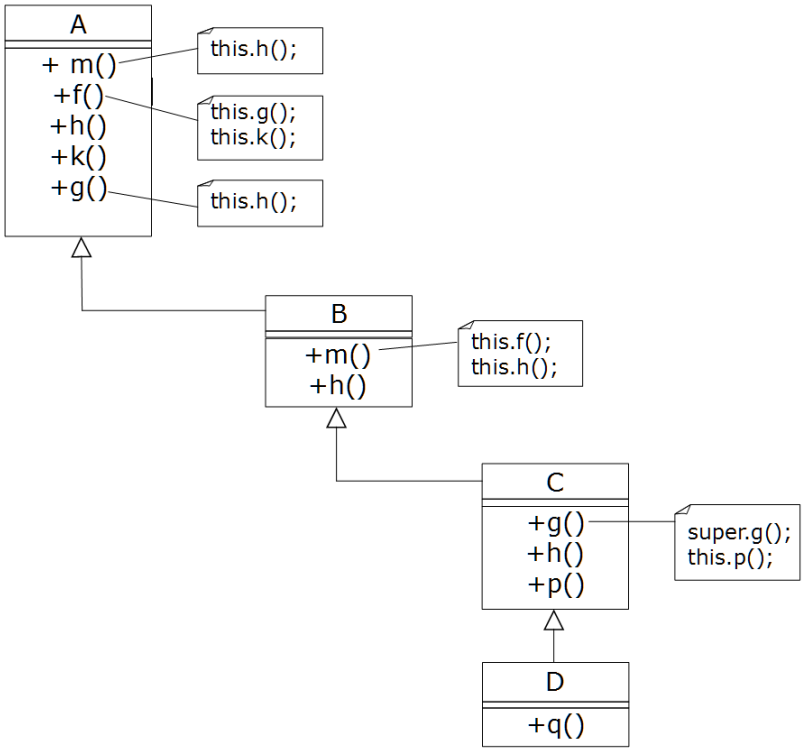
Dictionnaire de langue (papier ou électronique « dédié ») autorisé

Tout autre document interdit

Les exercices sont indépendants. Leur ordre ne préjuge pas de leur difficulté. Il est conseillé de lire entièrement un exercice avant de le traiter.
La javadoc et les méthodes de test ne sont à fournir que si elles sont demandées.
Les durées indiquées sont données à titre indicatif et prennent en compte le temps de lecture du sujet.

Exercice 1 : Lookup (15 mn)

On donne le diagramme de classes suivant :



En plus des portions de code indiquées sur le diagramme, chacune des méthodes commence par l'instruction : « `System.out.println("`*`NomDeClasse.nomMéthode`*`");` » où *NomDeClasse* est évidemment remplacé par le nom de la classe dans laquelle le corps de la méthode est codée et *nomMéthode* par le nom de cette méthode[1].

Q 1 . Indiquez précisément ce qu'affiche le programme suivant :

```
public static void main(String[] args) {
    List<A> listeA = new ArrayList<A>();

    listeA.add(new B());
    listeA.add(new D());

    for(A a : listeA) {
       System.out.println("----------------------");
       a.m();
    }
}
```

[1] L'exécution de « `new A().k()` » produit donc l'affichage « `A.k` »

Exercice 2 : VLille (55 mn)
On s'intéresse aux V'Lille, vélos en location à Lille.

Q 1 . (9 mn) Un V'Lille est représenté par une instance de la classe `VLille`. Un tel objet est caractérisé par un identifiant unique (une chaîne de caractères) et sa date de mise en circulation (du type `java.util.Date`). Ces informations sont fournies à la construction de l'objet.

La classe `lille.vlille.VLille` propose des accesseurs ainsi que les méthodes usuelles `toString` et `equals`. Deux objets `VLille` sont considérés égaux s'ils ont même identifiant.

Q 1.1. Donnez le **diagramme UML** complet et détaillé de la classe `VLille`.

Q 1.2. Donnez le **code java** de la classe `VLille`.

Il est possible de louer les V'Lille dans une station. Les stations sont représentées par des objets de la classe `DockingStation` du paquetage `lille`. Ces objets sont caractérisés par un nom (une chaîne de caractères), le nombre maximum de vélos que peut accueillir la station (un entier) et enfin la liste des V'Lille disponibles à la location pour cette station.

Q 2 . (5 mn) Donnez le **code java** de l'entête de la classe `lille.DockingStation` ainsi que la déclaration de ses attributs et de son constructeur sachant qu'initialement la liste des V'Lille disponibles est vide, les autres informations sont fournies au constructeur.

Pour la suite on suppose que pour chacun des attributs existe l'accesseur correspondant dont le nom est, classiquement, de la forme `get`*`Field`*`()` (pour l'attribut de nom *`field`*).

Q 3 . On considère les méthodes suivantes de la classe `DockingStation` :

- ▷ `takeBike` qui permet d'obtenir l'un (quelconque) des vélos disponibles à la location s'il y en a, il est alors fourni en résultat, dans le cas contraire une exception `EmptyStationException` est levée.
- ▷ `returnBike` qui prend en paramètre un V'Lille et permet de rendre ce vélo à cette station s'il reste de la place. Dans ce cas ce vélo devient donc disponible à la location dans cette station, dans le cas contraire une exception `FullStationException` est levée.

Q 3.1. (3 mn) Donnez le **code java** de la classe `lille.EmptyStationException` dont le constructeur prend en paramètre le nom de la station.

Pour la suite on supposera que la classe `lille.FullStationException` est définie de manière analogue.

Q 3.2. (8 mn) Donnez le **code des méthodes de test unitaire** JUnit qui permettent de tester le bon fonctionnement des deux méthodes `takeBike` et `returnBike` (seul le code des méthodes de test est demandé).

Q 3.3. (8 mn) Donnez le **code javadoc** et le **code java** de chacune des méthodes `takeBike` et `returnBike`.

Pour la suite on suppose que la classe `DockingStation` dispose également de la méthode `numberOfAvailableBikes()` qui fournit le nombre de vélos actuellement disponibles dans cette station.

Q 4 . (7 mn) On suppose que `station1` et `station2` représentent deux références de type `DockingStation` initialisées (c'est-à-dire non `null`) et auxquelles on a déjà ajouté au moins un V'Lille.

Donnez les lignes de **code java** permettant de prendre un vélo à `station1` et de le rendre à `station2` s'il reste de la place, sinon de le remettre à `station1`.

L'ensemble des stations du réseau est géré par la classe `lille.StationNetwork`. Cette classe fournit un certain nombre de fonctionnalités pour obtenir des informations sur une station à partir de son nom.
Il est ainsi possible

1. d'ajouter une station au réseau (méthode `addStation`) ;

2. de récupérer une collection des noms de toutes les stations du réseau (méthode `allStationNames`) ;
3. de connaître le nombre maximum de vélos que peut accueillir une station à l'aide de son nom (méthode `getNbMaxBikes`) ;
4. de connaître le nombre de vélos disponibles à une station à l'aide de son nom (méthode `getNbAvailableBikes`) ;
5. de connaître le nombre de vélos disponibles sur l'ensemble des stations du réseau (méthode `getNbTotalAvailableBikes`).

Le résultat des méthodes 3 et 4 est 0 si la chaîne de caractères fournie ne correspond pas au nom de l'une des stations.

Q 5 . (15 mn) Donnez le **code java** d'une telle classe **StationNetwork**.

Exercice 3 : Attrapez les tous (50 mn)

Les Pokémons sont d'étranges animaux. Leur activité principale consiste à se déplacer et à absorber de l'énergie.
Les Pokémons habitent sur des mondes particuliers modélisés par des objets de la classe `PokemonWorld`. Les mondes des Pokemons disposent d'un champ d'énergie qui influence l'activité des Pokemons, par exemple leur vitesse de déplacement. L'intensité de ce champ varie dans le temps mais elle est la même pour tous ces mondes. La classe `PokemonWorld` dispose donc de la méthode :

```
public static double energyLevel();
```

dont le résultat est la valeur de l'intensité de ce champ d'énergie (appelée *intensité d'énergie* dans la suite).

On distingue quatre grandes familles de Pokémons :

- ▷ les **terractifs** : ces Pokémons sont caractérisés par un nom, un poids (en kg), une taille (en m), un nombre de pattes et un coefficient de vitalité (un flottant). Ils se déplacent sur terre à une vitesse qui se calcule grâce à la formule :

  $$\textit{vitesse} = \textit{nombre de pattes} \times ((\textit{poids} + 1)/10) \times 2 \times \textit{intensité d'énergie}$$

  Ces Pokémons absorbent quotidiennement une quantité d'énergie qui se calcule par la formule

  $$\textit{énergie absorbée} = \textit{coefficient de vitalité} \times \textit{taille}^2 \times 100$$

- ▷ les **dormeurs** : ces pokémons sont caractérisés par un nom, un poids (en kg), une taille (en m), un nombre de pattes et un nombre d'heures de repos quotidien. Ils se déplacent sur terre à une vitesse qui se calcule grâce à la formule :

  $$\textit{vitesse} = \textit{nombre de pattes} \times ((\textit{poids} + 1)/10) \times 2 \times \textit{intensité d'énergie}$$

  Ces Pokémons absorbent quotidiennement une quantité d'énergie qui se calcule par la formule

  $$\textit{énergie absorbée} = \textit{taille}^2 - (\textit{heures de repos} \times 2)$$

- ▷ les **aquatiques** : ces pokémons sont caractérisés par un nom, un poids (en kg) et un nombre de nageoires. Ils se déplacent dans l'eau à une vitesse qui se calcule grâce à la formule :

  $$\textit{vitesse} = \textit{nombre de nageoires} \times (\textit{poids}/25) \times \textit{intensité d'énergie}$$

  Ces Pokémons absorbent quotidiennement une quantité d'énergie qui se calcule par la formule

  $$\textit{énergie absorbée} = \textit{poids}/2$$

- ▷ les **aqualents** : ces pokémons sont caractérisés par un nom, un poids (en kg) et un nombre de nageoires. Ils se déplacent dans l'eau deux fois plus lentement qu'un Pokémon aquatique de même poids et ayant autant de nageoires tout en absorbant autant d'énergie.

Pour chacun des Pokémons on doit disposer d'une méthode `toString` dont le résultat reprend les caractéristiques du Pokémon.
Par exemple cette méthode appliquée sur un Pokémon *actif* a pour résultat : `"Je suis Pikachu mon poids est de 18 kg, ma vitesse actuelle est de 11.4 km/h, j'absorbe quotidiennement une énergie de 79, j'ai 2 pattes, ma taille est de 0.85m, ma vitalité est de 1.1."`

et appliquée sur un Pokémon *aqualent* : `"Je suis le Pokémon Aquameche mon poids est de 15 kg, ma vitesse actuelle est de 0.9 km/h, j'absorbe quotidiennement une énergie de 7.5, j'ai 3 nageoires."`

Dans la suite on va s'intéresser à une modélisation objet pour représenter tous ces différents Pokémons.
A terme on doit pouvoir créer une collection dans laquelle on ajoute des Pokémons de toutes les familles et faire des traitements sur cette collection, comme par exemple calculer la vitesse moyenne ou l'énergie absorbée totale des Pokémons contenus dans cette collection.

Il reste peut-être d'autres familles de Pokémons à découvrir, votre proposition de conception devra donc permettre de facilement prendre en compte ces éventuelles nouvelles familles lorsqu'elles auront été découvertes.

Q 1 . (3 mn) Quel **principe de programmmation** orienté objet est exprimé dans la phrase précédente ?

Q 2 . (20 mn) Donnez grâce à un **diagramme UML clair et détaillé** une proposition de modélisation pour représenter les Pokémons.

N'hésitez pas à présenter votre diagramme en « format paysage » sur votre copie.

Dans votre diagramme vous ferez apparaitre :

- les liens d'héritage ou d'implémentation entre types,
- les noms et types de tous les attributs,
- les méthodes et constructeurs avec leurs paramètres et leurs types ainsi que les types des valeurs de retour.

  Les méthodes doivent apparaitre dans chaque classe qui définit un nouveau comportement pour cette méthode.

Q 3 . (15 mn) Donnez le code java de la classe et de toutes les super-classes (sauf `Object` évidemment) permettant de représenter les Pokémons de la famille des `aqualents`.

Q 4 . Des découvertes récentes ont montré que les Pokémons *dormeurs* et *aqualents* sont télépathes. Cela se traduit par l'existence pour ces Pokémons d'une méthode `public void telepathy()` (peu importe ici ce que fait son code).

On souhaite avoir la possibilité de créer des collections de Pokémons télépathes, dans lesquelles on pourrait donc trouver des Pokémons de ces deux familles.

Q 4.1. (7 mn) Sans reprendre l'ensemble du diagramme UML, expliquez clairement et précisément les **modifications** qu'il faudrait modifier par rapport à votre proposition de la question 2 pour prendre en compte cette découverte.

Q 4.2. (5 mn) Sans écrire le code indiquez tout aussi clairement **tous les changements** que ces modifications impliquent pour le code de la classe représentant les Pokémons *aqualents*.