UE Programmation Orientée Objet

Devoir Surveillé

jeudi 24 mai 2018 – 8h–10h

Copie des diapositives de cours annotées

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

Tout autre document interdit

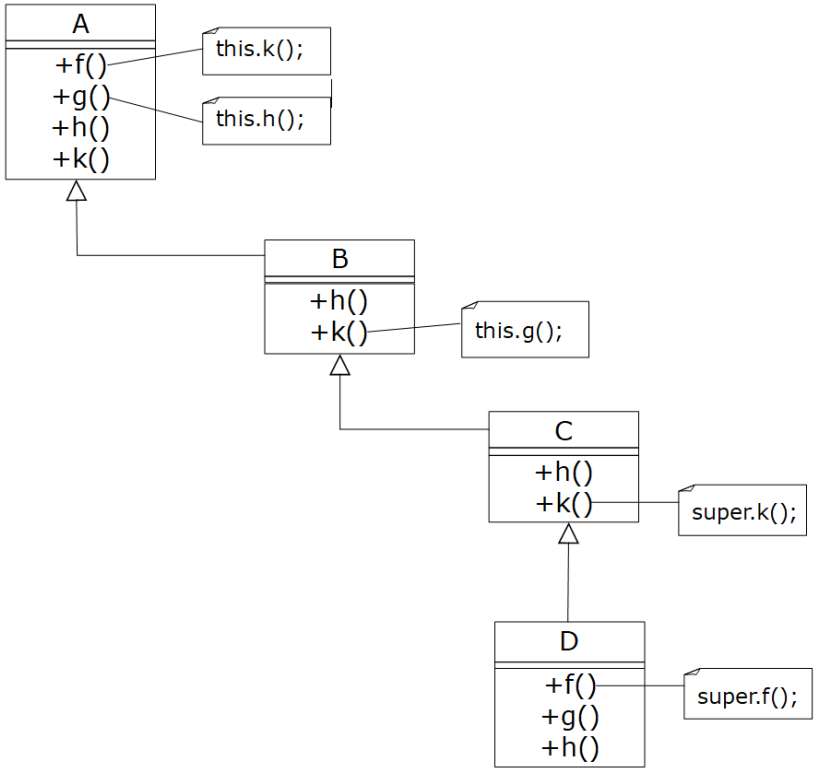
Indiquez votre numéro de groupe sur la copie.

Pas d'anonymat.

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.

Exercice 1 : Lookup

On donne le diagramme de classes suivant :




```
public static void main(String[] args) {
    A a;

    System.out.println("-----");
    a = new B();
    a.f();

    System.out.println("-----");
    a = new D();
    a.f();

    System.out.println("-----");
    ((A) new C()).h();
}
```

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 l'exécution** de la méthode `main` ci-dessus.

Exercice 2 :

On va s'intéresser dans cet exercice à un gestionnaire de connexions (*connection manager*) dans lequel l'authentification des utilisateurs se base classiquement sur un couple *identifiant*, *mot de passe*.
Un gestionnaire de connexion utilise la méthode `connect(String, String)` pour gérer les connexions. Le premier paramètre correspond l'identifiant de connexion et le second au mot de passe proposé pour cet identifiant pour cette connexion. La connexion sera refusée dans trois cas :

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

- si l'identifiant n'est pas connu par le gestionnaire,
- si l'identifiant a été désactivé (voir ci-dessous),
- si l'identifiant est connu mais le mot de passe n'est pas celui associé à cet identifiant.

La première situation déclenche une exception `UnknownLoginException`, les deux suivantes une exception `ConnectionRefusedException`.
Dans les autres cas la connexion est réussie.

Q 1 . Donnez le **code** d'une classe `UnknownLoginException` du paquetage `connection`.

On supposera l'exception `ConnectionRefusedException` définie de manière similaire.

Logins
L'association entre un identifiant et un mot de passe est gérée par une instance de la classe `Login`. Ces deux informations sont fournies sous la forme de chaînes de caractères à la construction d'un objet `Login`. Deux objets `Login` sont considérés égaux dès qu'ils ont le même identifiant.
Il n'y a évidemment pas d'accesseur pour le mot de passe, mais la méthode

```
public boolean acceptPassword(String)
```

vaut `true` si son paramètre correspond au mot de passe du *login*.
Un *login* est désactivé dès que trois (par exemple) tentatives consécutives de connexion avec son identifiant sont refusées à cause d'un mauvais mot de passe. La méthode `addRejection` de la classe `Login` est invoquée pour signaler un tel refus de connexion pour mauvais mot de passe. Cette méthode gère également la désactivation du *login* si nécessaire.

Q 2 . Donnez un **diagramme de classe** détaillé de la classe `Login`

Q 3 . Ecrivez les **tests** qui permettent de vérifier qu'un *login* est désactivé une fois que le nombre autorisé de refus de connexion consécutifs est dépassé, mais pas avant.

Q 4 . Donnez le **code** de la classe `Login` définie dans le paquetage `connection.account`.

Gestionnaires de connexions
Les gestionnaires de connexions sont modélisés dans la classe `ConnectionManager`. La méthode `addLogin(Login)` permet d'ajouter à un gestionnaire un nouveau *login* connu.
En plus de cette méthode et de la méthode `connect` évoquée précédemment, un gestionnaire de connexion dispose des méthodes :

- `connectedIds()` qui renvoie la liste de tous les identifiants actuellement connectés,
- `disconnect(String)` qui prend en paramètre un identifiant et permet sa déconnexion.

  Cela consiste à le retirer des identifiants connectés, sans plus de vérification.
- `getLogin(String)` qui permet d'obtenir l'objet `Login` associé à un identifiant fourni en paramètre à condition que celui-ci soit l'un des logins connectés. Dans le cas contraire, que l'identifiant soit l'un des identifiants connus ou non, une exception `NotConnectedException` est déclenchée.

  Cette exception est supposée définie de la même manière que `ConnectionRefusedException`.
- `disabledLogins()` qui renvoie parmi les *logins* gérés par ce gestionnaire la liste des *logins* de ceux qui sont désactivés.

Q 5 . A l'étude des méthodes demandées, quelle **structure de données** est selon vous la mieux adaptée pour gérer :

Q 5.1. les logins connus par le gestionnaire,
Q 5.2. les identifiants connectés.

Q 6 . Donnez le code des **méthodes de tests** permettant de vérifier que l'implémentation de la `getLogin` est conforme au cahier des charges ci-dessus.

Q 7 . Donnez la **javadoc** de la méthode `getLogin`.

Q 8 . Donnez le **code java** complet de la classe `ConnectionManager` du paquetage `connection`.

Exercice 3 :
On se place dans le contexte d'un jeu de simulation au tour par tour où les joueurs doivent gérer un territoire et en exploiter les ressources à l'aide d'unités de personnages ayant différentes compétences. À chaque tour le joueur récupère des points en fonction de la collecte de ressources réalisée par les unités qu'il contrôle.
Dans cet exercice on ne s'intéressera pas directement à la logique du jeu mais à la gestion d'une partie des unités de personnage (appelées simplement *unités* par la suite).
On suppose que toutes les classes de l'exercice appartiennent à un même paquetage : `simulation`.

Le terrain et les ressources

Le jeu considéré se déroule sur une carte divisée en cases de terrain (*land*). Au début du jeu chaque (case de) terrain possède un certain nombre de ressources. On identifie trois types de ressources : le bois (*wood*), le minerai (*ore*) et des cultures (*crop*). Les joueurs peuvent, petit à petit, collecter ces ressources grâce à leurs unités et obtiennent en retour un certain nombre de points. Une case de terrain est modélisée par le type `Land` suivant :

Land
- wood : int - ore : int - crop : int
+ Land(wood : int, ore : int, crop : int) + collectWood(amount : int) : int + collectOre(amount : int) : int + collectCrop(amount : int) : int + getWood() : int + getOre() : int + getCrop() : int

Les méthodes *`collectXxx`* sont les méthodes qui permettent de récolter *`amount`* ressources du type *`Xxx`* sur cette case. Elles prennent en paramètre le montant de ressource que l'on essaie de prélever (s'il en reste suffisamment sur le terrain). Les ressources disponibles sont diminuées d'autant. Le résultat de ces méthodes est le nombre de points reçus pour la collecte, il vaut 0 si la ressource collectée n'est plus disponible sur ce terrain.

Les unités

Le joueur peut déplacer des unités (*units*) pour les positionner sur les cases de terrain.
Toutes les unités sont caractérisées par une vitesse de déplacement représentée par un entier. De plus chaque unité dispose d'un « coût d'entretien » que doit payer le joueur à chaque tour pour pouvoir conserver cette unité. Cette information est un entier fourni par la méthode `cost()`.
On distingue deux types d'unité : les unités militaires (*military unit*) et les unités ouvrières (*worker unit*).

Les unités militaires Les unités militaires ont pour caractéristique la force (*strength*) et le coût d'une unité militaire est fournie par la formule :

$$\frac{force+1}{5} + 2$$

Les unités militaires ont la possibilité de capturer une autre unité grâce à la méthode `capture()` qui prend en paramètre l'unité capturée.

Certaines unités militaires peuvent être « montées » (« à cheval ») (*mounted*). Le cheval qu'utilise une telle unité est passé en paramètre lors de sa construction. Il s'agit d'une instance de la classe `Horse` qui dispose d'une méthode `getSpeed()` qui fournit la vitesse du cheval[2].
La vitesse des unités militaires montées est celle de leur cheval et leur coût d'entretien est augmenté de 2 par rapport à celui d'une unité militaire « non montée » pour prendre en compte le cheval. La méthode `getHorse()` permet d'accéder au cheval de l'unité.

[2]Pour simplifier, ce sera la seule méthode utile dans ce sujet pour la classe `Horse`.

Les unités ouvrières Une unité ouvrière peut exploiter un terrain pour y collecter des ressources. Ce travail se fait grâce à la méthode `work(Land)` dont le résultat est le nombre de points obtenu suite à la collecte de ressources effectuée par l'unité sur le terrain passé en paramètre.
Toute unité ouvrière possède un niveau d'aptitude (*ability*) et une force de travail (*working power*). Ces données sont représentées par des entiers. La force de travail peut éventuellement être modifiée au cours du jeu et le coût d'entretien d'une unité ouvrière est fournie par la formule :

$$\frac{\textit{force de travail}}{10} + 1$$

Les unités ouvrières se répartissent en trois types qui peuvent chacune collecter un type de ressource spécifique, via leur méthode `work()`.

- les fermiers (*farmer*) qui récoltent les cultures d'un terrain. Ils doivent donc pour cela appeler la méthode `collectCrop` sur l'unité de terrain qu'ils travaillent.
- les mineurs (*miner*) qui récoltent le minerai. Ils doivent donc pour cela appeler la méthode `collectOre` sur l'unité de terrain qu'ils travaillent.
- les bucherons (*lumberjack*) qui récoltent le bois. Ils doivent donc pour cela appeler la méthode `collectWood` sur l'unité de terrain qu'ils travaillent.

Q 1 . Donnez grâce à un **diagramme UML clair et détaillé** une proposition de modélisation pour représenter les unités (ne vous occupez pas des classes `Land` et `Horse` dans votre schéma).

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 2 . Donnez le code d'une méthode `main` qui :

- crée une variable unité de terrain `land` avec 10 ressources de chaque type
- crée une liste d'unités `workers` ouvrières et lui ajoute un fermier, un bûcheron et un mineur, ayant tous 1 pour vitesse, force de travail et niveau d'aptitude
- calcule dans la variable `points` les points reçus en un tour par le travail des unités dans `workers` sur le terrain `land`.

Q 3 . Donnez le **code java** de la classe et de toutes les super-classes (sauf `Object` évidemment) permettant de représenter les `unités militaires montées` (à nouveau vous n'avez pas à vous occuper des classes `Land` et `Horse`).

Vous remplacerez le code de la méthode `capture` par l'affichage dun message `"capture done"`.