Université Lille 1 – Licence 2 Informatique 2015-2016

UE Programmation Orientée Objet

Devoir Surveillé

Mercredi 13 mai 2015 – 10h30-12h30

Copie des diapositives de cours annotées + une feuille A4 recto-verso de notes personnelles autorisées

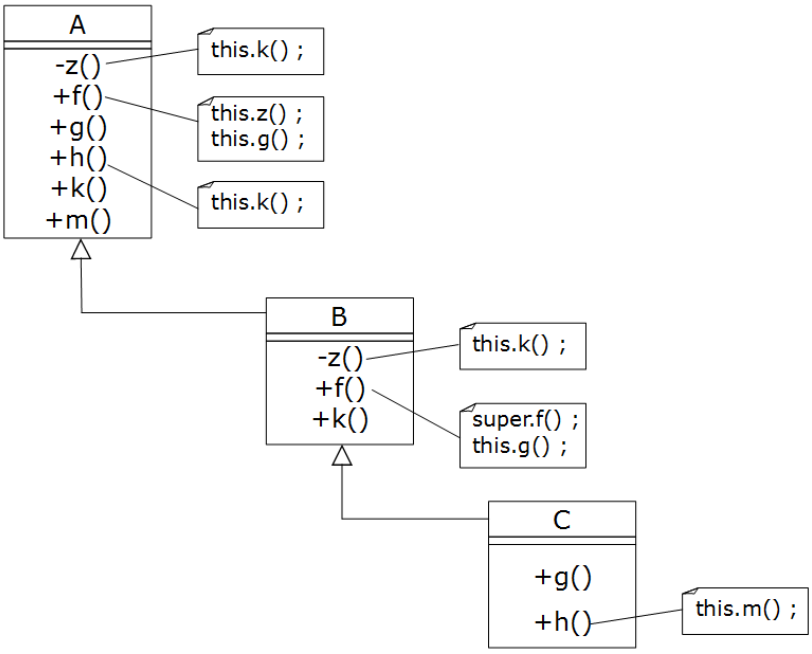
Dictionnaire de langue (papier ou électronique "dédié") autorisé

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 (20mn)

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 A());
     listeA.add(new B());
     listeA.add(new C());

     for(A a : listeA) {
         System.out.println("--- appel de h()---");
         a.h();
         System.out.println("--- appel de f()---");
         a.f();
     }
}
```

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

Exercice 2 : Des colis (50mn)

Tous les types de cet exercice sont à définir dans un paquetage `shipment`.

Le contexte de ce sujet pourrait être celui d'une société de vente en ligne qui propose des articles divers et doit les expédier. On s'intéresse plus particulièrement aux colis nécessaires à leur expédition.

Les articles Les articles (« *good* ») mis en vente et expédiés sont caractérisés par un nom, un prix et un poids (en grammes) précisés à la construction d'un objet article. Des frais d'expédition (« *shipping cost* ») sont également liés à chacun des articles. Leur valeur dépend de la catégorie de l'article. La classe suivante permet de modéliser des articles.

shipment::Good
...
+ Good(name : String, price : float, weight : int) + getName() : String + getWeight() : float + getPrice() : float + shipmentCost() : float

Les colis Un colis regroupe plusieurs articles pour leur expédition. On trouve 3 types de colis : des colis « normaux » (« *parcel* »), des colis « express » (« *express parcel* ») et des colis « avec assurance » (« *insured parcel* ») . On suppose qu'il n'est pas prévu de créer des colis « express avec assurance ».
En plus de la liste des articles qui le compose, un colis est défini par une adresse d'expédition, de type `shipment.Address`, fournie à la construction. Le type `shipment.Address` est supposé défini (si besoin, pour simplifier, on pourra supposer que cette classe dispose d'un constructeur sans paramètre).
Les colis « avec assurance » sont en plus définis par la valeur maximale assurée, un nombre entier d'euros, également fourni à la construction du colis.
Un colis peut être complété jusqu'à son expédition par l'ajout de nouveaux articles grâce à la méthode `addGood`. L'expédition d'un colis est gérée par une méthode `ship` (on ne se préoccupe pas ici de savoir comment se déroule réellement cette expédition).
La tentative d'ajout d'un article (via `addGood`) à un colis déjà expédié provoque la levée d'une exception `ParcelShippedException`

Q 1 . (5mn) Donnez un code java pour la classe `ParcelShippedException`.

Les frais d'expédition d'un colis « normal » sont obtenus en réalisant la somme des frais de chacun des articles expédiés. Pour un colis « avec assurance », les frais d'expédition sont calculés sur la base de ceux d'un colis normal avec un surcoût obtenu ainsi : si le montant assuré du colis est inférieur à 100 euros alors le surcoût est de 5 euros, sinon le surcoût est de 5% du montant assuré. Pour un colis « express », les frais d'expédition sont le double de ceux d'un colis normal.
On veut évidemment disposer d'une méthode `getShippingCost` permettant d'obtenir les frais d'expédition d'un colis. On souhaite également disposer d'une méthode qui fournit la liste des articles composant le colis et une autre fournissant le prix global de l'ensemble des articles, hors frais d'expédition.

Q 2 . (15mn) Donnez grâce à un diagramme UML **clair et détaillé** (liens d'héritage/implémentation, méthodes, types des attributs, paramètres et valeurs de retour, etc.) une proposition de modélisation pour représenter les différents colis.

Les méthodes surchargées s'il y en a doivent apparaître dans le diagramme.

Q 3 . Pour les colis « normaux », donnez la javadoc, le code java et la ou les méthodes de test[2] que vous jugez pertinentes pour les méthodes :

Q 3.1. (8mn) `getShippingCost`

Q 3.2. (12mn) `addGood`

Q 4 . (10mn) Donnez tout le code java de la classe permettant la définition des colis « avec assurance ».

[2]Seul le code des méhodes de test est demandée, pas celui de la classe de test.

Exercice 3 : Météo (50mn)

Tous les types de cet exercice sont à définir dans un paquetage `meteo`.

On va s'intéresser dans cet exercice à des données météorologiques fournies par des stations météorologiques et sur lesquelles on exécutera des traitements simples.

Données météorogologiques. Les données météorologiques mesurées portent sur les températures minimales et maximales et la hauteur des précipitations (la pluviométrie en mm). Le jour où ont été faites ces mesures est également enregistré. Cette information est fournie sous la forme d'un entier représentant le nombre de jours écoulés depuis une date de référence (le 1er janvier 1900 par exemple).
Ces données sont représentées par des instances de la classe `meteo.WeatherData` dont voici le diagramme UML :

meteo::WeatherData
- day : int - tMin : double - tMax : double - precipitation : double
+ WeatherData(day : int, tMin : double, tMax : double, precip : double) + getDay() : int + getTMin() : double + getTMax() : double + getPrecipitation() : double

Stations météorogologiques. Les stations sont représentées par un identifiant unique et un nom (généralement une ville). Elles sont représentées par la classe :

meteo::WeatherStation
- id : String - name : String
+ WeatherStation(id : String, name : String) + getId() : String + getName() : String + toString() : String

Analyse des données L'analyse des données météorologiques est réalisée par des objets de la classe `WeatherAnalysis`. Le squelette, avec la documentation, du code source de cette classe est fournie à la figure 1.

Q 1 . (10mn) A la lecture de ce code source, quels éléments manquent dans les classes `WeatherData` et/ou `WeatherStation` présentées précédemment ?

Donnez le code java correspondant aux éléments à ajouter à ces classes (uniquement ce code là).

Q 2 . Donnez le code java :

Q 2.1. (2mn) du constructeur de `WeatherAnalysis` ;

Q 2.2. (5mn) de la méthode `addDailyDataForStation` ;

Q 2.3. (5mn) de la méthode `receiveDailyData`.

Les opérations permettant de manipuler les données météorologiques sont définies par l'interface :

≪ interface ≫ *Operation*
+ process(dataList : List<WeatherData>) : double + getDescription() : String

La méthode `process` réalise un traitement sur une liste d'objets `WeatherData` pour produire un résultat. La méthode `getDescription` fournit simplement une chaîne de caractères décrivant l'opération réalisée.

Q 3 . (7mn) Définissez un type `MinTMinOperation` pour définir des opérations dont le traitement consiste à fournir la température minimale présente dans une liste de `WeatherData`.

Q 4 . (7mn) Définissez `AccumulatedPrecipitation` pour définir des opérations, paramétrées à leur construction par la donnée de deux jours, `start` et `end`, représentés par des entiers comme décrit ci-dessus, et dont le traitement consiste à fournir le cumul des précipitations pour les `WeatherData` de la liste qui ont été mesurées entre les jours `start` et `end`.

Q 5 . (7mn) Donnez la signature puis le code de la méthode `processData` de `WeatherAnalysis` qui doit permettre à un objet `WeatherAnalysis` de réaliser un traitement, comme ceux décrits ci-dessus, sur les données connues pour une station donnée.

D'autres types de traitement doivent pouvoir être ajoutés sans que cela ne pose de problème.

Q 6 . (7mn) Indiquez les lignes de code qui complètent la méthode `main` ci-dessous de manière à calculer la température minimale connue pour `station1`. Le comportement que l'on veut obtenir est le suivant :

1. si aucune donnée n'est connue pour `station1`, un message indiquant cette situation est affichée ;
2. dans le cas contraire, un message annonçant la description de l'opération exécutée ainsi que le résultat obtenu est affiché.

```
public static void main(String[] args) {
  WeatherAnalysis analysis = new WeatherAnalysis();
  // ... dailyData created, initialized and filled with values...
  analysis.receiveDailyData(dailyData);

  WeatherStation station1 = new WeatherStation("1234","SomeWhereCity");

  // COMPLETER ICI
}
```

```java
package meteo;
import java.util.*;
public class WeatherAnalysis {

  /** the data managed by this object :
   * associates a station with the list of weather data known for this station */
  private Map<WeatherStation, List<WeatherData>> allData;

  /** Builds a WeatherAnalysis object with an initial empty allData map */
  public WeatherAnalysis() {
    ...
  }

  /** Adds a list of new weather data to the given station.
   *  If station is not yet in allData, it is added, else data in the list are added
   *    to already existing data.
   * @param station the station for which data are given
   * @param dataList the list of weather data added for given station
   */
  protected void addDailyDataForStation(WeatherStation station, List<WeatherData> dataList)
    ...
  }

  /** Adds the received weather data into allData
   * @param dailyData the received data : a map that associated station to the list
   *                  of data that must be added to each station in allData
   * @see WeatherAnalysis#addDailyDataForStation(WeatherStation, List)
   */
  public void receiveDailyData(Map<WeatherStation,List<WeatherData>> dailyData) {
    ...
  }

  /** Performs some operation on the list of data known for given station.
   * The result of the operation is returned.
   * @param station the station
   * @param operation the operation to be executed on the data list
   * @return the result of operation processing on the list of data for the given station
   * @throws WeatherDataEmptyListException if there exists no data for the given station
   */
  public double processData(...) ... {
    ...
  }
}
```

Figure 1: Squelette du code source de la classe `WeatherAnalysis`