Initiation à la programmation Java
IP2 - Séance No 12

Yan Jurski

Université de Paris

Révisions

Plan

- Examen Session 1 2014-2015
- Examen Session 1 2016-2017
- A noter : AUCUN DOCUMENT AUTORISES

Université de Paris

Révisions - Session 1 2014-2015

Un service de nettoyage de trains modélise le travail de ses employés.
On représente un train par une liste chaînée, chaque wagon étant une cellule.
On utilisera les classes :

- Train pour la tête de liste
- Wagon pour chaque cellule.

Université de Paris

Révisions - Session 1 2014-2015

Un service de nettoyage de trains modélise le travail de ses employés.
On représente un train par une liste chaînée, chaque wagon étant une cellule.
On utilisera les classes :

- Train pour la tête de liste
- Wagon pour chaque cellule.

Chaque train posséde un identifiant entier individuel unique et chaque wagon un booléen permettant de savoir s'il est propre ou non.
Tous les attributs des classes Train et Wagon doivent être privés.

Université de Paris

Révisions - Session 1 2014-2015

Un service de nettoyage de trains modélise le travail de ses employés.
Il représente un train par une liste chaînée, chaque wagon étant une cellule.
On utilisera les classes :

- Train pour la tête de liste
- Wagon pour chaque cellule.

Chaque train posséde un identifiant entier individuel unique et chaque wagon un booléen permettant de savoir s'il est propre ou non.
Tous les attributs des classes Train et Wagon doivent être privés.
L'interface de la classe Train devra contenir (au minimum) :

```
int id(); // renvoie l'identifiant du train
int longueur(); // renvoie le nombre de wagons du train
void estToutSale(); // passe tous les wagons a "sale"
int wagonsANettoyer();  // renvoie le nombre de wagons sales
```

Exercice 1 (2,5 points) Donner les attributs et les constructeurs des classes Train et Wagon , ainsi que la méthode id.
(Les constructeurs doivent permettre d'obtenir un train, vide ou non)

Université de Paris

Fichier Train.java

```
public class Train {
 private Wagon tete;
 private final int id;

 private static int nbId=0;

 public Train(){
  this.id=nbId++;
  this.tete=null;
 }

 public Train(Wagon x){
  this();
  tete=x;
 }

 public int id(){
  return id;
 }
}
```

Fichier Train.java

```
public class Train {
 private Wagon tete;
 private final int id;

 private static int nbId=0;

 public Train(){
  this.id=nbId++;
  this.tete=null;
 }

 public Train(Wagon x){
  this();
  tete=x;
 }

 public int id(){
  return id;
 }
}
```

Fichier Wagon.java

```
public class Wagon {
 private boolean estPropre;
 private Wagon next;

 public Wagon(){
  this.estPropre=true;
  next=null;
 }
}
```

Université de Paris

Exercice 2. (3,5 points)

Écrire les méthodes

```
int longueur(); // renvoie le nombre de wagons du train
void estToutSale(); // passe tous les wagons a "sale"
int wagonsANettoyer();  // renvoie le nombre de wagons sales
```

Université de Paris

Exercice 2. (3,5 points)

Écrire les méthodes

```
int longueur(); // renvoie le nombre de wagons du train
void estToutSale(); // passe tous les wagons a "sale"
int wagonsANettoyer();  // renvoie le nombre de wagons sales
```

Fichier Train.java

```
public class Train {
 private Wagon tete;
 private final int id;
 private static int nbId=0;
 ...
 public int longueur(){
  if (tete==null) return 0;
  return tete.longueur();
 }
}
```

Université de Paris

Exercice 2. (3,5 points)

Écrire les méthodes

```
int longueur(); // renvoie le nombre de wagons du train
void estToutSale(); // passe tous les wagons a "sale"
int wagonsANettoyer();  // renvoie le nombre de wagons sales
```

Fichier Train.java

```
public class Train {
 private Wagon tete;
 private final int id;
 private static int nbId=0;
 ...
 public int longueur(){
  if (tete==null) return 0;
  return tete.longueur();
 }
}
```

Fichier Wagon.java

```
public class Wagon {
 private boolean estPropre;
 private Wagon next;

 // version récursive
 public int longueur(){
  if (next==null) return 1;
  return 1+next.longueur();
 }
}
```

Université de Paris

Exercice 2. (3,5 points)

Écrire les méthodes

```
int longueur(); // renvoie le nombre de wagons du train
void estToutSale(); // passe tous les wagons a "sale"
int wagonsANettoyer();  // renvoie le nombre de wagons sales
```

Fichier Train.java

```
public class Train {
 private Wagon tete;
 private final int id;
 private static int nbId=0;
 ...
 public int longueur(){
  if (tete==null) return 0;
  return tete.longueur();
 }
}
```

Université de Paris

Exercice 2. (3,5 points)

Écrire les méthodes

```
int longueur(); // renvoie le nombre de wagons du train
void estToutSale(); // passe tous les wagons a "sale"
int wagonsANettoyer();  // renvoie le nombre de wagons sales
```

Fichier Train.java

```
public class Train {
 private Wagon tete;
 private final int id;
 private static int nbId=0;
 ...
 public int longueur(){
  if (tete==null) return 0;
  return tete.longueur();
 }
}
```

Fichier Wagon.java

```
public class Wagon {
 private boolean estPropre;
 private Wagon next;

 // version itérative
 public int longueur(){
  int rep=0;
  Wagon tmp=this;
  while(tmp!=null) {
   rep++;
   tmp=tmp.next;
  }
  return rep;
 }
}
```

Exercice 2. (3,5 points)

Écrire les méthodes

```
int longueur(); // renvoie le nombre de wagons du train
void estToutSale(); // passe tous les wagons a "sale"
int wagonsANettoyer();  // renvoie le nombre de wagons sales
```

Fichier Train.java

```
public class Train {
 private Wagon tete;
 private final int id;
 private static int nbId=0;
 ...
 public void estToutSale(){
  if (tete !=null)
      tete.estToutSale();
 }
}
```

Université de Paris

Exercice 2. (3,5 points)

Écrire les méthodes

```
int longueur(); // renvoie le nombre de wagons du train
void estToutSale(); // passe tous les wagons a "sale"
int wagonsANettoyer();  // renvoie le nombre de wagons sales
```

Fichier Train.java

```
public class Train {
 private Wagon tete;
 private final int id;
 private static int nbId=0;
 ...
 public void estToutSale(){
  if (tete !=null)
      tete.estToutSale();
 }
}
```

Fichier Wagon.java

```
public class Wagon {
 private boolean estPropre;
 private Wagon next;

 // version itérative
 public void estToutSale(){
  Wagon tmp=this;
  while(tmp!=null) {
   tmp.estPropre=false;
   tmp=tmp.next;
  }
 }
}
```

Université de Paris

Exercice 2. (3,5 points)

Écrire les méthodes

```
int longueur(); // renvoie le nombre de wagons du train
void estToutSale(); // passe tous les wagons a "sale"
int wagonsANettoyer();  // renvoie le nombre de wagons sales
```

Fichier Train.java

```
public class Train {
 private Wagon tete;
 private final int id;
 private static int nbId=0;
 ...
 public void estToutSale(){
  if (tete !=null)
      tete.estToutSale();
 }
}
```

Université de Paris

Exercice 2. (3,5 points)

Écrire les méthodes

```
int longueur(); // renvoie le nombre de wagons du train
void estToutSale(); // passe tous les wagons a "sale"
int wagonsANettoyer();  // renvoie le nombre de wagons sales
```

Fichier Train.java

```
public class Train {
 private Wagon tete;
 private final int id;
 private static int nbId=0;
 ...
 public void estToutSale(){
  if (tete !=null)
      tete.estToutSale();
 }
}
```

Fichier Wagon.java

```
public class Wagon {
 private boolean estPropre;
 private Wagon next;

 // version récursive
 public void estToutSale(){
  this.estPropre=false;
  if (next!=null)
      next.estToutSale();
 }
}
```

Université de Paris

Exercice 2. (3,5 points)

Écrire les méthodes

```
int longueur(); // renvoie le nombre de wagons du train
void estToutSale(); // passe tous les wagons a "sale"
int wagonsANettoyer();  // renvoie le nombre de wagons sales
```

Fichier Train.java

```
public class Train {
 private Wagon tete;
 private final int id;
 private static int nbId=0;
 ...
 public int wagonsANettoyer(){
  if (tete !=null) return
       tete.wagonsANettoyer();
  return 0;
 }
}
```

Université de Paris

Exercice 2. (3,5 points)

Écrire les méthodes

```
int longueur(); // renvoie le nombre de wagons du train
void estToutSale(); // passe tous les wagons a "sale"
int wagonsANettoyer();  // renvoie le nombre de wagons sales
```

Fichier Train.java

```
public class Train {
 private Wagon tete;
 private final int id;
 private static int nbId=0;
 ...
 public int wagonsANettoyer(){
  if (tete !=null) return
       tete.wagonsANettoyer();
  return 0;
 }
}
```

Fichier Wagon.java

```
public class Wagon {
 private boolean estPropre;
 private Wagon next;

 // version récursive
 public int wagonsANettoyer(){
  int x = (estPropre)?0:1;
  if (next==null) return x;
  return
       x+next.wagonsANettoyer();
 }
}
```

Université de Paris

Exercice 2. (3,5 points)

Écrire les méthodes

```
int longueur(); // renvoie le nombre de wagons du train
void estToutSale(); // passe tous les wagons a "sale"
int wagonsANettoyer();  // renvoie le nombre de wagons sales
```

Fichier Train.java

```
public class Train {
 private Wagon tete;
 private final int id;
 private static int nbId=0;
 ...
 public int wagonsANettoyer(){
  if (tete !=null) return
       tete.wagonsANettoyer();
  return 0;
 }
}
```

Université de Paris

Exercice 2. (3,5 points)

Écrire les méthodes

```
int longueur(); // renvoie le nombre de wagons du train
void estToutSale(); // passe tous les wagons a "sale"
int wagonsANettoyer();  // renvoie le nombre de wagons sales
```

Fichier Train.java

```
public class Train {
 private Wagon tete;
 private final int id;
 private static int nbId=0;
 ...
 public int wagonsANettoyer(){
  if (tete !=null) return
       tete.wagonsANettoyer();
  return 0;
 }
}
```

Fichier Wagon.java

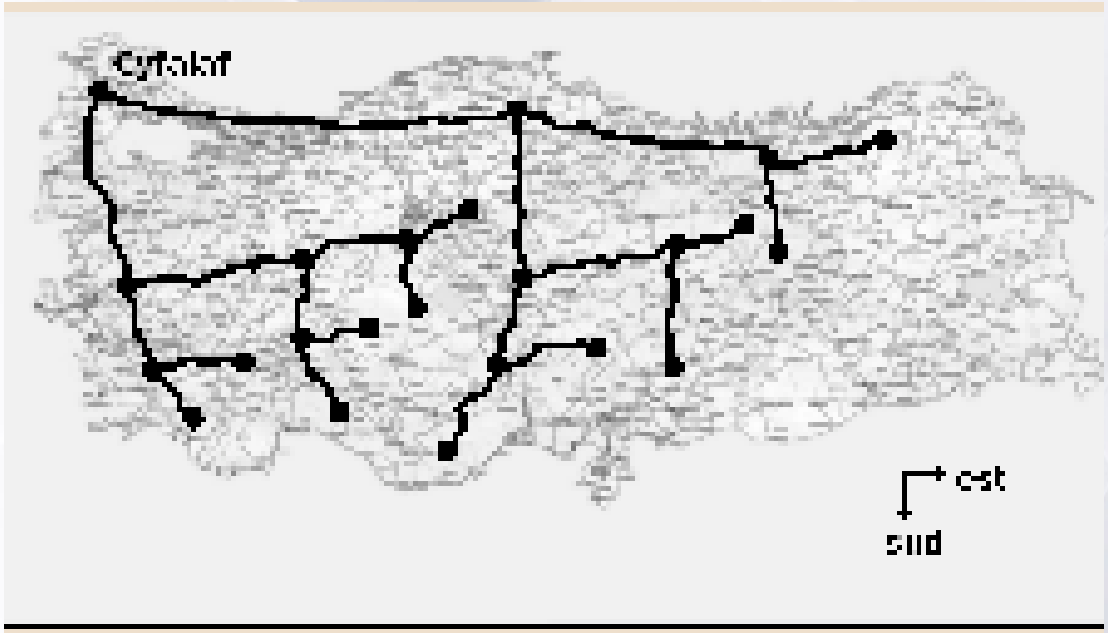
```
public class Wagon {
 private boolean estPropre;
 private Wagon next;

 // version itérative
 public int wagonsANettoyer(){
  int x = 0;
  Wagon tmp=this;
  while(tmp!=null){
   x += (tmp.estPropre)?0:1;
   tmp=tmp.next;
 }
}
```

Université de Paris

Session 1 2014-2015 - Seconde partie

- Dans un pays rectangulaire, la capitale se situe au coin nord-ouest.
- Un réseau ferré où toutes les voies partent ou finissent sur la capitale :



- Le trajet d'un train se fait entre la capitale et une gare terminale (dans un sens ou dans l'autre, avec arrêt à chaque gare intermédiaire)
- On veut modéliser ce réseau par un arbre binaire.

Session 1 2014-2015 - Seconde partie

- Quand on étend le réseau, on construit toujours deux tronçons :
  - un à l'est
  - un autre au sud

  en évitant tout quadrillage :
  - excepté la capitale, chaque gare est sur le trajet d'une unique ligne
- La classe Gare représente les nœuds de l'arbre et la classe ReseauFerre stockera une référence vers la racine de cet arbre.
- Un nœud ne contient pas de référence vers son père.
- Les gares terminales sont celles qui n'ont pas de fils
- Les autres gares ont toutes deux fils.

Université de Paris

Session 1 2014-2015 - Seconde partie

Fichier ReseauFerre.java

```
Gare aLEstToute(); // donne la gare obtenue en allant toujours à l'est
void lsGares(); // affiche les noms de toutes les gares
ArrayList<Gare> trajetVers(Gare gare);
// donne la liste des gares situées entre la capitale et la gare
// ici vous devrez utiliser la librairie Java indiquée
ArrayList<Gare> trajetEntre(Gare depart, Gare arrivee);
// donne les étapes du trajet le plus court entre départ et arrivée
int tempsTrajet(Gare depart, Gare arrivee);
// renvoie le temps de trajet entre ces deux gares
```

- Chaque instance de la classe Gare contient :
  - une chaîne de caractères qui est le nom de la gare
  - un entier correspondant au temps (en minutes) mis par un train pour arriver à cette instance à partir de son père.

Question (2 points)

Donner la liste des attributs des deux classes.

Fichier RéseauFerré.java

```
public class RéseauFerré {
 private final Gare capitale;
 public RéseauFerré(Gare c){
  capitale=c;
 }
 // capitale toujours non null
 // vu le seul constructeur,
 // et final
}
```

Fichier Gare.java

```
public class Gare {
 private final String nom;
 private final int tempsAccès;
 private Gare filsSud;
 private Gare filsEst;
}
```

Université de Paris

Fichier RéseauFerré.java

```
public class RéseauFerré {
 private final Gare capitale;
 public RéseauFerré(Gare c){
  capitale=c;
 }
 // capitale toujours non null
 // vu le seul constructeur,
 // et final
}
```

Fichier Gare.java

```
public class Gare {
 private final String nom;
 private final int tempsAccès;
 private Gare filsSud;
 private Gare filsEst;
}
```

Question (1 point)

Ecrivez **Gare aLEstToute()** au sens où en partant de la racine on va systématiquement à l'Est, le plus loin possible.

Université de Paris

Question (1 point)

Ecrivez **Gare aLEstToute()** au sens où en partant de la racine on va systématiquement à l'Est, le plus loin possible.

Fichier RéseauFerré.java

```
public class RéseauFerré {
 private final Gare capitale;
 public Gare aLestToute(){
  // capitale toujours non null
  return capitale.aLEstToute();
 }
}
```

Université de Paris

Question (1 point)

Ecrivez **Gare aLEstToute()** au sens où en partant de la racine on va systématiquement à l'Est, le plus loin possible.

Fichier RéseauFerré.java

```
public class RéseauFerré {
 private final Gare capitale;
 public Gare aLestToute(){
  // capitale toujours non null
  return capitale.aLEstToute();
 }
}
```

Fichier Gare.java

```
public class Gare {
 private String nom;
 private int tempsAccès;
 private Gare filsSud;
 private Gare filsEst;

 // version récursive
 public Gare aLEstToute(){
  if (filsEst==null)
      return this;
  return filsEst.aLEstToute();
 }
}
```

Université de Paris

Question (1 point)

Ecrivez **Gare aLEstToute()** au sens où en partant de la racine on va systématiquement à l'Est, le plus loin possible.

Fichier RéseauFerré.java

```
public class RéseauFerré {
 private final Gare capitale;
 public Gare aLestToute(){
  // capitale toujours non null
  return capitale.aLEstToute();
 }
}
```

Université de Paris

Question (1 point)

Ecrivez **Gare aLEstToute()** au sens où en partant de la racine on va systématiquement à l'Est, le plus loin possible.

Fichier RéseauFerré.java

```
public class RéseauFerré {
 private final Gare capitale;
 public Gare aLestToute(){
  // capitale toujours non null
  return capitale.aLEstToute();
 }
}
```

Fichier Gare.java

```
public class Gare {
 private String nom;
 private int tempsAccès;
 private Gare filsSud;
 private Gare filsEst;

 // version itérative
 public Gare aLEstToute(){
  Gare tmp=this;
  while (tmp.filsEst != null)
      tmp=tmp.filsEst;
  return tmp;
 }
}
```

Université de Paris

Question (2 points)

Ecrivez **void lsGares()** qui affiche les noms de toutes les gares

- Il s'agit de faire le parcours de votre choix

Université de Paris

Question (2 points)

Ecrivez **void lsGares()** qui affiche les noms de toutes les gares

- Il s'agit de faire le parcours de votre choix

Fichier RéseauFerré.java

```
public class RéseauFerré {
 private final Gare capitale;
 public void lsGares(){
  // capitale toujours non null
  return capitale.lsGare();
 }
}
```

Université de Paris

Question (2 points)

Ecrivez **void lsGares()** qui affiche les noms de toutes les gares

- Il s'agit de faire le parcours de votre choix

Fichier RéseauFerré.java

```
public class RéseauFerré {
 private final Gare capitale;
 public void lsGares(){
  // capitale toujours non null
  return capitale.lsGare();
 }
}
```

Fichier Gare.java

```
public class Gare {
 private String nom;
 private int tempsAccès;
 private Gare filsSud;
 private Gare filsEst;

 // version préfixe
 public void lsGare(){
  System.out.println(nom);
  if (filsSud!=null)
   filsSud.lsGare();
  if (filsEst!=null)
   filsEst.lsGare();
  }
}
```

Question (3 points)

Ecrivez **ArrayList<Gare> trajetVers(Gare x)**...

- Il faut réfléchir un peu :

Question (3 points)

Ecrivez **ArrayList<Gare> trajetVers(Gare x)**...

- Il faut réfléchir un peu :
  - lorsqu'on trouve la gare, le chemin est implicite : la branche du parcours

Université de Paris

Question (3 points)

Ecrivez **ArrayList<Gare> trajetVers(Gare x)**...

- Il faut réfléchir un peu :
  - lorsqu'on trouve la gare, le chemin est implicite : la branche du parcours
- $\Rightarrow$ c'est un cas de transmission de paramètre
  - fonction auxilliaire :
    ... trajetVers(Gare x,ArrayList<Gare> chemin)

Université de Paris

Question (3 points)

Ecrivez **ArrayList<Gare> trajetVers(Gare x)**...

- Il faut réfléchir un peu :
  - lorsqu'on trouve la gare, le chemin est implicite : la branche du parcours
- $\Rightarrow$ c'est un cas de transmission de paramètre
  - fonction auxilliaire :
    boolean trajetVers(Gare x,ArrayList<Gare> chemin)
- et de remontée conditionnelle d'information $\rightarrow$ boolean

Université de Paris

Question (3 points)

Ecrivez **ArrayList<Gare> trajetVers(Gare x)**...

- Il faut réfléchir un peu :
  - lorsqu'on trouve la gare, le chemin est implicite : la branche du parcours
- $\Rightarrow$ c'est un cas de transmission de paramètre
  - fonction auxilliaire :
    boolean trajetVers(Gare x,ArrayList<Gare> chemin)
- et de remontée conditionnelle d'information $\rightarrow$ boolean

Rappels ArrayList de E (et de l'interface List en général)

```
ArrayList(); // Constructs an empty list
boolean add(E e); // Appends the specified element
void add(int index, E element); // Inserts the element at the position
E get(int index); // Returns the element at the specified position
boolean isEmpty();
E remove(int index); // Removes the element at the specified position
boolean remove(E e); // Removes the first occurrence if present
int size(); // Returns the number of elements
```

Question (3 points)

Ecrivez **ArrayList<Gare> trajetVers(Gare x)**...

Fichier RéseauFerré.java

```
public ArrayList<Gare> trajetVers(Gare x){
 ArrayList<Gare> chemin;
 chemin=new ArrayList<Gare>(); // rq qu'on ne met pas la capitale, on
     fait donc un choix d'interprétation du chemin vide
 // capitale toujours non null
 boolean found;
 found=capitale.trajetVers(x,chemin); // on fait le choix de travailler
     toujours sur la même référence de chemin
 if (found) chemin.add(0,capitale);
 return chemin;
}
```

Université de Paris

Fichier Gare.java

```
public boolean trajetVers(Gare x, ArrayList<Gare> chemin){
 if (x.nom.equals(nom)){
  chemin.add(this);
  return true;
 }
 // les deux fils existent ensemble
 if (filsEst==null) return false;
 boolean found;
 found=filsEst.trajetVers(x,chemin);
 if (found) {
  chemin.add(0,this); // rq : on choisit de construire à la remontée
  return true;
 }
 found=filsSud.trajetVers(x,chemin);
 if (found) {
  chemin.add(0,this);
  return true;
 }
 return false;
}
```

Session 1 2014-2016 - Seconde partie

Question (3 points)

- Écrire une fonction qui prend deux ArrayList<Gare> et renvoie le plus grand entier n tel que les n premières gares coïncident
- En déduire la méthode trajetEntre

Fichier RéseauFerré.java

```
private static int coincident(ArrayList<Gare> l1,ArrayList<Gare>l2){
 int n=0;
 while( n < l1.size()
     && n < l2.size()
     && l1.get(n).getNom().equals(l2.get(n).getNom())
     ) n++;
 return n;
}
```

Université de Paris

Fichier RéseauFerré.java

```
public ArrayList<Gare> trajetEntre(Gare x,Gare y){
 ArrayList<Gare> t1=trajetVers(x);
 ArrayList<Gare> t2=trajetVers(y);
 ArrayList<Gare> rep=new ArrayList<Gare>();
 int n=coincide(t1,t2);
 for (int i=t1.size()-1;i>=n;i--){
  Gare g;
  g=t1.get(i);
  rep.add(g);
 }
 for (int i=n;i<t2.size();i++){
  rep.add(t2.get(i));
 }
 return rep;
}
```

Question (3 points)

- Les trains s'arrêtent sur chaque gare du parcours pendant 2 minutes
- Ils circulent toujours selon le temps prévu
- En cas de changement de train :
  - à la gare de la capitale, il faut compter 15 minutes
  - dans une autre gare 10 minutes.
- Ecrivez **int tempsTrajet(Gare x, Gare y)**

Université de Paris

Question (3 points)

- Les trains s'arrêtent sur chaque gare du parcours pendant 2 minutes
- Ils circulent toujours selon le temps prévu
- En cas de changement de train :
  - à la gare de la capitale, il faut compter 15 minutes
  - dans une autre gare 10 minutes.
- Ecrivez **int tempsTrajet(Gare x, Gare y)**

- A faire vous même ... c'est plus simple qu'il n'y paraît (parcours de liste)

Université de Paris

Seul document autorisé : une feuille A4 recto-verso manuscrite (non photocopiée). Aucune machine.
Le barème est indicatif. Une question peut toujours être traitée en utilisant les précédentes (traitées ou non).
Les morceaux de code Java devront être clairement présentés, indentés et commentés.
L'utilisation de collections de l'API Java (en particulier `LinkedList` et `ArrayList`) est interdite, sauf mention explicite contraire.

- Tous les attributs d'instance de toutes les classes introduites doivent être privés.
- Il est possible d'écrire des méthodes non spécifiquement demandées si c'est plus pratique, en spécifiant pour chaque méthode dans quelle classe elle se trouve.

Le gérant du "gland de chêne" a trouvé très efficace de passer à une gestion informatisée des commandes. Il voudrait améliorer maintenant la productivité en cuisine et éviter les erreurs de composition de sandwichs. Le but de ce sujet est de lui proposer une implémentation de "recettes" de sandwichs.

Ingrédient

Un ingrédient est représenté par son nom, son stock courant et son stock d'alerte (ces deux derniers étant représentés par des entiers, sans se soucier des unités). La classe **`Ingredient`** intègre par ailleurs un mécanisme qui permet de connaître à chaque instant la liste des ingrédients dont le stock est passé en dessous du seuil d'alerte. Cette liste peut être modélisée en utilisant la classe **`java.util.ArrayList`**.

La classe **`Ingredient`** contient donc au minimum un attribut de type chaîne de caractères et deux attributs entiers représentant le stock courant et le stock d'alerte. Son interface est la suivante (le caractère statique ou non des méthodes n'est pas spécifié, ce sera à vous de le donner, en justifiant votre choix) :

```
/** affiche la liste des ingrédients à acheter (ceux dont le stock courant est strictement
inférieur au stock d'alerte) avec la quantité à acheter: pour chaque ingrédient, la quantité
à acheter doit permettre de mettre le stock courant au stock d'alerte plus la quantité
passée en paramètre */
void listeDeCourse(int surplus);
```

Le gérant du “gland de chêne” a trouvé très efficace de passer à une gestion informatisée des commandes. Il voudrait améliorer maintenant la productivité en cuisine et éviter les erreurs de composition de sandwichs. Le but de ce sujet est de lui proposer une implémentation de “recettes” de sandwichs.

Ingrédient

Un ingrédient est représenté par son nom, son stock courant et son stock d'alerte (ces deux derniers étant représentés par des entiers, sans se soucier des unités). La classe **`Ingredient`** intègre par ailleurs un mécanisme qui permet de connaître à chaque instant la liste des ingrédients dont le stock est passé en dessous du seuil d'alerte. Cette liste peut être modélisée en utilisant la classe **`java.util.ArrayList`**.

La classe **`Ingredient`** contient donc au minimum un attribut de type chaîne de caractères et deux attributs entiers représentant le stock courant et le stock d'alerte. Son interface est la suivante (le caractère statique ou non des méthodes n'est pas spécifié, ce sera à vous de le donner, en justifiant votre choix) :

```
/** affiche la liste des ingrédients à acheter (ceux dont le stock courant est strictement
inférieur au stock d'alerte) avec la quantité à acheter: pour chaque ingrédient, la quantité
à acheter doit permettre de mettre le stock courant au stock d'alerte plus la quantité
passée en paramètre */
void listeDeCourse(int surplus);

/** simule l'achat d'un ingrédient en ajoutant la valeur du paramètre au stock courant */
void achat(int quantite);

/** simule la consommation d'une unité de l'élément, quand c'est possible */
void consomme();
```

Cette classe possède un constructeur qui prend en argument le nom d'un ingrédient et son stock d'alerte.

Exercice 1. (3 points) Écrire la classe **`Ingredient`** (attributs, constructeurs et méthodes). N'oubliez pas d'expliquer ce qui doit être statique. Notez bien que le surplus en argument de **`listeDeCourse`** est commun à tous les ingrédients : ce n'est pas réaliste, mais c'est pour simplifier l'implémentation.

Recettes

Une recette est une suite ordonnée d'ingrédients. On peut voir apparaître plusieurs fois le même ingrédient dans une recette (ex : dans un double cheeseburger, il y a deux steaks hachés séparés par une tranche de fromage). Pour simplifier, on supposera qu'une apparition d'un ingrédient dans une recette correspond à une unité de cet ingrédient : quand on réalise la recette, chaque apparition décrémente donc le stock de cet ingrédient d'une unité.

L'ensemble des recettes est représenté dans un arbre binaire. L'idée étant qu'en suivant un chemin depuis la racine dans l'arbre jusqu'à un nœud reconnu comme étant un sandwich (mais qui n'est pas nécessairement une feuille), on a la liste des ingrédients *dans l'ordre* pour ce sandwich. Chaque nœud représente une option sur un ingrédient :

```
à acheter doit permettre de mettre le stock courant au stock d'alerte plus la quantité
passée en paramètre */
void listeDeCourse(int surplus);

/** simule l'achat d'un ingrédient en ajoutant la valeur du paramètre au stock courant */
void achat(int quantite);

/** simule la consommation d'une unité de l'élément, quand c'est possible */
void consomme();
```

Cette classe possède un constructeur qui prend en argument le nom d'un ingrédient et son stock d'alerte.

Exercice 1. (3 points) Écrire la classe `Ingredient` (attributs, constructeurs et méthodes). N'oubliez pas d'expliquer ce qui doit être statique. Notez bien que le surplus en argument de `listeDeCourse` est commun à tous les ingrédients : ce n'est pas réaliste, mais c'est pour simplifier l'implémentation.

Recettes

Une recette est une suite ordonnée d'ingrédients. On peut voir apparaître plusieurs fois le même ingrédient dans une recette (ex : dans un double cheeseburger, il y a deux steaks hachés séparés par une tranche de fromage). Pour simplifier, on supposera qu'une apparition d'un ingrédient dans une recette correspond à une unité de cet ingrédient : quand on réalise la recette, chaque apparition décrémente donc le stock de cet ingrédient d'une unité.

L'ensemble des recettes est représenté dans un arbre binaire. L'idée étant qu'en suivant un chemin depuis la racine dans l'arbre jusqu'à un nœud reconnu comme étant un sandwich (mais qui n'est pas nécessairement une feuille), on a la liste des ingrédients *dans l'ordre* pour ce sandwich. Chaque nœud représente une option sur un ingrédient : s'il est intégré au sandwich la recette se poursuit en explorant le fils gauche ; et s'il n'est pas intégré, une autre option d'ingrédient est envisagée : celle du fils droit.

Pour celà, on crée les classes suivantes :

- `Recette` : référence la racine de l'arbre ;
- `Creation` : référence un nœud de l'arbre et contient :

- une référence vers un ingrédient
- deux références vers des nœuds,
- une référence vers le nœud père dans l'arbre (nulle dans le cas de la racine de l'arbre),
- une référence vers un sandwich, non nulle si et seulement si le nœud correspond à un sandwich ;

— **`Sandwich`** : référence un sandwich et contient :

- un attribut de type chaîne de caractères contenant le nom du sandwich,
- une référence vers le nœud de l'arbre qui correspond à ce sandwich (c'est-à-dire celui qui contient l'instance courante de **`Sandwich`** comme attribut).

Exercice 2. (1 point) Écrire les déclarations des attributs des classes **`Recette`**, **`Creation`** et **`Sandwich`**.

Exercice 3. (1 point) Dessiner une instance de **`Recette`** qui contient les recettes des trois sandwichs suivants (les ingrédients sont notés dans l'ordre) :

- le *matelot* : beurre, saumon, aneth ;
- le *parigot* : beurre, jambon ;
- le *grassouillet* : saumon, mayonnaise.

L'interface de la classe **`Recette`** est la suivante :

```
/** renvoie le nombre de sandwichs différents représentés dans l'arbre */
int nbSandwichs();

/** renvoie le nombre de sandwichs contenant l'ingrédient passé en paramètre représentés
dans l'arbre */
int nbSandwichs(Ingredient i);

/** supprime les branches qui ne contiennent pas de sandwich */
void grandNettoyage();

/** ajoute une recette dans l'arbre */
void sandwichDuMois(String nom, java.util.ArrayList<Ingredient> recette);
```

Exercice 4. (4,5 points) Écrire les deux méthodes **`nbSandwichs`** de la classe **`Recette`**. On rappelle qu'un sandwich peut contenir plusieurs fois le même ingrédient.

Exercice 5. (2,5 points) Écrire la méthode **`grandNettoyage`** de la classe **`Recette`** : toute branche de l'arbre qui ne contient pas de sandwich disparaît.

— le *grassouillet* : saumon, mayonnaise.

L'interface de la classe **Recette** est la suivante :

```
/** renvoie le nombre de sandwichs différents représentés dans l'arbre */
int nbSandwichs();

/** renvoie le nombre de sandwichs contenant l'ingrédient passé en paramètre représentés
dans l'arbre */
int nbSandwichs(Ingredient i);

/** supprime les branches qui ne contiennent pas de sandwich */
void grandNettoyage();

/** ajoute une recette dans l'arbre */
void sandwichDuMois(String nom, java.util.ArrayList<Ingredient> recette);
```

Exercice 4. (4,5 points) Écrire les deux méthodes **nbSandwichs** de la classe **Recette**. On rappelle qu'un sandwich peut contenir plusieurs fois le même ingrédient.

Exercice 5. (2,5 points) Écrire la méthode **grandNettoyage** de la classe **Recette** : toute branche de l'arbre qui ne contient pas de sandwich disparaît.

Exercice 6. (3,5 points) Écrire la méthode **sandwichDuMois** de la classe **Recette** : il s'agit d'ajouter un sandwich dont le nom et la liste dans l'ordre des ingrédients sont donnés en paramètre, en se contentant d'ajouter ce qui est nécessaire dans l'arbre des recettes, tout en tenant compte de ce qui existe déjà et sans modifier les autres recettes contenues dans cet arbre. On suppose que la classe **Sandwich** dispose d'un constructeur qui prend en argument une chaîne de caractères pour le nom et une instance de **Creation** pour la référence au nœud.

L'interface de la classe **Sandwich** est la suivante :

```
/** teste si tous les ingrédients pour le sandwich sont disponibles */
boolean ok();

/** affiche la liste des ingrédients dans l'ordre et les enlève des stocks, si le sandwich
peut être préparé */
void preparer();
```

Exercice 7. (2 points) Écrire la méthode **ok** de la classe **Sandwich**.

Exercice 8. (2,5 points) Écrire la méthode **preparer** de la classe **Sandwich** : celle-ci affiche la liste des ingrédients, dans l'ordre, pour faire le sandwich, s'il y en a suffisament en stock, et met le stock à jour ; sinon elle affiche un message disant qu'il faut faire les courses.

Exercice 5. (2,5 points) Écrire la méthode **`grandNettoyage`** de la classe **`Recette`** : toute branche de l'arbre qui ne contient pas de sandwich disparaît.

Exercice 6. (3,5 points) Écrire la méthode **`sandwichDuMois`** de la classe **`Recette`** : il s'agit d'ajouter un sandwich dont le nom et la liste dans l'ordre des ingrédients sont donnés en paramètre, en se contentant d'ajouter ce qui est nécessaire dans l'arbre des recettes, tout en tenant compte de ce qui existe déjà et sans modifier les autres recettes contenues dans cet arbre. On suppose que la classe **`Sandwich`** dispose d'un constructeur qui prend en argument une chaîne de caractères pour le nom et une instance de **`Creation`** pour la référence au nœud.

L'interface de la classe **`Sandwich`** est la suivante :

```
/** teste si tous les ingrédients pour le sandwich sont disponibles */
boolean ok();

/** affiche la liste des ingrédients dans l'ordre et les enlève des stocks, si le sandwich
peut être préparé */
void preparer();
```

Exercice 7. (2 points) Écrire la méthode **`ok`** de la classe **`Sandwich`**.

Exercice 8. (2,5 points) Écrire la méthode **`preparer`** de la classe **`Sandwich`** : celle-ci affiche la liste des ingrédients, dans l'ordre, pour faire le sandwich, s'il y en a suffisament en stock, et met le stock à jour ; sinon elle affiche un message disant qu'il faut faire les courses.