Travail pratique #1

Ce travail doit être fait individuellement.

1. Description et exigences logicielles

Il s'agit de concevoir et d'implémenter les classes de base pour la création du prototype d'un nouveau site de commerce électronique : Amizone. Ce dernier doit intégrer un système de recommandations élémentaire, basé sur les évaluations des utilisateurs ainsi que sur leur profil d'intérêts (leurs préférences).

Ce travail consiste en deux parties :

1. Modélisation et implémentation des utilisateurs du site.

2. Modélisation et implémentation élémentaire du fonctionnement du site et du système de recommandations.

1.1 CLASSES FOURNIES (À NE PAS MODIFIER)

La classe `Produit`

Cette classe modélise un produit qui peut être vendu par un fournisseur ou acheté par un consommateur. Elle contient, entre autres, les attributs d'instances suivants :

ATTRIBUT	DESCRIPTION	TYPE
code	le numéro d'identification unique de ce produit. Ce code doit être attribué automatiquement en utilisant la variable de classe seqId.	`int`
description	la description du produit *	`String`
categorie	la catégorie du produit. **	`String`
quantite	la quantité du produit ***	`int`
prix	le prix du produit	`double`
idFournisseur	le numéro d'identification (unique) du fournisseur de ce produit	`int`

* On suppose que la description du produit est formée d'une suite de mots, simplement séparés par des espaces et qu'elle ne contient aucun signe de ponctuation. Ex. : "table de chevet" (3 mots) ou "scie sauteuse" (2 mots) ou " livre " (1 mot).

** La catégorie du produit doit être l'un des termes contenus dans le tableau Produit.CATEGORIES. Ex : "ANIMALERIE", "FOUNITURE BUREAU", "OUTILS ET BRICOLAGE", etc. Cependant, la casse n'a pas d'importance. Par exemple, les termes "ANIMALERIE" ou "AniMAleRie" ou "animalerie" correspondent à la catégorie "ANIMALERIE" du tableau CATEGORIES, à l'indice 0.

*** La quantité du produit peut signifier deux choses : 1) lorsque c'est un produit vendu par un fournisseur, elle représente la quantité en stock et 2) lorsque c'est un produit acheté par un consommateur, elle représente la quantité achetée de ce produit.

Mélanie Lord
lord.melanie@uqam.ca

Vous ne devez pas modifier la classe Produit, car ce sera cette classe, telle quelle, qui sera utilisée pour tester votre programme.

La classe `TabUtils`

La classe `TabUtils` contient deux méthodes de classe utiles pour la manipulation de tableaux. Utilisez-les autant que possible.

Attention : Vous ne devez pas modifier les classes `Produit` et `TabUtils`, car ce seront ces classes, telles quelles, qui seront utilisées pour tester votre programme.

1.2 PARTIE 1 – LES UTILISATEURS D'AMIZONE.

Les utilisateurs d'Amizone sont de deux types : les consommateurs et les fournisseurs. Les consommateurs sont ceux qui achètent les produits vendus par les fournisseurs. Comme les consommateurs et les fournisseurs possèdent des caractéristiques communes (un nom d'utilisateur, un mot de passe…), vous décidez alors de créer une classe abstraite `Utilisateur` pour regrouper ces attributs communs. Les classes concrètes `Consommateur` et `Fournisseur`, qui modélisent les deux types d'utilisateurs, hériteront de cette classe abstraite.

À FAIRE :

A) Complétez la classe abstraite Utilisateur

Il s'agit ici de compléter la classe `Utilisateur.java`, fournie avec l'énoncé du TP1. Tout ce qui est déjà contenu dans cette classe ne DOIT PAS être modifié, mais vous devez y ajouter :

A.1 Les attributs d'instance suivants :

ATTRIBUT	DESCRIPTION	TYPE
id	numéro d'identification unique pour cet utilisateur	`int`
pseudo	le nom d'utilisateur (pseudonyme) de cet utilisateur	`String`
motPasse	le mot de passe de cet utilisateur	`String`
courriel	le courriel de cet utilisateur	`String`
evaluations	le tableau des évaluations reçues	`int []`
nbrEval	le nombre d'évaluations dans le tableau evaluations	`int`

***Note 1** : Puisqu'un tableau doit être déclaré avec une longueur fixe, pour faciliter la manipulation, on lui associe un nombre entier qui indique le nombre d'éléments effectivement dans le tableau (qui n'est pas nécessairement égal à la longueur du tableau). C'est ce à quoi sert l'attribut nbrEval. Ne pas oublier de mettre à jour cette variable lors de l'ajout d'éléments au tableau.*

A.2 Les constructeurs suivants :

```
/**
 * Construit un Utilisateur avec le pseudo, le motPasse et le courriel
 * donnes en paramètres. L'attribut id (numero unique) est assigne
 * automatiquement. La longueur du tableau evaluations est initialisee
 * a LONG_TAB et nbrEval est a 0.
 * @param pseudo le nom d'utilisateur (pseudonyme) de cet utilisateur.
 * @param motPasse le mot de passe de cet utilisateur
 * @param courriel le courriel de cet utilisateur
 *
 * ANTECEDENTS : le pseudo, le motPasse et le courriel doivent etre
 *               valides (non null, non vides, correctement formes,
 *               etc.). On suppose aussi que ces valeurs sont uniques
 *               pour chaque utilisateur cree.
 */
public Utilisateur (String pseudo, String motPasse, String courriel) {...}
```

***Note 2** : L'étiquette "ANTECEDENTS" signifie que les conditions qui y sont mentionnées sont supposées vraies avant l'appel du constructeur. En d'autres termes, le constructeur ne prend pas la responsabilité de les valider.*

Ajouter aussi le constructeur de copie suivant, tel quel. Il sera possiblement utilisé dans les tests.

```
/**
 * Constructeur de copie.  POUR TESTS SEULEMENT.
 * @param utilisateur l'utilisateur dont on veut une copie.
 */
public Utilisateur (Utilisateur utilisateur) {
   this.pseudo = utilisateur.pseudo;
   this.motPasse = utilisateur.motPasse;
   this.courriel = utilisateur.courriel;
   this.evaluations = utilisateur.evaluations;
   this.nbrEval = utilisateur.nbrEval;
   this.id = utilisateur.id;
}
```

A.3 Les 2 méthodes d'instance abstraites (publiques) suivantes :

`compilerProfil`
Cette méthode retourne un tableau de type `String` représentant le profil de l'utilisateur et ne prend aucun paramètre.

`evaluer`
Cette méthode permet à un utilisateur d'évaluer un autre utilisateur. En effet, les utilisateurs du site Amizone peuvent s'évaluer entre eux. Un fournisseur (respectivement consommateur) peut évaluer un consommateur (respectivement fournisseur) en lui attribuant une note d'appréciation (une évaluation) basée sur l'expérience d'une transaction conclue avec ce consommateur (respectivement fournisseur). Une évaluation valide est une note comprise entre 1 et 5 où 5 constitue la meilleure évaluation.

Cette méthode prend donc comme paramètres (et dans cet ordre) : l'utilisateur évalué (type `Utilisateur`), ainsi que l'évaluation donnée à l'utilisateur (type `int`).

De plus, cette méthode doit lever une Exception (type `Exception`) lorsque 1) l'utilisateur évalué n'a jamais conclu de transaction avec l'utilisateur qui l'évalue ou lorsque 2) l'évaluation donnée en paramètre n'est pas valide.

A.4 Les 2 méthodes d'instance publiques suivantes :

`evaluationMoyenne`
Cette méthode calcule et retourne la moyenne de toutes les évaluations de cet utilisateur (contenues dans le tableau evaluations). Elle ne prend aucun paramètre et retourne un nombre (type `double`), arrondi à deux décimales. Évidemment, si l'utilisateur n'a reçu aucune évaluation, la moyenne retournée est 0.

Pour arrondir le résultat à deux décimales, vous pouvez utiliser l'instruction suivante :

```
(double)Math.round(moyenne * 100) / 100;
```

`ajouterEvaluation`
Un utilisateur peut recevoir différentes évaluations provenant de divers utilisateurs. Cette méthode permet donc d'ajouter une nouvelle évaluation reçue au tableau des évaluations de cet utilisateur. Cette méthode ne retourne rien et prend seulement une évaluation (type `int`) en paramètre. C'est cette évaluation qui sera ajoutée au tableau des évaluations déjà reçues.

Cette méthode doit lever une Exception (type `Exception`) contenant le message `Utilisateur.MSG_ERR_EVAL_3` lorsque l'évaluation donnée en paramètre n'est pas valide.

A.5 Les *getters* et *setters* suivants :

- Les *getters* pour les 6 attributs d'instance (`getId`, `getPseudo`, `getMotPasse`, `getCourriel`, `getEvaluations`, `getNbrEval`).

- Les *setters* pour les **3** attributs d'instance pseudo, motPasse et courriel (`setPseudo`, `setMotPasse`, `setCourriel`).

***Note 4** : Les valeurs passées en paramètres pour les 3 setters n'ont pas à être validées. On les suppose valides, comme dans le cas du constructeur (voir ANTECEDENTS).*

A.5 La redéfinition de la méthode `equals` :

Deux utilisateurs sont considérés comme étant égaux s'ils ont le même `id`.

A.6 La redéfinition de la méthode `toString()`:

Ajoutez, telle quelle, la méthode suivante. Elle sera possiblement utilisée pour les tests.

```
/**
 * Retourne une representation de cet utilisateur sous forme d'une chaine
 * de caracteres (son id, son pseudonyme, son mot de passe, son courriel
 * et le nombre de ses evaluations reçues).
 * @return une representation de cet utilisateur sous forme d'une chaine de
 *         caracteres.
 */
public String toString() {
     return id + " : " + pseudo + " - " + motPasse + " - " + courriel
             + " - " + nbrEval;
}
```

Vous pouvez ajouter des constantes de classe publiques dans la classe `Utilisateur`, mais RIEN d'AUTRE.

B) Créez la classe Consommateur

La classe `Consommateur` est une classe concrète qui hérite de la classe `Utilisateur`. Elle doit contenir exactement :

B.1 Les attributs d'instance suivants :

ATTRIBUT	DESCRIPTION	TYPE
achats	tableau des produits achetés par ce consommateur	`Produit[]`
nbrAchats	le nombre de produits achetés (présents dans le tableau achats)	`int`

***Note 5** : Ne pas oublier de mettre à jour la variable nbrAchats lors de l'ajout d'éléments au tableau achats.*

B.2 Les constructeurs suivants :

```
/**
 * Construit un Consommateur avec le pseudo, le motPasse et le courriel
 * passes en paramètres. L'attribut id (numero unique) est assigne
 * automatiquement. La longueur du tableau des achats de ce consommateur
 * est initialisee a Utilisateur.LONG_TAB et nbrAchats est a 0. La longueur
 * du tableau evaluations est initialisee a Utilisateur.LONG_TAB et nbrEval
 * est a 0.
 * @param pseudo le nom d'utilisateur (pseudonyme) de ce consommateur
 * @param motPasse le mot de passe de ce consommateur
 * @param courriel le courriel de ce consommateur
 *
 * ANTECEDENTS : le pseudo, le motPasse et le courriel doivent être
 *               valides (non null, non vides, correctement formes,
 *               etc.). On suppose aussi que ces valeurs sont uniques
 *               pour chaque utilisateur cree.
 */
public Consommateur (String pseudo, String motPasse, String courriel) {...}
```

Ajouter aussi, tel quel, le constructeur de copie suivant (pour tests seulement).

```
/**
 * Constructeur de copie. Pour tests seulement.
 * @param consommateur le consommateur dont on veut une copie.
 */
public Consommateur (Consommateur consommateur) {
   super(consommateur);
}
```

B.3 L'implémentation des 2 méthodes abstraites de la classe `Utilisateur` :

En plus des informations données à la section A.3, voici des informations supplémentaires pour l'implémentation des méthodes abstraites dans la classe Consommateur :

`compilerProfil`
Le profil d'un consommateur représente ses préférences, ce qu'il aime consommer. Il est possible d'obtenir cette information en se basant sur les achats déjà effectués par ce consommateur. Dans le cadre de ce travail, la compilation du profil d'un consommateur consiste à recenser toutes les catégories de produits (voir attribut catégorie dans classe `Produit`) qu'il a achetés par le passé.

Cette méthode retourne donc un tableau contenant toutes les catégories différentes recensées sur tous les produits achetés par ce consommateur (produits dans le tableau des achats). De plus, le tableau retourné ne doit contenir aucun doublon (une même catégorie ne peut apparaître plus d'une fois dans le tableau) et doit être de longueur minimale : la longueur du tableau doit être égale au nombre de catégories dans le tableau. Si ce consommateur n'a encore acheté aucun produit, la méthode doit retourner la valeur `null`.

`evaluer`
Dans la classe `Consommateur`, cette méthode permet à ce consommateur d'évaluer un fournisseur (reçu en paramètre). Le paramètre évaluation est la note d'appréciation attribuée au fournisseur évalué.

De plus, cette méthode doit lever certains types d'Exception dans différentes circonstances :

- `ClassCastException` : si l'utilisateur passé en paramètre n'est pas de type `Fournisseur`.
- `NullPointerException` : si l'utilisateur passé en paramètre est `null`.
- `Exception` avec le message `Utilisateur.MSG_ERR_EVAL_2` : si le fournisseur passé en paramètre n'a jamais vendu de produit(s) à ce consommateur.
- `Exception` avec le message `Utilisateur.MSG_ERR_EVAL_3` : si l'évaluation passée en paramètre est invalide.

Si aucune Exception n'est levée, l'évaluation passée en paramètre est ajoutée au tableau des évaluations du fournisseur passé en paramètre.

***Note 6** : Pour ne pas trop complexifier le travail, on ne conservera aucune information sur l'identité des évaluateurs.*

B.4 Les *getters* suivants :

Deux *getters* (**`getAchats`** et **`getNbrAchats`**) pour les 2 attributs achats et nbrAchats.

B.5 Les méthodes d'instance publiques suivantes :

`acheter`
Cette méthode consiste à ajouter un produit au tableau des achats. Elle ne retourne rien et prend en paramètres (dans cet ordre) : un produit (type `Produit`) et une quantité achetée (type `int`).

De plus, cette méthode lève les Exceptions suivantes :

- `ExceptionProduitInvalide` : si le produit donné est `null`.
- `Exception` (avec le message `Utilisateur.MSG_ERR_ACHAT`) : si le produit n'est vendu par aucun fournisseur (idFournisseur = 0). Ceci signifie qu'un produit acheté par un consommateur doit toujours provenir d'un fournisseur et doit donc posséder un idFournisseur > 0.
- `Exception` (avec le message `Utilisateur.MSG_ERR_QTE`) si la quantité achetée est plus petite ou égale à 0 ou si elle est plus grande que la quantité en stock du produit.

Comme le produit reçu en paramètre est un produit vendu par un fournisseur, c'est une **copie** (voir constructeur de copie) de ce produit qui doit être ajouté à la liste des achats de ce consommateur (lorsqu'il n'y a aucune exception).

On doit aussi remplacer la quantité de cette copie du produit avec la quantité achetée reçue en paramètre puisque dans le tableau des achats, la quantité du produit représente la quantité achetée (contrairement aux tableaux des produits d'un fournisseur où la quantité de chaque produit représente la quantité en stock). Ne pas oublier d'ajuster l'attribut nbrAchats.

***Note 7** : Pour ne pas trop complexifier le travail, on ne peut acheter qu'un seul produit à la fois, mais de la quantité voulue (>0).*

***Note 8** : Un consommateur peut acheter le même produit plusieurs fois (cela compte pour différents achats).*

`fournisseurs`
Cette méthode ne prend aucun paramètre et retourne un tableau d'entiers (type `Integer []`) de tous les numéros d'identification des fournisseurs de qui ce consommateur a acheté un ou des produits. Un même fournisseur doit apparaître une seule fois dans le tableau retourné et celui-ci doit être de longueur minimale (de même longueur que le nombre de fournisseurs qui s'y trouvent).

Si le consommateur n'a encore fait aucun achat, la méthode retourne `null`.

Redéfinition de la méthode `toString()`
Ajouter la méthode suivante telle quelle. Elle sera possiblement utilisée pour les tests.

```
/**
 * Retourne une representation sous forme de chaine de caracteres
 * de ce consommateur.
 * @return une representation sous forme de chaine de caracteres
 *         de ce consommateur.
 */
public String toString() {
   return super.toString() + " - " + nbrAchats;
}
```

Vous pouvez ajouter des méthodes privées ou des constantes de classe publiques dans la classe `Consommateur`, mais RIEN D'AUTRE.

C) Créer la classe Fournisseur

La classe `Fournisseur` est une classe concrète qui hérite de la classe `Utilisateur`. Elle doit contenir exactement :

C.1 Les attributs d'instance suivants :

ATTRIBUT	DESCRIPTION	TYPE
produits	tableau des produits vendus par ce fournisseur	`Produit[]`
nbrProduits	le nombre de produits vendus (présents dans le tableau des produits)	`int`

***Note 9** : Ne pas oublier de mettre à jour la variable nbrProduits lors de l'ajout d'éléments au tableau produits.*

C.2 Les constructeurs suivants :

```
/**
 * Construit un Fournisseur avec le pseudo, le motPasse et le courriel
 * donnes en parametres. L'attribut id (numero unique) est assigne
 * automatiquement. La longueur du tableau des produits vendus par ce
 * fournisseur est initialisee a Utilisateur.LONG_TAB et nbrProduits
 * est a 0. La longueur du tableau evaluations est initialisee a
 * Utilisateur.LONG_TAB et nbrEval est a 0.
 * @param pseudo le nom d'utilisateur (pseudonyme) de ce fournisseur.
 * @param motPasse le mot de passe de ce fournisseur
 * @param courriel le courriel de ce fournisseur
 *
 * ANTECEDENTS : le pseudo, le motPasse et le courriel sont des valeurs
 *               valides non vides et non egales a null. On suppose aussi
 *               que ces valeurs sont uniques pour chaque utilisateur.
 */
public Fournisseur (String pseudo, String motPasse, String courriel) {...}
```

Ajouter aussi, tel quel, le constructeur de copie suivant (pour tests seulement).

```
/**
 * Constructeur de copie. Pour tests seulement.
 * @param fournisseur le fournisseur dont on veut une copie.
 */
public Fournisseur (Fournisseur fournisseur) {
   super(fournisseur);
}
```

C.3 L'implémentation des 2 méthodes abstraites de la classe `Utilisateur` :

En plus des informations données à la section A.3, voici des informations supplémentaires pour l'implémentation des méthodes abstraites dans la classe Fournisseur :

`compilerProfil`
Le profil d'un fournisseur représente sa spécialité, les types de produits qu'il vend. Il est possible d'obtenir cette information en se basant sur les produits vendus par ce fournisseur. Dans le cadre de ce travail, la compilation du profil d'un fournisseur consiste à recenser toutes les catégories des produits (voir attribut catégorie dans classe `Produit`) qu'il vend.

Cette méthode retourne donc un tableau contenant toutes les catégories différentes recensées sur les produits vendus par ce fournisseur (produits dans le tableau des produits). Attention, seuls les produits dont la quantité en stock (voir attribut quantite de la classe `Produit`) est strictement plus grande que 0 sont considérés pour recenser les catégories.

De plus, le tableau retourné ne doit contenir aucun doublon (une même catégorie ne peut apparaître plus d'une fois dans le tableau) et doit être de longueur minimale : la longueur du tableau doit être égale au nombre de catégories dans le tableau. Si ce fournisseur ne vend aucun produit de quantité supérieure à 0, la méthode doit retourner la valeur `null`.

`evaluer`
Dans la classe Fournisseur, cette méthode permet à ce fournisseur d'évaluer un consommateur (reçu en paramètre). Le paramètre évaluation est la note d'appréciation attribuée au consommateur évalué.

De plus, cette méthode doit lever certains types d'Exception dans différentes circonstances :

- `ClassCastException` : si l'utilisateur passé en paramètre n'est pas de type `Consommateur`.
- `NullPointerException` : si l'utilisateur passé en paramètre est `null`.
- `Exception` avec le message `Utilisateur.MSG_ERR_EVAL_1` : si le consommateur passé en paramètre n'a jamais acheté de produit(s) de ce fournisseur.
- `Exception` avec le message `Utilisateur.MSG_ERR_EVAL_3` : si l'évaluation passée en paramètre est invalide.

Si aucune Exception n'est levée, l'évaluation passée en paramètre est ajoutée au tableau des évaluations du consommateur passé en paramètre.

(*Voir aussi Note 6*)

C.4. Les *getters* suivants :

Deux *getters* : `getProduits` et `getNbrProduits` pour les 2 attributs produits et nbrProduits.

C.4 Les méthodes d'instance publiques suivantes :

`ajouterNouveauProduit`
Cette méthode permet d'ajouter un nouveau produit à vendre dans le tableau des produits de ce fournisseur, si celui-ci n'y est pas déjà. Elle ne retourne rien et reçoit trois paramètres, dans cet ordre :

- un produit (type `Produit`), le nouveau produit à ajouter au tableau des produits.
- une quantité en stock (type `int`), la quantité en stock initiale de ce nouveau produit à vendre.
- un prix de vente (type `double`), le prix de vente de ce nouveau produit à vendre.

De plus, cette méthode doit lever divers types d'exceptions dans les cas suivants :

- `ExceptionProduitInvalide` : si le produit donne en paramètre est `null`.
- `Exception` (avec le message d'erreur `Utilisateur.MSG_ERR_QTE`) : si la quantité en stock donnée est plus petite ou égale à 0. Lors de l'ajout d'un produit à vendre, celui-ci doit avoir une quantité en stock > 0.
- `Exception` (avec le message d'erreur `Utilisateur.MSG_ERR_PRIX`) : si le prix de vente donné est plus petit ou égal à 0. Lors de l'ajout d'un produit à vendre, on doit lui assigner un prix valide (>0).
- `Exception` (avec le message d'erreur `Utilisateur.MSG_ERR_AJOUT_PROD`) : si le produit donné en paramètre existe déjà dans le tableau des produits de ce fournisseur.

Comme plusieurs fournisseurs peuvent vendre les mêmes produits, mais avec des prix différents et des quantités en stock diverses, le produit qui sera ajouté au tableau des produits doit être une **copie** du produit passé en paramètre.

Finalement, on assigne les valeurs aux attributs idFournisseur (ce fournisseur), quantite (quantité en stock donnée) et prix (le prix de vente donne) de la copie du produit qu'on ajoutera ensuite au tableau des produits (si pas d'exception, évidemment). Ne pas oublier d'ajuster la valeur de nbrProduits.

`obtenirProduit`
Cette méthode recherche, dans le tableau des produits de ce fournisseur, le produit ayant le code donné. Elle prend donc en paramètre un code de produit (type `int`) et retourne le produit (type `Produit`) s'il a été trouvé, sinon elle retourne la valeur `null`.

`vendre`
Cette méthode permet de diminuer la quantité en stock d'un produit vendu. Elle ne retourne rien et prend deux paramètres (dans cet ordre) : le code du produit dont on veut diminuer la quantité en stock (type `int`) et la quantité vendue (type `int`) qu'on veut soustraire à la quantité en stock du produit donné.

De plus, elle doit lever certaines exceptions dans les cas suivants :

- `Exception` (avec le message `Utilisateur.MSG_ERR_VENTE_PROD`) : si le code du produit donné ne correspond à aucun des produits vendus par ce fournisseur.

- `Exception` (~~avec le message `Utilisateur.MSG_ERR_VENTE_PROD`~~) (avec le message `Utilisateur.MSG_ERR_QTE`) : si la quantité vendue donnée est plus petite ou égale à 0 ou si elle est plus grande que la quantité en stock du produit ayant le code donné.

Redéfinition de la méthode `toString()`
Ajouter la méthode suivante telle quelle. Elle sera possiblement utilisée pour les tests.

```
/**
 * Retourne une representation sous forme de chaine de caracteres de ce
 * Fournisseur.
 * @return une representation sous forme de chaine de caracteres de ce
 *         Fournisseur.
 */
public String toString() {
   return super.toString() + " - " + nbrProduits;
}
```

Vous pouvez ajouter des méthodes privées ou des constantes de classe publiques dans la classe `Fournisseur`, mais RIEN D'AUTRE.

1.3 PARTIE 2 – LE FONCTIONNEMENT DU SITE

Pour la partie 2, vous devez créer la classe `Amizone` qui contiendra les méthodes de gestion du site Amizone. Cette classe met donc en relation les consommateurs et les fournisseurs. Elle permet, entre autres, à un utilisateur de s'inscrire au site dans le but de consommer des produits vendus par des fournisseurs ou de vendre des produits aux consommateurs. Elle fournit aussi des services de recommandations (d'utilisateurs et de produits) ainsi que des services de recherche de produits.

La classe `Amizone` doit contenir :

D.1 Les constantes de classes publiques suivantes (mais vous pouvez en ajouter d'autres si requis) :

```
public final static String ERR_MSG_UTILIS_NULL = "Erreur, utilisateur null.";
public final static String ERR_MSG_FOURN_AUCUN_PRODUIT =
      "Erreur, ce fournisseur ne vend aucun produit.";
public final static String ERR_MSG_UTILIS_EXISTANT =
      "Erreur, cet utilisateur existe deja.";
```

D.2 Un seul attribut d'instance :

ATTRIBUT	DESCRIPTION	TYPE
utilisateurs	liste des utilisateurs d'Amizone.	`ArrayList<Utilisateur>`

D.3 Constructeur :
Un constructeur sans argument qui instancie la liste des utilisateurs. La liste doit être initialement vide.

D.4 Les méthodes d'instance publiques suivantes :

`public ArrayList<Utilisateur> getUtilisateurs() {...}`
Cette méthode sans argument retourne la liste des utilisateurs (*getter*).

`public void inscrireUtilisateur(Utilisateur utilisateur)`
`throws Exception {…}`
Cette méthode ajoute l'utilisateur donné à la liste des utilisateurs d'Amizone. L'ajout doit se faire EN FIN DE LISTE. Si l'utilisateur donné est un fournisseur, celui-ci doit vendre au moins un produit dont la quantité (en stock) est plus grande que 0.

Cette méthode lève :

- une `Exception` avec le message d'erreur ERR_MSG_UTILIS_NULL si l'utilisateur donné est null.
- une `Exception` avec le message d'erreur ERR_MSG_FOURN_AUCUN_PRODUIT si l'utilisateur donné est un fournisseur (type Fournisseur) et qu'il ne vend aucun produit dont la quantité est strictement plus grande que 0.
- une `Exception` avec le message d'erreur ERR_MSG_UTILIS_EXISTANT si l'utilisateur donné est déjà dans la liste des utilisateurs d'Amizone.

~~**`public ArrayList<Utilisateur> recommanderUtilisateur`**~~
~~**`(Utilisateur utilisateur) throws Exception {...}`**~~
`public ArrayList<Utilisateur> recommanderUtilisateurs`
`(Utilisateur utilisateur) throws Exception {`
Cette méthode fait partie du système de recommandations. Elle permet de recommander à un consommateur des fournisseurs potentiellement intéressants pour ce consommateur ou bien de recommander à un fournisseur des consommateurs potentiellement intéressés par les produits que vend ce fournisseur. Cette recommandation se fait sur la base des profils des utilisateurs. On dira que deux utilisateurs qui ont des profils similaires sont potentiellement intéressants l'un pour l'autre. Pour les besoins de ce petit prototype, nous dirons que deux utilisateurs ont des profils similaires s'ils ont au moins une catégorie en commun dans leur profil (voir méthode `compilerProfil` des classes `Fournisseur` et `Consommateur`).

Cette méthode retourne donc une liste d'utilisateurs potentiellement intéressants pour l'utilisateur donné. Si l'utilisateur donné est un `Consommateur`, la liste retournée doit contenir tous les fournisseurs (type `Fournisseur`) qui ont des profils similaires à l'utilisateur (`Consommateur`) donné. Si l'utilisateur donné est un `Fournisseur`, la liste retournée doit contenir tous les consommateurs (type `Consommateur`) qui ont des profils similaires à l'utilisateur (`Fournisseur`) donné. Lorsqu'aucun utilisateur de profil similaire n'est trouvé, la méthode retourne une liste vide.

De plus, cette méthode lève :

- Une Exception avec le message d'erreur ERR_MSG_UTILIS_NULL si l'utilisateur donné est `null`.

`public ArrayList<Produit> recommanderProduits`
`(Fournisseur fournisseur, Consommateur consommateur)`
`throws Exception {...}`
Cette méthode fait aussi partie du système de recommandations et retourne une liste de tous les produits vendus par le fournisseur donné qui sont potentiellement intéressants pour le consommateur donné. Les produits potentiellement intéressants pour le consommateur donné sont ceux qui possèdent une catégorie qui est présente dans le profil (voir méthode `compilerProfil`) du consommateur.

De plus, la liste retournée ne doit contenir que les produits dont la quantité en stock, chez le fournisseur donné, est strictement supérieure à 0.

Si aucun produit potentiellement intéressant de quantité > 0 n'est trouvé, la méthode retourne une liste vide.

De plus, cette méthode lève :

- une Exception avec le message d'erreur ERR_MSG_UTILIS_NULL si le fournisseur ou le consommateur donné est `null`.

```
public void effectuerTransaction(Fournisseur fournisseur,
            Consommateur consommateur, int codeProduit, int quantite)
            throws Exception {...}
```

Cette méthode permet d'effectuer une transaction achat/vente entre un fournisseur et un consommateur. Elle permet au fournisseur donné de vendre le produit du code donné, de la quantité donnée, au consommateur donné (autrement dit, permet au consommateur donné, d'acheter le produit du code donné, de la quantité donnée, et du fournisseur donné).

La quantité donnée représente donc la quantité vendue (du point de vue du fournisseur) ou la quantité achetée (du point de vue du consommateur).

Après la transaction, une **copie** (voir constructeur de copie) du produit acheté par le consommateur donné doit avoir été ajoutée à la liste de ses achats (lorsqu'il n'y a aucune exception). De plus, la quantité de cette copie du produit doit être remplacée par la quantité achetée (quantité donnée).

Aussi, la quantité en stock du produit vendu, chez le fournisseur donné, doit être ajustée en lui soustrayant la quantité vendue.

De plus, cette méthode lève :

- une `Exception` (avec le message `Utilisateur.MSG_ERR_VENTE_PROD`) : si le code du produit donné ne correspond à aucun des produits vendus par le fournisseur donné.
- une `Exception` (avec le message `Utilisateur. MSG_ERR_QTE`) : si la quantité donnée est plus petite ou égale à 0 ou si elle est plus grande que la quantité en stock du produit ayant le code donné, chez le fournisseur donné.

Note 10 : *Voir méthode `acheter` (de la classe `Consommateur`) et `vendre` (de la classe `Fournisseur`).*

```
public ArrayList<Produit> rechercherProduitsParMotCle(String motCle) {...}
```

Cette méthode retourne une liste de tous les produits (parmi les produits vendus par tous les fournisseurs) dont la description contient le mot clé donné. Pour simplifier la recherche, nous supposerons ici que la description des produits est une suite de mots séparés uniquement par un ou des espaces (aucun signe de ponctuation) et que le mot clé donné est un seul mot, sans espace. La recherche ne doit pas tenir compte de la casse.

Les produits retournés doivent avoir une quantité en stock strictement plus grande que 0.

Un même produit peut revenir plusieurs fois dans la liste retournée si celui-ci est vendu par différents fournisseurs (et qu'il correspond à la recherche, évidemment). Si aucun produit n'est trouvé, la méthode retourne une liste vide.

~~`public ArrayList<Fournisseur> rechercherFournisseurs(int codeProduit) {...}`~~
`public ArrayList<Fournisseur> rechercherFournisseurParEvaluation (int codeProduit) {`

Cette méthode retourne une liste des fournisseurs qui vendent le produit du code donné et dont la quantité (en stock) est strictement plus grande que 0. De plus, les fournisseurs retournés doivent être ordonnés selon leur évaluation (en ordre décroissant des évaluations : du meilleur (eval 5) au pire (eval 1)) . Si aucun fournisseur n'est trouvé, la méthode retourne une liste vide.

Note 11 *: Pour la partie 2, réutilisez autant que possible les méthodes des classes de la partie 1.*

Note 12 *: Vous ne devez rien ajouter d'autre à la classe* `Amizone`*, à part des méthodes d'instance privées (pour la séparation fonctionnelle) ou des constantes de classe publiques.*

Note 13 *: Assurez-vous de bien respecter les entêtes des méthodes données. Si vous ne respectez pas le contrat spécifié, les tests ne compileront pas et vous perdrez des points pour non-respect des spécifications.*

Note 14 *: Vous pouvez utiliser le gabarit de la classe Amizone fourni avec l'énoncé du TP (sur Moodle).*

2. Précisions

Vous devez respecter le principe d'encapsulation des données.

Pour tous les tableaux créés, on suppose que la longueur spécifiée (LONG_TAB = 100) est suffisante. Vous n'avez pas à vérifier s'il reste de la place dans le tableau avant d'y ajouter un élément.

Les tableaux créés doivent contenir leurs éléments, de manière consécutive, à partir du début du tableau. Par exemple, si un tableau de longueur 100 contient 3 éléments, les 3 éléments doivent se trouver aux indices 0, 1 et 2. De plus, lorsque vous ajoutez un élément dans un tableau, vous devez l'ajouter à l'indice du dernier élément ajouté + 1.

Toutes vos classes doivent se trouver dans le paquetage par défaut.

Utilisez des constantes autant que possible.

Réutiliser le code autant que possible.

Utilisez les méthodes fournies dans les diverses classes autant que possible.

Il est fortement recommandé de faire des méthodes privées pour bien structurer/modulariser le code (séparation fonctionnelle).

Votre code doit compiler et s'exécuter avec le JDK 6.

Il ne doit y avoir aucun affichage dans vos méthodes (pas de `System.out`)

Toutes les classes doivent se trouver dans le **paquetage par défaut**.

Si quelque chose est ambigu, obscure, s'il manque de l'information, si vous ne comprenez pas les spécifications, si vous avez des doutes… vous avez la responsabilité de vous informer auprès de votre enseignante.

3. Détails sur la correction

Votre code sera noté selon deux aspects :

Le code (40 points)

Concernant les critères de correction du code, lisez attentivement le document "**critères généraux de correction du code Java**" dans la section TRAVAUX sur Moodle.

L'exécution (60 points)

partie 1 (30 points)

partie 2 (30 points)

Votre code sera testé en tout ou en partie. Les points seront calculés sur les tests effectués. Ceci signifie que si votre code fonctionne dans un cas x et que ce cas x n'est pas testé, vous n'obtiendrez pas de points pour ce cas. Il est donc dans votre intérêt de vous assurer du bon fonctionnement de votre programme dans tous les cas possibles.

Note : Un travail qui ne compile pas se verra attribuer la note 0 pour l'exécution.

3. Date et modalités de remise

Date de remise : **21 février** 2014 avant minuit.

Politique concernant les retards

Une pénalité de 10% de la note finale, par jour de retard, sera appliquée aux travaux remis après la date limite. La formule suivante sera utilisée pour calculer la pénalité pour les retards :

`Nbr points de pénalité = m / 144,` où `m` est le nombre de minutes de retard par rapport à l'heure de remise. Ceci donne 10 points de pénalité pour 24 heures de retard, 1.25 point de pénalité pour 3 heures, etc.

Aucun travail ne sera accepté après 3 jours de retard et la note attribuée sera 0.

Les 4 fichiers à remettre

- Utilisateur.java (classe complétée)
- Consommateur.java
- Fournisseur.java
- Amizone.java (partie 2)

Remise via Moodle uniquement (menu gauche, section Activités, lien Devoirs)

Vous devez remettre (téléverser) vos quatre fichiers (.java), NON ZIPPÉS, sur le site du cours (Moodle), en cliquant sur le lien Devoirs, dans la section Activités du menu de gauche, puis en cliquant ensuite sur REMISE DU TP1.

Remarques générales

- Identifiez tout document avec votre code permanent.
- Les règlements sur le plagiat seront strictement appliqués.
- Aucun programme reçu par courriel ne sera accepté.

Ce travail est strictement individuel. Le règlement sur le plagiat sera appliqué sans exception. Vous devez ainsi vous assurer de ne pas échanger du code avec des collègues ni de laisser sans surveillance votre travail au laboratoire. Vous devez également récupérer rapidement toutes vos impressions de programme au laboratoire.