TD 1

Ce document comporte 7 pages.

1. Salle de cinéma

L'objectif de cet exercice est d'écrire une classe `SalleCinema`. Cette classe est schématiquement décrite dans le diagramme de la figure 1

- A la création d'une instance de la classe, il faut préciser :
  - le film projeté dans la salle,
  - la capacité de la salle,
  - le prix d'un billet d'entrée.
- La méthode `vendrePlace()` correspondra à la vente d'une place. On supposera que la salle n'est pas pleine.
- Les méthodes `tauxRemplissage()` et `chiffreAffaires()` indiquent, respectivement, la proportion de places vendues par rapport à la capacité de la salle et le chiffre d'affaires de la salle.
- La méthode `estPleine()` indique si la salle est pleine ou non.
- La méthode `nbPlacesDisponibles()` indique le nombre de places vides.
- La méthode `toString()` retourne une chaîne de caractères construite de telle sorte que le résultat de l'instruction `System.out.println(salle)` soit (on suppose que `salle` est une variable de type `SalleCinema`) :

```
Film projete       : Java
Tarif              : 7.5
Nombre de places   : 1000
Taux remplissage   : 0.933
Chiffre d'affaires : 6997.5
```

2. Calcul de paye

Une entreprise EnterpriseJ, souhaite mettre en place une application pour calculer la paye hebddmadaire de ses salariés. Chaque employé est payé à l'heure. La durée légale du travail est de 40 heures hebdomadaires. Les heures supplémentaires sont payées 1,5 fois de plus que l'heure de base. Une heure de base est payée au minimum à 13 euros. Les salariés ne doivent pas dépasser 60 heures hebdomadaires.

Les règles de gestion suivantes doivent être respectées :

- Jusqu'à 40 heures, le salaire hebdomadaire est : Nombre d'heures travaillées x salaire de base
- Pour chaque heure au-delà de 40, l'heure supplémentaire est : 1,5 x heure de base

SalleCinema	
-	film
+	nbPlaces
-	tarif
-	nbPlacesVendues
+	vendrePlace()
+	nbPlacesDisponibles()
+	estPleine()
+	tauxRemplissage()
+	chiffreAffaires()
+	versChaine()

FIGURE 1 – Description de la classe `SalleCinema`

— L'heure de base ne doit pas être en dessous de 13 euros. Sinon, l'application affiche une erreur
— Si le nombre d'heures travaillées hebdomadaires dépasse 60, l'application affiche une erreur.

Question 1. Écrivez une classe `Employe` caractérisée par le nom de l'employé, son salaire horaire de base et le nombre d'heures travaillées hebdomadaires.

Question 2. Écrivez une classe `EntrepriseJ` contenant la méthode `main`, pour calculer et afficher le salaire hebdomadaire des employés suivant, ou affichez une erreur si une règle de gestion n'est pas respectée.

— John, salaire horaire : 7.5 euros ; heures travaillées : 35
— Cécile, salaire horaire : 8.2 euros ; heures travaillées : 47
— Pierre, salaire horaire : 10.00 euros ; heures travaillées : 73
— Laurent, salaire horaire : 25.00 euros ; heures travaillées : 60
— Laurent, salaire horaire : 14.00 euros ; heures travaillées : 45

Question 3. Modifier le constructeur de la classe `Employe` de telle sorte qu'on ne puisse pas embaucher un employé pour lequel les règles de gestion ne sont pas respectées. En d'autres mots, l'instanciation de la classe ne doit pas réussir si au moins une règle de gestion n'est pas respectée.

3. Automates

Exercice 3.1. Dans cet exercice, nous souhaitons travailler sur des automates finis. Un automate fini est une machine composée de :

— un nombre fini d'états dont
  — un unique état de départ,
  — et un ensemble d'états finaux,
— un ensemble fini de transitions.

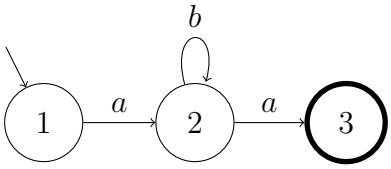


FIGURE 2 – Exemple d'automate : il reconnaît la suite d'actions $a(b)^*a$.

Avant de commencer cet exercice, on récupérera le projet **AF.zip** qui se trouve avec l'énoncé. Ce squelette contient les classes suivantes :

- la classe *Etat* qui représente un état d'automate fini,
- la classe *EtatFinal* pour les états finaux,
- la classe *AutomateFini* pour les automates,
- la classe *AFException* pour les exceptions sur les automates.

Question 4.

Il manque dans le squelette une implémentation des transitions. Pour cela il faut définir une nouvelle classe *Transition* comportant :

- un attribut *suivant* pour l'état suivant le franchissement de la transition,
- un attribut *etiquette* qui est un objet quelconque servant d'étiquette de transition,
- un constructeur prenant en argument une étiquette et un état suivant,
- un accesseur pour l'étiquette,
- une méthode $franchir()$ qui retourne l'état suivant.

Question 5.

On veut ajouter un contrat sur le constructeur de transition. Pour cela, on identifie les pré-requis suivants :

- l'étiquette fournie à la construction ne doit pas être null,
- l'état suivant ne doit pas être null.

Écrire les deux pré-requis et lever une exception *TransitionException* (classe à écrire), si l'un des pré-requis n'est pas valide à l'appel du constructeur.

Question 6.

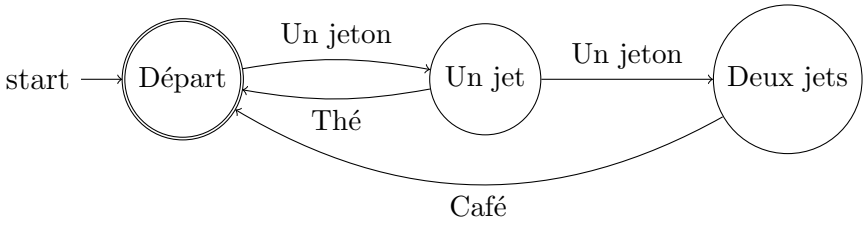
Il manque dans la classe *AutomateFini* un contrat pour la méthode *step(Object entree)* qui permet, a partir de l'état courant, de tenter de franchir une transition dont l'étiquette correspond à l'entrée fournie. Les pré-requis que l'on désire mettre en place sont :

- l'entrée n'est pas un objet null,
- la machine est en cours de fonctionnement (cf. *isActive*() de *AutomateFini*),
- un état suivant existe pour l'entrée considérée (cf. *getSuivant*() de Etat),

Si l'un de ces pré-requis n'est pas satisfait, alors on lèvera une exception *AFException* avec un message d'explication adéquat.

Question 7.

Écrire une classe *MachineCafe* qui simule une machine à café représentée par l'automate suivant (cet automatique est loin d'être parfait, mais suffisant pour notre cas) :



4. Gestion de la billetterie d'un parc d'attraction

On souhaite développer une application de gestion de la billetterie d'un parc d'attraction. Les billets sont vendus selon les catégories de tarifs suivantes :

1. tarif normal individuel journalier avec accès illimité à toutes les attractions du parc ;
2. tarif normal individuel journalier avec accès illimité à 5 attractions seulement du parc ;
3. tarif réduit (familles nombreuses, étudiants,...) avec accès illimité à toutes les attractions, correspondant à 70% du premier tarif normal ;
4. tarif réduit avec accès illimité à 5 attractions du parc, correspondant à 70% du deuxième tarif normal.

4.1. Classe ParcAttraction

La classe devant servir à gérer la billetterie s'appellera `ParcAttraction`. Elle est caractérisée par :

— son nom, contenu dans une chaîne de caractères,

— sa capacité d'accueil, donnée par la valeur d'un entier,

— les prix unitaires des billets à tarif normal (réels),

— les nombres de billets vendus de chaque catégorie de tarif (entiers).

Le nom, la capacité d'accueil ainsi que les prix unitaires des billets sont fixés à la création d'un nouvel objet de type `ParcAttraction`. Il n'est plus possible de les modifier après la création de l'objet.

La classe `ParcAttraction` expose les opérations suivantes :

▶ `int nbPlacesDispo()` - renvoie la capacité d'accueil restante du parc.

▶ `void vendreBillets(int nb, Tarif tarif)` - permet de vendre des billets, où `nb` exprime la demande en nombre de places et `tarif` le type de tarif demandé. Le type `Tarif` de ce paramètre est une énumération. La vente n'est pas effectuée si la capacité d'accueil restante ne permet pas de satisfaire la demande (exprimée dans `nb`). Un message doit afficher cette impossibilité.

▶ `void reinitialiser()` - remet à zéro tous les compteurs de billets.

▶ `double chiffreAffaires()` - calcule et renvoie le chiffre d'affaire produit par le parc depuis la mise en service de l'objet correspondant ou la dernière réinitialisation.

▶ `double tauxRemplissage()` - renvoie le taux (en pourcentage) de remplissage du parc.

▶ `String toString()` - renvoie une représentation sous forme de chaîne de caractères de l'objet courant de type `ParcAttraction`. Un exemple :

```
Parc MiageWonderland,
Capacité d'accueil : 180,
Tarif normal illimité : 38.50 euros,
Tarif normal illimité 5 : 28.50 euros,
24 billets vendues au tarif illimité,
19 billets vendues au tarif illimité 5,
32 billets vendues au tarif réduit,
23 billets vendues au tarif réduit 5.
```

4.2. Classe Tarif

`Tarif` est une énumération des différentes catégories de tarif : illimité, illimité 5, réduit, réduit 5.

Question 8. Définissez les classes `ParcAttraction` et `Tarif`.

5. Comptes bancaires

Exercice 5.1. L'objectif de cet exercice est de développer une petite application manipulant des comptes bancaires. Définissez l'architecture d'une telle application et développez-la, en utilisant des exceptions pour la gestion des erreurs.

Il existe deux types principaux de comptes :

— `CompteAvecLimite`, qui dispose d'un plafond de dépôt, non modifiable après sa création.

— `CompteSansLimite`, dont le plafond de dépôt n'est pas limité.

Dans la catégorie des comptes avec limite, il existe :

— `CompteEpargne`, pour lequel un minimum pour les opérations de crédit est fixé. Ce minimum est spécifié à la création et non modifiable ensuite. Ce type de compte doit bénéficier d'un versement minimal initial à sa création. Ce minimum est également fixé.

Un compte d'épargne dispose également d'un taux de rémunération variable. Son capital actuel est donc augmenté de l'intérêt acquis à chaque opération de calcul d'intérêt. Pour faire simple, le calcul de l'intérêt se fera selon le moment de l'année où ce calcul est demandé. Par exemple : que vaut le l'intérêt I sur le montant M, selon le taux R à 12 mois ? R est le taux de rémunération variable fixé par la banque. Maintenant, que vaut I sur le même montant M, selon le taux R, sur 3 mois ? 6 mois, 8 mois ? Le nombre de mois dépend du moment dans l'année où le calcul de I est demandé.

Il n'y a pas de découvert autorisé pour ce type de compte. Tout retrait qui rendrait le solde négatif est donc interdit. De plus, tout retrait qui rendrait le solde inférieur au versement minimal initial doit automatiquement entraîner la fermeture du compte.

Dans la catégorie des comptes sans limite, il existe :

— `CompteCourant`, qui dispose d'un découvert autorisé, qui est un plancher négatif. Ce plancher peut être fixé à la création du compte et peut être modifié. Cependant, il ne peut être modifié si le solde actuel du compte est déjà négatif. Si le plancher n'est pas spécifié initialement, alors il vaut zéro. Tout retrait devant faire passer le solde (même négatif) au-dessous de ce plancher doit être refusé. Il n'y a pas de limite sur les opérations de crédit.

— `CompteAction`, pour lequel un plafond pour les opérations de crédit est fixé. Ce plafond est spécifié à la création et non modifiable ensuite. La création d'un compte de ce type doit être également conditionnée à un versement minimal initial, qui est fixé. Il est interdit d'effectuer un débit ponctuel d'un compte de ce type, à moins de lever les actions, et donc de le clôturer définitivement. Par contre, il est autorisé d'effectuer un virement vers un autre compte (épargne ou courant seulement !). Un solde négatif n'est pas autorisé. Tout retrait qui rendrait le solde inférieur au minimum initial doit être refusé.

Chaque compte possède un nom (son détenteur) et un numéro. Tout compte peut être créé avec un montant initial en respectant les contraintes spécifiques. On ne peut créer un compte déjà existant et on ne peut clôturer un compte inexistant.

Votre application `Banque` doit proposer les services suivants à ses utilisateurs :

- Création de compte
- Liste de ses comptes (on peut également particulariser par type de compte) et leurs soldes respectives
- Liste de tous les comptes de la banque
- Crédit d'un compte
- Débit d'un compte
- Virement d'un compte vers un autre
- Calcul d'intérêt d'un (ou tous) compte d'épargne
- Affichage du solde d'un compte
- Clôture d'un compte.

Ces services sont activables via la saisie de l'utilisateur à la console.

Tout utilisateur démarre une session en indiquant son nom à la connexion avec la banque. Tout utilisateur non administrateur ne peut créer des comptes, ni clôturer un compte. Seul l'administrateur peut créer des comptes et les clôturer. Un administrateur peut effectuer toutes les opérations listées ci-dessus, mais il ne ne peut pas posséder de compte. Il peut lister les comptes d'un client dont il fournit le nom (ou le numéro).

6. Gestion des exceptions

Exercice 6.1. Ecrivez une classe `MonException` qui dérive de `Exception`. Ecrivez ensuite une classe `Finally` qui contient une méthode statique `f(int n)` qui ne déclare pas l'exception `MonException` dans sa signature. Elle sera invoquée par la méthode main (qui se trouve dans la même classe) de la manière suivante :

Listing 1 – Gestion de la clause finally

```
1 public static void main(String[] args) {
2  int n;
3  f(1);
4  f(2);
5 }
```

Ces invocations globalement rendront l'affichage suivant :

- dans finally, n = 1
- catch dans f, n = 2
- dans finally, n = 2

Exercice 6.2. Ecrivez une classe `MonException` qui dérive de `Exception`. Ecrivez ensuite une classe `Finally2` qui contient une méthode statique `f(int n)` déclarant l'exception `MonException` dans sa signature. Elle sera invoquée par la méthode main (qui se trouve dans la même classe) de la manière suivante :

Listing 2 – Gestion d'exception

```
1  public static void main(String[] args) {
2   int n = 0 ;
3   try {
4      for (n=1 ; n<5 ; n++) f(n) ;
5    } catch (Except e) {
6      System.out.println ("catch dans main - n = " + n) ;
7     } finally {
8      System.out.println ("dans finally de main - n = " + n) ;
9     }
10 }
```

Ces invocations globalement rendront l'affichage suivant :

— dans finally de f, n = 1
— catch dans f, n = 2
— dans finally de f, n = 2
— catch dans main, n = 2
— dans finally de main, n = 2