TP3 : Accounting

2024-2025

1 Introduction

Dans ce TP, nous allons nous intéresser à la "Comptabilité en partie double". Pour faire court, dans la comptabilité en partie double, chaque opération est associée à une autre opération de la même somme mais de signe opposé. Cela permet un meilleur suivi des opérations.

2 Account

En comptabilité, un **Account** est un enregistrement utilisé pour catégoriser et suivre les transactions financières liées à un aspect spécifique d'une entreprise. Chaque compte représente un élément individuel des activités financières d'une entreprise, tels que les actifs, les passifs, les capitaux propres, les revenus ou les dépenses. Les comptes fournissent un moyen structuré d'organiser les informations financières, permettant aux entreprises de surveiller leur santé financière, de préparer des états financiers et de prendre des décisions éclairées.

Attention, il faut différencier le compte bancaire et le compte en comptabilité. Un compte bancaire est un instrument financier permettant le dépôt, le retrait et la gestion des fonds, tandis qu'un compte en comptabilité est une entité utilisée pour suivre et enregistrer les transactions financières d'une entreprise.

Dans ce TP, nous aurons besoin d'une classe pour représenter les **Account**, avec les informations suivantes :
- **label** : une chaîne de caractères qui permet d'identifier le compte de manière unique
- **balance** : un entier représentant l'accumulation des mouvements (débit et crédit) du compte

2.1 Implémenter la classe **Account**. Il faut prévoir :
- un **constructeur** qui prend en paramètre une chaîne de caractères pour initialiser le champ **label**. Le **constructeur** initialisera aussi le champ **balance** = 0.
- des getters pour chaque champ mais pas de setter.

2.2 Implémenter dans la classe **Account**, les deux méthodes suivantes :

```
1 public void debit(int amount)
```

qui augmente la valeur du champ **balance** de la valeur du paramètre **amount**

```
1 public void credit(int amount)
```

qui diminue la valeur du champ **balance** de la valeur du paramètre **amount**

2.3 Implémenter dans la classe **Account** une méthode **static** qui cherche dans une liste de compte (**List<Account>**) le compte dont le champ **label** est égale à une chaîne de caractères passée en paramètre.
Cette fonction a pour signature :

```
1 public static Account findByLabel(List<Account> accounts, String labelToFind)
```

3 Date

Un autre aspect important en comptabilité, est la notion de date pour analyser l'évolution des **Account** à travers le temps.

3.1 Implémenter la classe **Date**. Cette classe a trois entiers pour champ : **day**, **month**, and **year**. Dans cette classe **Date**, ajouter :
- un **constructeur** qui prend en paramètre trois entiers pour initialiser chaque champ **day**, **month**, et **year**.
- des getters pour chaque champ mais pas de setter.
- un **constructeur** qui prend en paramètre une chaîne de caractères, qui la découpe avec le caractères "/" et qui initialise chaque champ **day**, **month**, et **year** avec les valeurs. Basiquement, on veut un constructeur qui prend en paramètre une chaîne de caractères comme "01/01/2024" et qui créer un objet **Date**.
- une methode toString() qui renvoie chaîne de caractères représentant la date dans le format suivant : "JJ/MM/AAAA".

Pour des questions de simplicité, les années ont 12 mois et tous les mois ont 30 jours.

3.2 On veut pouvoir comparer deux dates. Pour cela :
- Implémenter l'**interface Comparable** dans votre classe **Date**.
- Implémenter la méthode **public int compareTo(Date that)** qui retourne 0 si l'argument **Date that** est égale a la **Date** courante ; une valeur négative (< 0) si la **Date** courante est avant l'argument ; et une valeur positive (> 0) si la **Date** courante est après l'argument. Pour vous aider, vous pouvez comparer les champs un par un en commençant par l'année. La logique est : si les deux champs **year** sont différents, alors je renvoie la valeur de la comparaison de ces champs, i.e. **Integer.compare(this.year, that.year)**, sinon, je procède de la même manière pour les mois, puis pour les jours.

4 Transaction

En comptabilité, une **Transaction** fait référence à un échange de ressources entre deux parties. Cela peut inclure l'achat ou la vente de biens ou de services, le paiement de dettes, les investissements, les emprunts, les paiements de salaires, etc.

4.1 Implémenter la classe **Transaction** qui a les champs suivants :
- **id** : un **UUID** (importez la classe depuis le **package java.util.UUID** qui permet d'identifier la transaction de manière unique.
- **date** : une **Date** que vous avez précédemment implémentée.
- **description** : une chaîne de caractères qui décrit la transaction.
- **debitAccountLabel** : une chaîne de caractères qui identifie le compte de débit.
- **creditAccountLabel** : une chaîne de caractères qui identifie le compte de crédit.
- **amount** : un entier représentant le montant de la transaction.

4.2 Ajouter à votre classe **Transaction** :
- un **constructeur** qui prend des paramètres correspondant à chaque champ pour les initialiser. **ATTENTION**, concernant l'**id** et la **date**, le constructeur prendra en paramètres des chaînes de caractères (**String**) et initialisera les champs comme ceci :
  - Utilisera le constructeur de la classe **Date** qui prend une chaîne de caractères en paramètre.
  - Utilisera la fonction static **fromString** de la classe UUID comme suivant : **UUID.fromString(myUUIDAsString)**
- des **getters** pour chaque champ.
- une **méthode check** qui prend en paramètre une liste de comptes (**List<Account> accounts**) et qui retourne :
  - **true** dans le cas ou la liste contient bien les comptes désignés par les champs **debitAccountLabel** et **creditAccountLabel** de la **Transaction** courante.
  - **false** dans le cas ou la liste ne contient ni l'un, ni l'autre, ni les deux comptes désignés par les champs **debitAccountLabel** et **creditAccountLabel** de la **Transaction** courante.
- une **méthode execute** qui prend en paramètre une liste de comptes (**List<Account> accounts**) et qui :
  - invoque la **méthode check** implémentée ci-dessus.
  - renvoie **false** si la **méthode check** elle-même retourne **false**.
  - invoque la **méthode debit** sur le compte désigné par le champ **debitAccountLabel** avec en paramètre le champ **amount**.
  - invoque la **méthode credit** sur le compte désigné par le champ **creditAccountLabel** avec en paramètre le champ **amount**.

5 Period

En comptabilité, une période fait référence à une période de temps spécifique utilisée pour rapporter les informations financières d'une entreprise. Cette période peut être mensuelle, trimestrielle, annuelle, etc. La période comptable est généralement utilisée pour enregistrer les transactions, produire des états financiers et évaluer la performance financière de l'entreprise sur une base régulière. Elle permet également de comparer les performances financières d'une période à l'autre.
On la classe **Period** avec deux champs de type **Date**, nommés **startDate** et **endDate**.

5.1 Implémenter la classe **Period**. Ajoutez à votre nouvelle classe :
- un **constructeur** qui prend en paramètre deux **Date**, un pour initialiser chacun des champs **startDate** et **endDate**.
- des getters pour chaque champ mais pas de setter.

5.2 Implémenter la méthode **toString** qui renvoie la période sous le format suivant :

```
JJ/MM/AAAA - JJ/MM/AAAA
```

Par exemple :

```
Period period = new Period(new Date(12, 03, 2024), new Date(26, 05, 2024));
System.out.println(period);
//12/3/2024-26/5/2024
```

5.3 Implémenter la fonction **static** suivante :

```
public static Period fromTransactions(List<Transaction> transactions)
```

Qui instancie un nouvel objet **Period** dont la **startDate** est la **Date** de la **Transaction** la plus tôt et **endDate** est la **Date** de la **Transaction** la plus tard.

6 Accounting

Arrivés a ce stade, vous devriez avoir plusieurs classes plus ou moins liées. Nous allons maintenant implémenter la classe qui va travailler avec toutes ces classes pour réaliser une comptabilité : la classe **Accounting**.

6.1 Implémenter la classe **Accounting**. Cette classe a deux champs :
- **accounts** une liste de comptes (**List<Account>**).
- **transactions** une liste de transactions (**List<Transaction>**).

6.2 Ajouter à cette classe :
- un **constructeur** qui prend en paramètre une liste de comptes (**List<Account>**) et une liste de transactions (**List<Transaction>**), et qui initialise les deux champ **accounts** et **transactions**.
- des **getters** pour chaque champ **accounts** et **transactions**.

6.3 Ajouter à cette classe une méthode dont la signature est :

```
public boolean addTransaction(Transaction transaction)
```

et qui :
- invoque la méthode **check** du paramètre **transaction**.
- retourne **false** si le retour de la méthode **check** est **false**.
- ajoute le paramètre **transaction** au champ **transaction** sinon, i.e. si la méthode **check** a retourné **true**.
- retourne elle-même **true** dans ce cas.

6.4 Avant de passer au gros morceau, nous allons ajouter une méthode **copy** à la classe **Account**. Cette méthode a pour signature :

```
1 public Account copy() {
```

et pour implémentions :
- instancie un nouvel objet de type **Account**, la copie, en utilisant le champ **label** du **Account** courant comme paramètre de construction.
- affecte au champ **balance** de la copie la valeur du champ **balance** du **Account** courant.
- retourne la copie.

RAPPEL : normalement, nous n'avons pas de setter pour le champ **balance** de la classe **Account**. C'est pour cela que nous implémentons cette méthode **copy** pour nous permettre d'obtenir une copie exacte. Nous ne souhaitons pas un setter pour ce champ car on souhaite modifier le champ **balance** qu'a travers les méthode métier **debit** et **credit**. La copie nous est nécessaire car nous avons besoin d'avoir différentes "versions" du même compte au cours des périodes.

6.5 Implémenter dans la classe **Date** la méthode suivante :

```
1 public Period toMonthPeriod()
```

qui va instancier et retourner un nouvel objet **Period** tel que :
- son champ **startDate** est le premier jour (1) du mois de la **Date** courante.
- son champ **endDate** est le dernier jour (30) du mois de la **Date** courante.

Nous allons maintenant implémenter la tenue de comptes (bookkeeping).

6.6 Ajouter un nouveau champ **journal** à votre classe **Accounting**, qui associe le label d'un compte, correspondant au champ **label** de la classe **Account**, à une seconde association entre une chaîne de caractères et un compte, i.e. **Account**, ainsi qu'un **getter** pour le champ **journal**.

6.7 Ajouter à votre classe **Accounting** la méthode suivante :

```
1 public void bookkeep()
```

Qui va :
- instancier le champ **journal**.
- pour chaque **Account** du champ **accounts**, ajouter une association au **journal** entre le **label** de l'**Account** et une nouvelle **Map**.
- trier les **Transactions** du champ **transactions** selon leur date.
- pour chaque **Transaction** du champ **transactions**, invoque la méthode **execute**
- transforme la date de la **transaction** en une periode mensuelle en invoquant la methode **toMonthPeriod**.
- pour chaque **Account** du champ **accounts**, ajouter une association entre la version textuelle de la **periode** de la **transaction** et une copie de l'account.

Une fois la methode **bookkeep** exécutée, votre champ **journal** contiendra pour chaque **Account** du champ **accounts** ses différentes valeurs, à travers des copies, au cours des mois.

6.8 Nous allons avoir besoin d'une nouvelle méthode dans la classe **Period**. Sa signature est :

```
1 public List<Period> asMonths()
```

Cette méthode a pour but de transformer la **Period** en une liste de périodes (List<Period>) qui sont toutes des mois, donc du 01/MM/YYYY au 30/MM/YYYY. Par exemple, on applique cette méthode sur la période suivante :

```java
Period period = new Period(new Date(12, 03, 2024), new Date(26, 05, 2024));
List<Period> months = period.asMonths();
System.out.println(months);
// [1/3/2024-30/3/2024, 1/4/2024-30/4/2024, 1/5/2024-30/5/2024]
```

Implémenter la méthode **asMonths()** dans la classe **Period**.

7 Import

En annexe, vous trouverez le code d'une classe qui vous permet d'importer un **Accounting** à partir d'un fichier texte.
Copier-coller ce code dans votre projet. Pour l'utiliser, il faut spécifier le chemin vers un fichier avec le format correct. Comme par exemple :

```java
final Accounting accounting = Import.parse("src/fr/esir/prog1/tp3/accounting/
    dunder_mifflin.txt");
```

8 Report

Dans cette nouvelle section, nous allons attaquer le **Report**. Les comptables éditent des documents dans une forme particulière afin de rendre compte sur l'état financier d'une entreprise. Cependant, nous en ferons une version allégée, mais qui n'en reste pas moins pertinente.

8.1 Pour commencer, nous allons ajouter la méthode suivante à notre classe **Account** :

```java
public void mergeWith(Account account)
```

qui va ajouter la valeur du champ **balance** du **Account** passé en paramètre au champ **balance** du **Account** courant si le champ **label** de l'**Account** passé en paramètre est égale au champ **label** de l'**Account** courant.

8.2 Nous allons ajouter la méthode suivante à notre classe **Accounting** :

```java
public Account getAccountForPeriod(String accountLabel, Period basePeriod)
```

Cette méthode va :
- instancier un nouvel objet **Account** en utilisant le **accountLabel**
- transformer en mois avec la méthode **asMonths** la **Period basePeriod** passée en paramètre
- pour chaque mois, récupérer l'**Account** dans le champ **journal** grâce au paramètre **accountLabel** et le mois modélisé comme une **Period** (on utilisera la fonction toString()).
- accumuler les valeurs en appelant **mergeWith** sur l'**Account** instancié au début et en utilisant l'**Account** récupéré juste avant comme paramètre.

8.3 Implémenter la classe **Report** qui a deux champs :
- **columnHeaders** qui est une liste de chaînes de caractères (**List<String>**)
- **rows** qui est une liste de de listes chaînes de caractères (**List<List<String>**)

Ajouter un **constructeur** qui prend en paramètre un objet **Accounting** et qui va :
- créer une nouvelle période globale grâce aux transactions du paramètre **Accounting** et la fonction **fromTransactions** précédemment implémentée
- découper la période globale en mois avec la méthode **asMonths**
- ajouter chaque représentation textuelle de mois au champ **columnHeaders**
- boucler sur les comptes de l'objet **Accounting**
- instancier une nouvelle liste de chaînes de caractères (**List<String>**)
- ajouter comme premier élément à cette liste le champ **label** du **Account** courant de la boucle
- boucler sur les périodes mensuelles

- récupérer la copie de l'**Account** du **journal** de **Accounting** avec la méthode **getAccountForPeriod** en utilisant comme paramètre les variables de type **Account** et **Period** des boucles
- ajouter la **balance** de la copie de l'**Account** pour la **Period** actuelle à la liste de chaînes de caractères précédemment créée.
- une fois sortie de la boucle des **Period**, ajouter la liste de chaînes de caractères au champ **row** du **Report** courant.

8.4 Les comptables aiment beaucoup les tableaux, notamment Excel. On va donc maintenant implémenter une méthode **toString** qui renvoie une chaînes de caractères représentant notre **Report** comme une table.

9 Pour aller plus loin

Dans cette partie, on vous propose de penser par vous-même à une solution polymorphique. A contrario du reste du tp, cette partie est beaucoup moins guidée. Ci-dessous, vous trouverez des indices qui vont vous apporter au fur et a mesure des éléments pour vous permettre de trouver une solution. N'hésitez pas à vous arrêter et à prendre le temps de réfléchir.

Les comptables aiment aussi beaucoup avoir différents types de périodes. Notre **Report** est basé sur des mois, i.e. chaque colonne est une période d'un mois. Mais parfois, on a besoin de grouper les mois par deux (bimestriel), ou par trois (trimestriel) ou par année.

Indice 1 Réfléchissez à un moyen "propre" pour permettre à l'utilisateur de la classe **Report** de spécifier les colonnes du **Report**.

Indice 2 En regardant le code de la classe **Report**, on peut s'apercevoir qu'on a besoin de différentes variations du découpage en mois, à savoir cette étape du **constructeur** décrite ici :
- découper la période globale en mois avec la méthode **asMonths**

Indice 3 Une solution est d'utiliser ici une classe externe, à qui on va déléguer la logique de découpage de la période globale. On veut donc que quelque ce soit cette classe, elle possède la méthode suivante :

```
1 public List<Period> split(Period period);
```

Indice 4 On crée une interface avec la méthode ci-dessus et on aura autant de sous-classes que de façons de grouper les mois.

Indice 5 On ajoute au **constructeur** du **Report** un paramètre qui a pour type cette interface et appelle la méthode **split()** en lieu et place de l'étape :
- découper la période globale en mois avec la méthode **asMonths**

Indice 6 On cherche ici à mettre en place le patron de conception [1] "Strategy" [2].

"Stratégie est un patron de conception comportemental qui permet de définir une famille d'algorithmes, de les mettre dans des classes séparées et de rendre leurs objets interchangeables."

1. https://refactoring.guru/fr/design-patterns
2. https://refactoring.guru/fr/design-patterns/strategy/java/example

```java
package fr.esir.prog1.tp2.accounting;

import fr.esir.prog1.tp2.accounting.accounts.Account;
import fr.esir.prog1.tp2.accounting.transactions.Transaction;

import java.io.BufferedReader;
import java.io.FileReader;
import java.util.ArrayList;
import java.util.HashSet;
import java.util.List;
import java.util.Set;

public class Import {

    public static Accounting parse(String pathFileName) throws Exception {
        final List<Transaction> transactions = new ArrayList<>();
        final Set<String> accountLabels = new HashSet<>();
        try (BufferedReader bufferedReader = new BufferedReader(new FileReader(
            pathFileName))) {
            bufferedReader.lines()
                    .forEach(line -> {
                        final String[] fields = line.split(", ");
                        transactions.add(
                                new Transaction(
                                        fields[0],
                                        fields[1],
                                        fields[2],
                                        fields[3],
                                        fields[4],
                                        Integer.parseInt(fields[5])
                                )
                        );
                        accountLabels.add(fields[3]);
                        accountLabels.add(fields[4]);
                    });
        }

        return new Accounting(
                accountLabels.stream().map(Account::new).toList(),
                transactions
        );
    }
}
```