1 Introduction

Un fichier *CSV* (*Comma-Separated Values*) est un format de fichier texte simple permettant de représenter des données tabulaires. Chaque ligne correspond à un enregistrement, et les champs sont séparés par des virgules (*comma*) ou parfois d'autres délimiteurs. Ce format est couramment utilisé pour échanger des données entre systèmes, comme des feuilles de calcul ou des bases de données.

Dans ce sujet, nous allons programmer une application pour analyser les performances d'un portefeuille constitué de plusieurs types d'actifs : actions, or, liquidités et obligations. Notre programme calculera le rendement global du portefeuille sur une période donnée. Les données nécessaires seront réparties dans plusieurs fichiers CSV qui sont fournis.

Le sujet est structuré en plusieurs questions indépendantes. Chaque étape peut être réalisée progressivement, sans obligation de lire la suite immédiatement (bien qu'une vue d'ensemble puisse s'avérer utile).

Ce projet nous permettra de travailler notamment sur :

— La manipulation des `Streams` en Java pour traiter efficacement les données provenant des fichiers.

— La gestion des *exceptions* afin de programmer un traitement robuste face à des erreurs potentielles (fichiers manquants, mal formatés, etc.).

2 Lecture d'une donnée élémentaire

Pour manipuler les données élémentaires d'un fichier CSV, nous utilisons une interface générique appelée `Data<T>`. Cette interface permet de lire une donnée depuis une chaîne de caractères et inversement de formater cette donnée en une chaîne de caractères. À chaque type de donnée `T` correspondra une instance de type `Data<T>`.

La version initiale de l'interface est la suivante :

```
public interface Data<T> {
    T read(String text);
    String format(T data);
}
```

Elle comprend donc deux méthodes :

— `read(String text)` : convertit une chaîne de caractères en une donnée du type générique `T`.

— `format(T data)` : convertit une donnée du type générique `T` en une chaîne de caractères.

Nous allons progressivement enrichir cette interface et ses implémentations au fil des questions.

Nous commençons par la lecture des entiers. La méthode `Integer.parseInt` permet de lire l'entier décrit par une chaîne de caractères de chiffres décimaux.

Compléter l'implémentation de l'interface `Data<Integer>` pour manipuler des entiers en écrivant une classe (`IntData`).

Vérifier l'implémentation à l'aide des tests unitaires fournis.

Décommentez les tests actuellement commentés dans le fichier `IntDataTest.java`.

Ces tests vont nous permettre d'affiner la gestion des exceptions pour les instances de `Data<T>`. Nous introduisons une exception `DataMismatchException` qui doit être levée lorsqu'une chaîne de caractères n'est pas correctement lue.

Créer la classe `DataMismatchException`.

Elle doit contenir une méthode `toString` permettant d'afficher des messages d'erreurs tels que ceux apparaissant dans les tests unitaires. Elle contiendra donc deux propriétés, l'une décrivant le type attendu, et l'autre contenant la chaîne de caractères qui n'a pas pu être lue. Enfin, en tant qu'exception, elle doit étendre la classe `Exception`.

Ajouter à la déclaration de la méthode `read` de l'interface `Data`, que la méthode peut lever une exception `DataMismatchException`.

Modifier `IntData`. Ajouter la déclaration de levée d'exception. De plus, si la donnée n'est pas lisible en tant que `int`, `Integer.parseInt` lève une exception de type `NumberFormatException`. Rattraper cette exception avant d'émettre une exception `DataMismatchException`. Ne pas oublier de déclarer la cause de la nouvelle exception.

Procéder ensuite de même pour définir une classe `DoubleData` implémentant `Data<Double>`.

Utiliser `Double.parseData` (qui peut lever `NumberFormatException`) et `Double.toString`.

Procéder de même pour définir une classe `DateData` implémentant `Data<LocalData>`.

La classe `java.time.LocalData` permet de représenter des dates (des jours du calendrier), il est conseillé de lire sa documentation. La méthode statique `LocalDate.format` permet de lire une date depuis une chaîne de caractères, selon un format qui doit être spécifier par le deuxième argument. Nous utiliserons le format `DateTimeFormatter.ISO_LOCAL_DATE` (de la forme `"2024-11-20"`). La méthode `date.format` permet d'écrire la date et prend aussi un format en argument (nous utiliserons le même). La lecture d'une date peut échouer si le format n'est pas respecté, en levant l'exception `DateTimeParseException`.

Pour terminer, nous fournissons `StringData` implémentant `Data<String>` qui lit les chaînes de caractères entre guillemets. Décommentez simplement les lignes nécessaires.

Ajouter des constantes `INT`, `DOUBLE`, `STRING` et `DATE` dans l'interface `Data<T>`, chacune contenant une instance des classes que nous avons définis, selon leur type.

3 Lecture d'une ligne

Après avoir traité la lecture de données élémentaires, nous passons à une tâche plus complexe : lire une ligne complète d'un fichier CSV et la transformer en un objet structuré. Nous proposons ici une API conçue pour être intuitive et extensible.

Prenons l'exemple d'un fichier contenant des informations sur des personnes : prénom, nom, âge, et date de naissance. Voici un exemple d'un tel fichier :

```
John,Doe,30,1994-05-15
Jane,Smith,28,1996-11-22
Alice,Johnson,35,1989-02-10
Bob,White,42,1982-07-07
Charlie,Brown,25,1999-08-30
```

Chaque ligne du fichier pourrait être représentée par l'enregistrement suivant, muni d'une méthode statique capable de lire une ligne du fichier :

```
record People(String firstName, String lastName, int age, LocalDate birthday) {
    public static People ofLine(String line) throws LineReaderException {
        LineReader<People> reader =
            LineDescription.<People>line()
                .with(DATE)
                .with(INT)
                .with(STRING)
```

```
                .with(STRING)
                .using(fn -> ln -> age -> bd -> new People(fn, ln, age, bd));
        return reader.read(line);
    }
}
```

Expliquons.

— L'instance de type `LineReader<T>` est capable de lire des lignes (type `String`), en les interprétant pour obtenir une donnée de type `T`.

— La méthode statique `line()` initialise une description des colonnes d'un fichier CSV, pour l'instant sans colonne.

— Les appels en chaîne à `with` permettent d'ajouter des colonnes, **de droite à gauche**, en précisant le type des données via une instance de `Data<T>`. Ici nous utilisons les constantes de l'interface `Data`, il faudra donc les importer avec l'import suivant :

```
import static fr.univamu.csvparser.data.Data.*;
```

Dans cet exemple, les colonnes sont ajoutées dans l'ordre inverse : d'abord la date, puis l'âge, puis les noms et prénoms.

— La méthode `using` prend une lambda imbriquée qui associe les colonnes à leur rôle. Elle construit l'objet cible en assemblant les données lues. Notons qu'il s'agit une fonction currifiée à 4 arguments [1], nous ne pouvons pas donner directement une référence au constructeur comme par exemple `People::new`. Le nombre d'arguments correspondra aux nombres de colonnes lues.

— Enfin, la méthode `read(String line)` utilise cette instance de type `LineReader<T>` pour analyser une ligne de texte et produire un objet structuré du type `T`, avec la possibilité de lever une exception de la classe `LineReaderException`.

L'exemple ci-dessus montre comment cette API permet de lire une ligne et de produire un objet `People`. Grâce à la généralité de `LineDescription<T>`, ce mécanisme s'applique à tout type de fichiers CSV, à condition de fournir une description avec les bonnes colonnes. Il s'articule autour de deux interfaces.

— `LineReader<T>` représente des instances capables de lire une ligne et de produire une valeur de type `T` ;

— `LineDescription<S,R>` est une description partielle de la liste des colonnes d'un fichier CSV.

Nous commençons par la deuxième de ces interfaces, qui est paramétrée par deux types `S` et `R`.

— `R` représente le type final des objets que l'on souhaite obtenir. Par exemple, une instance de `LineDescription<S, People>` décrit comment construire un objet de type `People`.

— `S` représente l'information qui, après avoir lue les premières colonnes, nous manque pour pouvoir terminer la création de la valeur de type `R`. En pratique, nous utiliserons pour `S` le type d'une fonction produisant une valeur de type `R`, en fonction des valeurs rencontrées dans les colonnes précédentes [2].

Prenons l'exemple de la classe `People`. Pour construire un objet `People` à partir des colonnes d'une ligne CSV, nous décrivons chaque étape intermédiaire avec le type `S`. Voici comment cela fonctionne :

— Initialement, nous ne savons rien sur les données, nous avons besoin de toute l'information pour construire l'instance de `People`, donc `S` est `People`. Cela correspond au type de la méthode `line()`, qui retourne un `LineDescriptor` avec `S = T`.

1. *cf.* le cours de structures discrètes.

2. Une valeur de type `Function<A,B>` représente la capacité de produire l'*information* `B` à partir de l'*information* `A`, c'est donc intuitivement l'*information* `B - A`, ou dit autrement l'information qui nous manque si on connaît `A` pour obtenir `B`. À l'extrême, `Consumer<A>` représente la négation de `A`.

```
LineDescription<People, People> desc = LineDescription.<People>ends();
```

— Ensuite, nous ajoutons une première colonne pour lire la date d'anniversaire (`LocalDate`). À ce stade de lecture de la ligne, la date est connue, et nous manquons des autres information pour construire l'instance de `People`. Cette instance pourrait être construite si nous disposions d'une fonction de la date connue vers `People`. Ainsi `S = Function<LocalDate,People>` convient :

```
LineDescription<Function<LocalDate, People>, People> desc =
  LineDescription.<People>line().with(Data.DATE);
```

— Ainsi de suite, chaque colonne correspondra en l'ajout d'un argument à une fonction produisant une instance de `People` :

```
LineDescription<Function<Integer, Function<LocalDate, People>>, People> desc =
LineDescription
  .<People>line()
  .with(Data.DATE)
  .with(Data.INT);
```

— Jusqu'à `using` dont le rôle est de fournir cette information manquante, du type `S`.

```
LineDescription<People, People> desc =
LineDescription.<People>line()
  .with(Data.DATE).with(Data.INT).with(Data.STRING).with(Data.STRING)
  .using(fn -> ln -> age -> bd -> new People(fn, ln, age, bd));
```

Plus concrètement, le type `LineDescriptor<S,R>` est l'interface suivante (à laquelle nous ajouterons ensuite la gestion des exceptions) :

```
public interface LineDescription<S,R> {
  R read(S s, Iterator<String> strings);
}
```

La méthode `read` est celle qui lira effectivement les données d'une ligne d'un fichier CSV. Le paramètre `s` contient donc "l'information manquante" de façon un peu mystérieuse. Le paramètre `strings` est un itérateur sur les chaînes de caractères correspondant aux portions de la ligne séparées par les virgules (lues de gauche à droite). La lecture produit comme promis une valeur de type `R`.

La méthode `read` produite par la méthode statique `ends` ne lit aucune colonne, et se contente de retourner l'information manquante qui lui est fournie.

Implémenter la méthode statique `line()` dans l'interface `LineDescription` à l'aide d'une lambda.

La méthode `read` produite par la méthode par défaut `with`

1. lit une colonne, prise dans l'itérateur ;
2. l'interprète avec l'instance de `Data<T>` ;
3. puis l'intègre dans l'information fournie, qui doit être une fonction acceptant la valeur interprétée ;
4. et enfin continue la lecture des colonnes en utilisant `this.read`.

L'interprétation pouvant lever une exception, il nous faut d'abord déclarer que `read` peut propager l'exception.

Modifier la déclaration de la méthode `read` dans l'interface `LineDescription`, afin de permettre la propagation de l'exception `DataMismatchException` levée par `Data.read`.

Implémenter à l'aide d'une lambda une première version de la méthode `with(Data<T> dataT)` sans gestion d'exception supplémentaire, en suivant les 4 étapes décrites.

La méthode `read` produite par la méthode par défaut `ignore` permet de sauter la colonne. Ainsi, on enlève simplement la donnée de l'itérateur, et on continue la lecture avec la même information. Autrement dit, il suffit de sauter les étapes 2 et 3.

Implémenter à l'aide d'une lambda une première version de la méthode `ignore()` sans gestion d'exception supplémentaire.

Surcharger la méthode `ignore()`, en créant une autre version prenant un entier en argument, correspondant au nombre de colonnes successives à ignorer.

Il nous reste la méthode `using`. Elle va nécessiter une classe implémentant l'interface `LineReader<T>`. Cette interface propose simplement une méthode `read`, prenant une ligne sous forme d'une `String`, et retourne la valeur lue de type <T>.

```
public interface LineReader<R> {
  R read(String line);
}
```

Comme pour l'interface `LineDescription`, la méthode `read` de l'interface `LineReader` peut propager des exceptions de type `DataMismatchException`.

Ajouter une déclaration d'exception à la méthode `read` de `LineReader`, et dans son implémentation `CsvLineReader`.

Il nous reste à implémenter la méthode `read` de `CsvLineReader`. L'essentiel de son travail va être accompli par l'autre méthode `read`, celle de l'instance de `LineDescription<S,R>`, qui prend en premier argument l'information de type `S`, qui est reçue par le constructeur (c'est la fonction passée en argument à la méthode `using`), et en deuxième argument un itérateur des sections de la ligne séparées par les virgules. Pour obtenir cet itérateur, nous utilisons la classe `Scanner`, avec ses méthodes `useDelimiter` pour spécifier le symbole du délimiteur, pour nous la virgule `","`, et `tokens` qui retourne les sections ainsi séparées sous forme d'un `Stream`. L'itérateur est produit par la méthode `iterator` des streams.

Compléter la méthode `read` de la classe `CsvLineReader`, qui obtient l'itérateur et le passe à `lineDescription.read`.

Utiliser les tests unitaires de la classe `CsvLineReaderTest` pour vérifier votre programme.

4 Lecture d'un fichier CSV

Nous souhaitons maintenant lire tout un fichier CSV. Pour cela, nous allons créer une classe `Report` contenant le rapport de lecture du fichier comprenant :

— la liste des valeurs lues,

— la liste des exceptions levées.

Par ailleurs, nous voudrons utiliser `Report` au sein d'un collecteur de streams. Nous voulons donc deux méthodes : l'une qui permet la lecture d'une ligne supplémentaire, l'autre pour fusionner deux rapports en un seul.

Ajouter des propriétés dans la classe `Report`, pour pouvoir stocker la liste des lignes lues (du type `R`), la liste des exceptions levées lors des lectures des lignes, et le nombre de lignes lues. Toutes ces propriétés doivent être correctement initialisées (sans passer par un constructeur).

Rendre le rapport itérable, en implémentant `Iterable<T>` (il suffit de retourner l'itérateur de la liste des valeurs lues).

Ajouter un accesseur sur la liste des lignes lues, et un sur la liste des exceptions levées.

Le rapport est actif : il supporte la lecture de lignes CSV additionnelles. Il posséde donc une propriété `LineReader<T>` permettant de lire les lignes.

Compléter la méthode `read`, qui reçoit une ligne, la lit, et enregistre la donnée lue. Si une exception est levée, elle enregistre l'exception mais pas de donnée.

Compléter la méthode `merge`, qui reçoit un deuxième rapport, et ajoute toutes les données et exceptions de ce deuxième rapport à `this`, ainsi que le nombre de lignes lues.

Vérifier que les tests de la classe `Report` passent correctement.

Compléter la méthode statique `fromFile` permettant la lecture d'un fichier. Son premier argument est le chemin du fichier, de type `Path`, le deuxième est le `LineReader`, et le troisième est le nombre de ligne à sauter en début de fichier (en général 1 car la première ligne contient souvent une description des colonnes. Surcharger cette méthode pour que le nombre de lignes à sauter soit 0 par défaut.

Pour ouvrir le fichier, nous utilisons la méthodes `Files::lines` qui crée une `Stream<String>` avec la liste des lignes. Ensuite il suffit de collecter en utilisant la classe `Report`.

5 Gestion améliorée des erreurs

Lorsque le fichier CSV n'est pas conforme aux attentes, une exception est levée. Cependant, actuellement les exceptions en renseignent ni la nature du problème rencontré, ni sa localisation. Nous voulons pouvoir, suite à la lecture du fichier :

- connaître le numéro de ligne de chaque erreur recensée ;
- savoir pour chaque erreur, s'il s'agit d'une donnée mal-formattée ou d'une donnée manquante ;
- repérer aussi les lignes trop longues dans le fichier CSV ;
- les problèmes d'ouvertures de fichier (inexistant ou impossible à lire) devront rester des `IOException`, mais les problèmes de formattage du fichier devront correspondre à un autre type, `LineReaderException`.

Créer trois extensions `MismatchDataException`, `TooMuchDataException`, `NotEnoughDataException` de la classe `LineReaderException`

- `MismatchDataException` a pour propriété une `DataMismatchException` ;
- `TooMuchDataException` contient la liste des termes non-lus sur la ligne ;
- `NotEnoughDataException` n'a pas d'information supplémentaire.

Pour chacune des trois extensions, implémenter une méthode `toString`, qui affiche le numéro de ligne suivi d'un message adapté à chaque cas.

Modifier les déclaration des méthodes `read` des interfaces `LineDescription` et `LineReader`, pour que les exceptions émises soient les extensions de `LineReaderException`.

Modifier en cohérence les implémentations de ces méthodes.

- dans l'implémentation de `with`, rattraper l'exception `DataMismatchException` pour émettre l'exception `MismatchDataException` ;
- dans l'implémentation de `with` et `ignore`, lever l'exception `NotEnoughDataException` s'il n'y a plus de donnée disponible ;
- dans l'implémentation de `CsvLineReader.read`, lever l'exception `TooMuchDataException` si toutes les données n'ont pas été lues.

Modifier la méthode `merge` de la classe `Report`, pour ajuster les numéros de lignes des exceptions du rapport fusionné à `this` (en leur ajoutant le nombre de ligne de `this`).

Écrire un test lisant un petit fichier CSV, contenant des lignes correctes plus quelques lignes erronées de chaque sorte. En déduire une méthode de test, vérifiant que les erreurs sont correctement repérées et catégorisées.

6 Une application

Nous proposons d'analyser le rendement d'un portefeuille financier composés de plusieurs produits, pendant une période de temps historique, en utilisant des valeurs réelles (récupérées sur le site datahub.io).

Les produits qui nous intéressent sont :

- le cash (garder de l'argent sans l'investir) en dollars ;
- l'or ;
- les actions, en l'occurence nous considérerons le SP500, l'équivalent du CAC40 sur la bourse de New-York, qui est un mix d'actions de diverses entreprises ;
- les obligations d'état des États-Unis d'Amérique, qui consistent en prêter de l'argent au gouvernement des É-U en échange d'intérêts fixes, sur 10 ans.

Chacun de ces produits évolue selon ses propres règles.

- Le cash garde toujours la même valeur faciale, mais se déprécie par l'inflation : un dollar de 1980 donne plus de "pouvoir d'achat" qu'un dollar de 2024. Le fichier `inflation.csv` donne les valeurs de l'inflation, avec en première colonne la date (mensuelle) et en deuxième colonne l'indice avec une base 100 en 1983. Ainsi 9.8 dollars en 1913 sont équivalents à 100 dollars en 1983 et à 315 dollars aujourd'hui.
- L'or, dont la valeur est donnée en dollar par once et déterminée par le marché. Le fichier `gold.csv` récapitule ses valeurs depuis 1833 (2^{e} colonne), l'once valant environ 2600 dollars actuellement.
- Les actions, dont la valeur est donnée en dollar par unité et déterminée par le marché. Pour nos calculs, la différence avec l'or est que les actions produisent aussi des dividendes. Ainsi chaque mois, chaque action rapporte une petite somme en cash, déterminée par chaque entreprise. Le fichier `sp500.txt` contient toutes ces données et d'autres, seules les trois premières colonnes nous intéressent, avec la date (mensuelle), la valeur d'une unité de SP500, et la valeur des dividendes versées sur le mois.
- Les obligations sont libellées en dollar, avec un montant fixe remboursée au dix ans. Par exemple, en achetant 100$ d'obligations au 1^{er} janvier 2010, on récupère 100$ de cash le 1^{er} janvier 2020. De plus les obligations donnent droit à des intérêts, aux taux fixées lors de l'achat de l'obligation. Dans cet exemple, le taux est de 3.73% annuel au 1^{er}janvier 2012, ce qui rapporte 0.31$ par mois pendant toute la durée de l'obligation. L'obligation ne peut pas être remboursée avant la fin de sa période de 10 ans, mais il est possible de la revendre à un tiers. Nous utiliserons comme approximation pour le prix de vente d'une obligation (sa valeur) la formule suivante :

$$\text{montant} \times \left(\frac{1200 + \text{taux de l'obligation}}{1200 + \text{taux actuel}} \right)^{\text{nombre de mois restants}}$$

Implémenter une classe `MonthlyStats`, regroupant toutes les statistiques utiles à une date précise. Pour chaque statistique, définir un accesseur et un mutateur.

Ajouter une classe `HistoricStats` dont le rôle sera de contenir toutes les statistiques provenant des fichiers CSV. Pour cela, utiliser un dictionnaire de type `Map<LocalDate,MonthlyStats>`.

Ajouter une méthode `ensureDate(LocalDate date)`, qui vérifie qu'il existe bien une statistique enregistrée pour cette date, sinon, elle ajoute une telle statistique (avec des valeurs à 0). Dans tous les cas elle renvoie la statistique pour cette date.

Pour chaque fichier CSV, compléter la méthode retournant une instance de `LineReader<Boolean>`. À la lecture d'une ligne, les données sont directement ajoutées aux statistiques du mois correspondant.

Compléter le constructeur de `HistoricStats` en ajoutant la lecture des fichiers CSV pour récupérer les statistiques.

Écrire une fonction `main`, chargeant les statistiques. Corriger toutes les erreurs détectées dans les fichiers CSV.

Implémenter la classe des obligations `Bond`, une obligation étant définie par sa date de remboursement, son taux et son montant. Lui ajouter une méthode `estimatedValue` retournant sa valeur en fonction de la date et des taux actuels, une méthode calculant les intérêts mensuels, et une méthode `toString` adaptée.

Implémenter la classe `Portfolio` décrivant un portefeuille composé de cash (en dollars), d'or (en onces), d'une liste d'obligations, et d'actions du SP500 (en unités), avec des accesseurs et une méthode `toString`.

Ajouter dans `MonthlyStats` une méthode `bondsFromCash` retournant une nouvelle obligation pour un montant donné, une méthode `ouncesFromCash` retournant une quantité d'or équivalente à un montant donné, et `sharesFromCash` retournant une quantité d'actions du SP500 équivalente à un montant donné.

Ajouter dans `HistoricStats` une méthode `adjustByInflation` permettant de convertir un montant en dollars entre deux dates, en ajustant selon l'inflation.

Ajouter dans `MonthlyStats` une méthode `evaluateBonds` permettant d'évaluer la valeur d'une obligation, et une méthode `evaluate` utilisant une stream pour évaluer la valeur d'un portefeuille.

Ajouter dans la classe `Portfolio` une méthode statique `evenlySplit` produisant un porte-feuille composé de 25% de cash, 25% d'actions, 25% d'obligations et 25% d'or.

Ajouter dans la classe `Portfolio` une méthode `double earnings(MonthlyStats currentMonth)` calculant les gains effectués sur un mois avec ce portefeuille (dividendes des actions + intérêts des obligations).

Nous souhaitons simuler le scénario d'épargne suivant. Un jeune actif décide d'épargner chaque mois pendant sa carrière, du 1erjanvier 1980 au 1erjanvier 2020, afin de préparer sa retraite. Son portefeuille est équitablement réparti (25% dans chaque type de produits), et doit le rester. Ainsi, chaque 1erdu mois, il rééquilibre son portefeuille (en pratique, nous calculons la valeur de son portefeuille, ajoutons le montant épargné pour le mois, et reconstruisons un portefeuille équilibré). Nous prendrons comme valeur épargné chaque mois 500$ en équivalent 2020 (soit environ 150$ en janvier 1980). Par comparaison, un actif français cotise un peu moins de 1000 euros par mois en 2023 (le montant des pensions est d'environ 330 milliards, pour 30 millions d'actifs).

Programmer ce scénario dans le `main`, pour calculer le montant épargné dont aurait disposer notre jeune retraité en janvier 2020.

7 Tâches optionnelles

Ajouter les méthodes suivantes dans `LineDescription`.

```
default <T> LineDescription<T,R> comap(Function<T,S> transform) {...}
default <T> LineDescription<S,R> consuming(Data<T> dataT, Consumer<T> consumer) {...}
```

La première ne lit pas de colonne, mais modifie l'information courante. La deuxième lit une colonne, et utilise immédiatement la valeur lue, mais ne modifie pas l'information courante.

Proposer des interfaces et méthodes pour gérer l'écriture de données au format CSV.