Calculatrice – version 4[1]

Compétence visée :

- Apprentissage des *streams* Java.

Qu'est-ce que les *streams*[2] Java ?

Jusqu'à présent[3], effectuer des traitements sur des Collections ou des tableaux (de type MapReduce) en Java passait essentiellement par l'utilisation du *design pattern*[4] Iterator. Depuis Java 8, l'API Stream est proposée pour simplifier ces traitements en introduisant un nouvel objet, Stream. Un *stream* se construit à partir d'une source de données (une collection, un tableau ou des sources I/O par exemple) et possède un certain nombre de propriétés spécifiques :

- un *stream* **ne stocke pas de données**, contrairement à une collection. Il se contente de les transférer d'une source vers une suite d'opérations.
- un *stream* **ne modifie pas les données** de la source sur laquelle il est construit. S'il doit modifier des données pour les réutiliser, il va construire un nouveau *stream* à partir du *stream* initial. Ce point est très important pour garder une cohérence lors de la parallélisation du traitement.
- le **chargement** des données pour des opérations sur un *stream* s'effectue de façon ***lazy*** (fainéante). Cela permet d'optimiser les performances des applications. Par exemple, si l'on recherche, dans un *stream* de chaînes de caractères, une chaîne correspondant à un certain *pattern*, cela nous permettra de ne charger que les éléments nécessaires pour trouver une chaîne qui conviendrait et le reste des données n'aura alors pas à être chargé.
- un *stream* **peut ne pas être borné**, contrairement aux collections. Il faudra cependant veiller à ce que nos opérations se terminent en un temps fini – par exemple avec des méthodes comme limit(n) ou findFirst().
- enfin, un *stream* **n'est pas réutilisable**. Une fois qu'il a été parcouru, si l'on veut réutiliser les données de la source sur laquelle il avait été construit, il faut reconstruire un nouveau *stream* sur cette même source.

Il existe deux types d'opérations que l'on peut effectuer sur un *stream* : les opérations intermédiaires et les opérations terminales :

---

[1] Ce sujet a été conçu initialement en collaboration avec Dr. Yoann Maurel, enseignant-chercheur à l'Université de Rennes 1.

[2] Les *streams* sont des flux qu'il ne faut pas confondre avec les InputStream et OutputStream, qui n'ont rien à voir. Les *streams* sont utilisés pour la programmation fonctionnelle dans Java.

[3] Présentation des Streams Java 8 issue du site http://blog.ippon.fr/2014/03/17/api-stream-une-nouvelle-facon-de-gerer-les-collections-en-java-8/.

[4] En français, patron de conception. *Design Patterns: Elements of Reusable Object-Oriented Software*, E. Gamma, R. Helm, R. Johnson, and J. Vlissides. Addison-Wesley Professional, 1ère édition, (1994).

- les **opérations intermédiaires** (Stream.map ou Stream.filter par exemple) sont effectuées de façon *lazy* et renvoient un nouveau *stream*, ce qui crée une succession de *streams* que l'on appelle *stream pipelines*. Tant qu'aucune opération terminale n'aura été appelée sur un *stream pipelines*, les opérations intermédiaires ne seront pas réellement effectuées.
- quand une **opération terminale** sera appelée (Stream.reduce ou Stream.collect par exemple), on va alors traverser tous les *streams* créés par les opérations intermédiaires, appliquer les différentes opérations aux données puis ajouter l'opération terminale. Dès lors, tous les *streams* seront dit consommés, ils seront détruits et ne pourront plus être utilisés.

L'ensemble des méthodes proposées dans l'API Stream est disponible dans la javadoc https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/stream/Stream.html

Exercice 1 – Pour débuter

L'objectif de cet exercice est d'apprendre à manipuler les *streams* Java.

1.1. Copiez votre projet Calculatrice-v3 et renommez-le en Calculatrice-v4.

1.2. La classe Calculatrice propose une méthode `chercherOperation` qui permet de rechercher une opération dans un ensemble d'opérations. Transformez le code de votre méthode afin d'utiliser un *stream*.

```
public Operation chercherOperation(String nom) {
	for(Operation o : operations) {
		if(o.getNom().equals(nom)) {
			return o;
		}
	}
	return null;
}
```

1.3. Afin de pouvoir réaliser d'autres opérations avec des *streams*, il va falloir compléter[5] votre code :

- en créant une énumération contenant trois valeurs possibles : `ONE`, `TWO` et `NO_LIMIT`. Cette énumération, nommée OperandeCardinalite, doit être mise dans le package fr.esisar.calculatrice.operations.

- en ajoutant une méthode `getNbOperandes()` à l'interface Operation. Cette méthode doit être implantée dans les classes abstraites qui implémentent l'interface Operation. Cette méthode retourne un OperandeCardinalite qui vaut ONE pour les opérations unaires, TWO pour les opérations binaires et NOT_LIMIT pour les opérations ensemblistes.

[5] Pensez à utiliser les outils de générations de code d'Eclipse (Source > Override/Implement Methods… et Source > Generate Getters and Setters…)

- en ajoutant, dans la classe `Calculatrice`, la méthode `getOperation()` qui retourne l'ensemble d'opérations de la calculatrice.

1.4. Dans votre classe `Calculateur`, créez une calculatrice ayant les opérations : Ajouter, Soustraire, Multiplier, Diviser, Valeur Absolue et Maximum.

Ecrivez le code qui permet d'afficher la phrase ci-après en utilisant l'API Stream. Grâce à l'API Stream vous devez calculer le nom d'opérations disponibles dans la calculatrice avec l'arité égale à 2 :

```
Calculatrice a 4 operations d'arite 2
```

Ecrivez le code qui permet de réaliser l'ensemble des calculs donnés ci-après grâce à l'API Stream :

```
2.3 + 5.4 = 7.7
2.3 * 5.4 = 12.42
2.3 / 5.4 = 0.4259259259259259
2.3 - 5.4 = -3.1000000000000005
```

Exercice 2 – Pour aller plus loin

Java[6] est un langage orienté objet : à l'exception des instructions et des données primitives, tout le reste est objet, même les tableaux et les chaînes de caractères.

Java ne propose pas la possibilité de définir une fonction/méthode en dehors d'une classe ni de passer une telle fonction en paramètre d'une méthode. Depuis Java 1.1, la solution pour passer des traitements en paramètres d'une méthode est d'utiliser les classes anonymes internes[7].

Pour faciliter, entre autres, cette mise à œuvre, Java 8 propose les expressions lambda. Les expressions lambda sont aussi nommées *closures* ou fonctions anonymes : leur but principal est de permettre de passer en paramètre un ensemble de traitements.

De plus, la programmation fonctionnelle est devenue prédominante dans les langages récents. Dans ce mode de programmation, le résultat de traitements est décrit mais pas la façon dont ces traitements sont réalisés. Ceci permet de réduire la quantité de code à écrire pour obtenir le même résultat.

2.1. Créez un nouveau projet Calculatrice-v5 qui contient un package fr.esisar.calculatrice ayant trois classes :

a. une classe `OperationInvalide` qui étend la classe Exception de Java et qui contient un attribut message (*cf.* version précédente de la calculatrice) ;

b. une classe `Calculatrice` qui contient une interface fonctionnelle `OperationDouble` qui propose une méthode `doCalculer` prenant en paramètre deux doubles et retournant un double avec la possibilité de lever une exception `OperationInvalide`. La classe Calculatrice doit également contenir une méthode `calculer` qui prendra en paramètre une opération de type

[6] Présentation des expressions lambda issue du site : https://www.jmdoudoux.fr/java/dej/chap-lambdas.htm.
[7] http://blog.paumard.org/cours/java/chap04-structure-classe-classe.html

`OperationDouble` ainsi que deux doubles et qui retournera le résultat du calcul sous la forme d'un double.

c. une classe `Calculateur` qui permettra de tester la calculatrice avec la création d'une calculatrice ainsi que de l'ensemble des opérations (addition, soustraction, multiplication et division) de type `OperationDouble`.

2.2. Créez un nouveau projet Calculatrice-v6 qui utilise les interfaces fonctionnelles proposées par le package java.util.function[8]. Vous n'aurez alors besoin que de la classe `Calculateur` pour réaliser votre calculatrice. Est-ce que la division par zéro est bien gérée ?

[8] https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/function/package-summary.html