Sornas BTS SIO 2

Workshop : Calculatrice en Java

Projet de Workshop de la semaine du 23 septembre 2019

1. Obligations

- Lire entièrement le sujet
- Suivre les consignes
- Respecter les règles de rendu

2. Règles de Rendu

Cette partie vous donne les différentes règles à respecter.

Questions	alexandre.th.manuel@gmail.com
Rendu Final	Vendredi 17h

Envoyez-moi un mail si vous rencontrez des difficultés.

2.1 La Triche

Les cas de triche seront sévèrement sanctionnés. Les tricheurs verront leur note multipliée par 0 et l'administration sera prévenue.

2.2 Le Nommage

Vous devrez rendre par mail (à olivier.krakus@sornas.fr) une archive zip.

Votre archive devra à minima contenir le fichier suivant :

```
Main.java
```

Votre mail devra avoir comme objet « [SORNAS] [WORKSHOP_SIO_2] Prénom Nom » et avoir l'archive zip en pièce jointe.

Bon courage à tous !

Palier 1 : opération simple

Dans un premier temps, vous devrez être capable de gérer les cas suivants :

```
>>> 20+3
23
>>> 2 - 1
1
>>> 2 * 2
4
>>> 3 / 2
1.5
```

Il est admis que des espaces peuvent se glisser ou non dans votre opération sans que cela ne pose de problème.

Pour effectuer ce traitement, vous devrez créer une interactivité avec l'utilisateur. On veut que le programme demande un nouveau calcul à effectuer en continu. Vous aurez donc recours à une boucle infinie.

Boucle infinie d'interaction

Votre programme devra avoir la forme suivante :

```java
package com.company;

import java.util.Scanner;

public class Main {

    public static Double Compute(String operation) {
        // Traitement de l'opération...
        return result;
    }

    public static void main(String[] args) {
        String operation;
        Double result;
        Scanner scanner = new Scanner(System.in);

        System.out.println("Bienvenue");
        while (true) {
            operation = scanner.nextLine();
            result = Compute(operation);
            if (result == Math.round(result))
                System.out.println(Math.round(result));
            else
                System.out.println(result);
        }
    }
}
```

Nous allons, tout le long de ce TP, nous concentrer sur le contenu de la fonction compute.

Evaluation de l'opération

On veut ici gérer des opérations sous la forme suivante :

```
OPERATION → NOMBRE OPERATEUR NOMBRE
OPERATEUR → '+'
          | '-'
          | '*'
          | '/'
NOMBRE    → constant
```

Nous allons devoir isoler les trois éléments qui composent notre opération :

- Les deux nombres
- L'opérateur

Commençons par retirer les espaces de notre opération :

```
operation = operation.replaceAll("\\s", "");
```

Il faut déterminer l'opérateur :

```
char operator;
if (operation.contains("+"))
    operator = '+';
else if // ...
// ...
else {
    System.out.println("Opération invalide");
    return 0d;
}
```

Une fois l'opérateur déterminé, on peut récupérer les deux nombres de notre opération :

```
List<Double> numbers = Arrays.stream(operation.split("[+/*-]"))
        .map(Double::parseDouble).collect(Collectors.toList());
```

Puis on applique l'opérateur à nos deux nombres :

```
if (operator == '+')
    return numbers.get(0) + numbers.get(1);
if (operator == '-')
    return numbers.get(0) - numbers.get(1);
// ...
```

Palier 2 : Opération multiple

On veut maintenant, en plus des cas précédents, gérer les cas suivants :

```
>>> 20+3+4
27
>>> 2 – 1 - 1
0
>>> 2 * 2 * 3
12
>>> 6 / 2 / 2
1.5
```

On veut maintenant gérer les opérations suivantes :

```
OPERATION → NOMBRE PLUS OP_PLUS
          | NOMBRE MOINS OP_MOINS
          | NOMBRE MULT OP_MULT
          | NOMBRE DIVISE OP_DIVISE

OP_PLUS   → NOMBRE PLUS OP_PLUS
          | NOMBRE
OP_MOINS  → NOMBRE MOINS OP_MOINS
          | NOMBRE
OP_MULT   → NOMBRE MULT OP_MULT
          | NOMBRE
OP_DIVISE → NOMBRE DIVISE OP_DIVISE
          | NOMBRE

PLUS      → '+'
MOINS     → '-'
MULT      → '*'
DIVISE    → '/'
NOMBRE    → constant
```

Pour ce faire, on garde quasiment exact le code précédent, on se contentera simplement de modifier la partie d'application de l'opérateur.

La variable numbers peut maintenant contenir plus de deux nombres.

```
if (operator == '+')
    return numbers.get(0) + numbers.get(1);
if (operator == '-')
    return numbers.get(0) - numbers.get(1);
// ...
```

Se transforme en :

```
if (operator == '+')
    return numbers.stream().mapToDouble(Double::doubleValue).sum();
if (operator == '-')
    return numbers.get(0) - numbers.subList(1, numbers.size())
            .stream().mapToDouble(Double::doubleValue).sum();
// ...
```

Palier 3 : Opération composite

On veut maintenant, en plus des cas précédents, gérer les cas suivants :

```
>>> 20 + 2 - 1
21
>>> 2 * 2 + 4
8
>>> 4 + 2 * 2
8
```

On veut maintenant gérer les opérations suivantes :

```
OPERATION → NOMBRE OPERATEUR OPERATION
          | NOMBRE
OPERATEUR → '+'
          | '-'
          | '*'
          | '/'
NOMBRE    → constant
```

Ici on va chercher à faire des opérations différentes. Il va donc falloir réorganiser notre code. Voici le code actuel de la fonction Compute :

```java
public static Double Compute(String operation) {
    // Suppression des espaces
    operation = operation.replaceAll("\\s", "");

    // Détection de l'opérateur
    char operator;
    if (operation.contains("+"))
        operator = '+';
    else if (operation.contains("-"))
        operator = '-';
    else if (operation.contains("*"))
        operator = '*';
    else if (operation.contains("/"))
        operator = '/';
    else {
        System.out.println("Opération invalide");
        return 0d;
    }

    // Extraction des nombres
    List<Double> numbers = Arrays.stream(operation.split("[+/*-]"))
            .map(Double::parseDouble).collect(Collectors.toList());
    Double firstNumber = numbers.get(0);
    Stream<Double> sequel = numbers.subList(1, numbers.size()).stream();

    // Calcul du résultat
    if (operator == '+')
        return sequel.reduce(firstNumber, Double::sum);
    if (operator == '-')
        return sequel.reduce(firstNumber, (a, b) -> a - b);
    if (operator == '*')
        return sequel.reduce(firstNumber, (a, b) -> a * b);
    else
        return sequel.reduce(firstNumber, (a, b) -> a / b);
}
```

Il apparait que certaines étapes sont inadaptées à notre nouveau problème. Par exemple, « détection de l'opérateur » n'a pas de sens quand il y en a plusieurs.

On va donc remplacer cette partie par une « détection des opérateurs ».

Le calcul du résultat devra par conséquent être adapté également.

Détection des opérateurs

Au lieu de détecter un unique opérateur, on va détecter une liste d'opérateurs qu'on appellera `operators`.

Pour cela, on va parcourir la chaîne de caractères `operation`, et on va stocker chaque opérateur dans une liste.

Calcul du résultat

Il est profondément changé. Il doit maintenant refléter le fait qu'il y a plusieurs opérateurs. On va parcourir deux listes en même temps : `numbers` et `operators`.

On va devoir faire deux passes dans l'opération :

- Une pour la multiplication/division
- Une pour l'addition/soustraction

La multiplication/division étant prioritaire, on veut l'effectuer en amont.

Au fur et à mesure qu'on calcule la multiplication/division, on construit une nouvelle liste `multiplied_numbers` qui contient les nombres à additionner/soustraire. Les nombres à soustraire seront sous forme négative, et les nombres à additionner sous forme positive.

On a plus qu'à faire la somme de multiplied_numbers.

Palier 4 : Parenthèses

On veut maintenant, en plus des cas précédents, gérer les cas suivants :

```
>>> 2 * (2 + 4)
12
>>> (4 + 2) * 2
12
>>> (4 + (2 / 2)) * 2
10
```

On veut maintenant gérer les opérations suivantes :

```
OPERATION → LEFT_PAR OPERATION RIGHT_PAR
          | NOMBRE OPERATEUR OPERATION
          | NOMBRE
OPERATEUR → '+'
          | '-'
          | '*'
          | '/'
LEFT_PAR  → '('
RIGHT_PAR → ')'
NOMBRE    → constant
```

A venir…

Palier 5 : Puissance et fonctions

On veut maintenant, en plus des cas précédents, gérer les cas suivants :

```
>>> sin(0)
0
>>> 2 ^ 3
8
>>> cos(0)
1
```

On veut maintenant gérer les opérations suivantes :

```
OPERATION → FUNCTION LEFT_PAR OPERATION RIGHT_PAR
          | LEFT_PAR OPERATION RIGHT_PAR
          | NOMBRE OPERATEUR OPERATION
          | NOMBRE
OPERATEUR → '+'
          | '-'
          | '*'
          | '/'
          | '^'
LEFT_PAR  → '('
RIGHT_PAR → ')'
NOMBRE    → constant
```

A venir…