m2103

TM confiné

Projet calculette

Table des matières

1 Présentation : le système à réaliser

On souhaite réaliser une calculette, c'est-à-dire un programme interactif avec lequel l'utilisateur

- tape des expressions,
- voit s'afficher leur valeur.

Exemple de déroulement, avec la version de base :

```
* Calculette v1.0
> 2 + 2
= 4
> 2*(12+1)-5
= 31
> \q
Bye
```

Dans un second temps, la calculette améliorée proposera des variables pour stocker des valeurs :

```
* Calculette v2.0
> a = 2+2
= 4
> b = 3*2+a
> 10
> a*b + 2
= 42
```

2 Boucle d'interaction

Voici une manière de faire une boucle d'interaction

```
Scanner in = new Scanner(System.in);
  System.out.println("Essai de boucle d'interaction");
  while (true) {
      System.out.print("> ");
      if (!in.hasNextLine()) {
          break;
      }
      String line = in.nextLine().trim();
      if (line.isEmpty()) {
          continue;
      }
      if (line.equals("\\q")) {
          break;
      }
      System.out.format(                    // à remplacer
            "[calcul et affichage de \"%s\"]\n",
            line);
  }
  System.out.println("Bye.");
```

L'instruction `format` en fin de corps de boucle est un "*placeholder*" qui doit être remplacé par du code qui fait réellement le travail voulu.

3 Évaluation

On pourrait envisager dans un premier temps, d'avoir une simple fonction d'évaluation

```
int value = evaluation(line);
System.out.format("> %d\n", value);
```

mais il faut penser que

- l'évaluation peut échouer (erreur de syntaxe, division par zero,...),
- la calculette devra mémoriser des variables, elle a donc un *état*.

On opte donc pour

- faire de la calculette un **objet** (classe `Calculator`),
- traiter les problèmes par des **exceptions**.

```
Calculator calculator = new Calculator();
...

try {
    int value = calculator.evaluation(line);
    System.out.format("> %d\n", value);
}
catch (SyntaxErrorException ex) {
        System.out.format("! Incorrect syntax %s\n",
                          ex.getMessage());
}
catch (EvaluationErrorException ex) {
        System.out.format("! Evaluation failed %s\n",
                          ex.getMessage());
}
```

Ceci nous conduit à mettre le code de la calculette sous forme d'un package `my.calculator` avec (pour commencer !) les classes

- `Calculator` avec sa méthode `evaluation()`,
- `SyntaxErrorException`
- `EvaluationErrorException`

4 Évaluation d'une somme

Dans un premier temps, on va réduire nos ambitions au calcul d'une somme de nombres :

```
> 20 + 1 + 400
= 421
```

4.1 Algorithme

Si on sait décomposer la ligne en éléments, ce n'est pas très compliqué :

```
ALGORITHME
lire un élément, vérifier que c'est un nombre
mettre sa valeur dans total

lire un élément
tant que c'est le symbole "+"
faire
  | lire un élément, vérifier que c'est un nombre
  | ajouter sa valeur à total
  | lire un élément

vérifier qu'on est arrivé à la fin
retourner le total
```

4.2 Découpage en `Tokens` par un `Tokenizer`

Un objet (que nous appellerons `Tokenizer`) sera chargé d'extraire les éléments (*tokens*) d'une ligne.

```java
class Tokenizer implements Supplier<Token> {
    public Tokenizer(String line) {
        // ...
    }

    @Override
    public Token get() {
        // ...
}
```

4.3 Documentation java : l'interface `Supplier<T>`

`Supplier<T>` est une interface fonctionnelle générique prédéfinie, pour des classes ayant une méthode `get` qui fournit un objet :

```java
interface Supplier<T> {
    T get();
}
```

4.4 Que faire avec un `Token` ?

Un token représente un élément lexical de la ligne, un "lexème".

À chaque `Token`, on pourra demander ce qu'il représente (un nombre, un symbole, …), et si c'est un nombre, lui demander sa valeur.

Il est pratique de considérer qu'un token spécial marque la fin du texte, et qu'un autre type de token retourne les éléments inconnus.

4.5 Le code Java

Le code Java ressemble beaucoup à l'algorithme, si on introduit les bonnes méthodes.

```
int evaluationSomme(String line) throws SyntaxErrorException {
    Tokenizer tokenizer = new Tokenizer(line);
    Token token = tokenizer.get();
    checkSyntax(token.isNumber(), "Number expected");
    int total = token.value();

    token = tokenizer.get();
    while (token.isSymbol("+")) {
        token = tokenizer.get();
        checkSyntax(token.isNumber(), "Number expected");
        total += token.value();
        token = tokenizer.get();
    }
    checkSyntax(token.isEnd(),
        String.format("End of expression expected, %s found", token));
    return total;
}
```

5 Les Tokens, implémentation

En partant de l'énumération

```
public enum TokenType {
    NUMBER,
    SYMBOL,
    INVALID,
    END,
}
```

on peut implémenter simplement les `Tokens`

```
class Token {

    final TokenType type;
    final String string;

    public Token(TokenType type, String string) {
        this.type = type;
        this.string = string;
    }

    public boolean isNumber() {
        return type == TokenType.NUMBER;
    }
```

```
    int value() {
        return Integer.parseInt(string);
    }
    // ...
}
```

Les méthodes de la classe `Token` sont laissées en exercice.

6 Le `Tokenizer`

```
class Tokenizer implements Supplier<Token> {

    final String SYMBOLS = "+-*/()";

    String line;
    int next;

    public Tokenizer(String line) {
        this.line = line;
        this.next = 0;
    }

    @Override
    public Token get() {
        // ignorer les espaces
        while (next < line.length()
                && Character.isSpaceChar(line.charAt(next))) {
            next++;
        }
        // on est au bout ?
        if (next >= line.length()) {
            return new Token(TokenType.END, "");
        }
        // délégation du travail à une méthode par type,
        // déterminé par le premier caractère

        char first = line.charAt(next);
        if (Character.isDigit(first)) {
            return getNumber();
        } else if (SYMBOLS.indexOf(first) >= 0) {
            return getSymbol();
        } else {
            next++;
            return new Token(TokenType.INVALID, Character.toString(first));

        }
    }

    private Token getNumber() {
        StringBuilder builder = new StringBuilder();
```

```
        do {
            builder.append(line.charAt(next));
            next++;
        }
        while (next < line.length()
                && Character.isDigit(line.charAt(next)));
        return new Token(TokenType.NUMBER, builder.toString());
    }

    private Token getSymbol() {
        String string = Character.toString(line.charAt(next));
        next ++;
        return new Token(TokenType.SYMBOL, string);
    }
}
```

7 Travail à faire

1. Mettre en place un projet, avec

- une classe `Demo`, avec un `main` qui lance le travail
- un paquetage `my.calculator`, avec `Calculator`, `Token`, `Tokenize`, etc.

2. Compléter pour faire fonctionner le code
3. Modifier pour permettre de faire des sommes et des différences

```
> 14 + 3 - 10 -2
= 5
```