m2103

TM confiné

# Projet calculette (phase 3)



## Table des matières

# 1 Présentation

À lire avant de coder.

## 1.1 Ce qui a été fait

À ce stade, notre calculette sait évaluer

- des sommes/différence de nombres `15 + 2 - 7 + 1`,
- des sommes/différences de produits/quotients de nombres : `2\*3 + 4\*5\*6 - 10/2`,
- des expressions parenthésées : `(2+3) \* (4+5)`.

## 1.2 Ce qu'on va faire : variables

Nous allons ajouter la possibilité d'utiliser des variables pour mémoriser des valeurs.

```
> resultat = 2 + 2
> 4
> resultat = resultat + 1
> 5
```

## 1.3 Ce que ça implique

- que le `Tokenizer` sache reconnaître un nouveau type de `Token`, qu'on va appeler "mot".
- que la calculette sache évaluer une **affectation**, de la forme "*mot = expression*"
- que la calculette sache retourner la valeur affectée à une variable.

## 1.4 Précisions

**Variable non définies** : on décide que, si on tente d'obtenir la valeur d'une variable non définie, il se produit une exception. (C'est un choix, on aurait pu décider que les variables ont une valeur par défaut, 0 par exemple).

**Les affectations comme expressions** : Dans les exemples, vous remarquez que l'exécution d'une affectation retourne une valeur (affichée par la boucle lecture-évaluation-affichage).

Il n'est donc pas complètement artificiel de considérer qu'une affectation peut apparaitre à l'intérieur d'une expression.

```
> a = (b = 3) + 1    # affectation de b dans une expression
= 4
> b
= 3
> x = y = 5          #  affectation multiple !
= 5
> y
= 5
```

C'est un choix qui correspond à ce qui est fait en langage C, par exemple.

### 1.5 Remarque

Une autre possibilité aurait été de différencier les affectations des expressions, par un mot-clé

```
> let x = 2 + 2
```

c'est ce qu'on a fait l'an dernier. Mais cette année on va faire sans `let` (en fait ça sera plus simple !).

### 1.6 Plan de travail

1. Compléter le `Tokenizer` pour reconnaître les "mots", en ajoutant un type supplémentaire `Token` (`TokenType ::WORD`).
2. Ajouter une structure de données pour mémoriser la valeur des variables. Spoiler : de type `Map<String,Integer>`
3. Compléter l'analyseur pour reconnaitre et évaluer les variables et les affectations.

### 1.7 Pour commencer :

- Sauvez votre projet Netbeans dans l'état où il est
- Faites-en une copie, dans votre dépôt GITLAB
- Dans votre dépôt GITLAB, complétez le fichier `README` pour qu'il indique à quoi correspond chaque projet Netbeans
- Faites maintenant : `git add`, `push`, and `commit`.

Vous travaillerez désormais sur la copie.

## 2 Ajout d'un nouveau type de `Token`

Pour pouvoir compléter le `Tokenizer` indépendamment du code de `Calculator` (que l'on fera évoluer plus tard), on introduit des tests unitaires sur le `Tokenizer` (ce qui aurait dû être fait avant, et aurait évité bien des problèmes…)

Étapes :

1. ajouter des **tests unitaires** pour le `Tokenizer` existant,
2. préciser les fonctionnalités voulues pour les Tokens qui représentent des mots,
3. ajouter des **tests** pour ces fonctionnalités,
4. compléter le code, tester, modifier, tester, etc.

### 2.1 Tests unitaires pour l'existant

Copiez-coller la classe de test ci-dessous :

```
package my.calculator;

import org.junit.Test;
import static org.junit.Assert.assertEquals;
import static org.junit.Assert.assertFalse;
```

```java
import static org.junit.Assert.assertTrue;

public class TokenizerTest {

    @Test
    public void testEnd() {
        String line = "    ";
        Tokenizer tokenizer = new Tokenizer(line);

        Token token = tokenizer.get();
        assertTrue(token.isEnd());
        // détection de confusions
        assertFalse(token.isNumber());
        assertFalse(token.isSymbol("+"));
    }

    @Test
    public void testNumber() {
        String line = " 123   ";
        Tokenizer tokenizer = new Tokenizer(line);

        Token token = tokenizer.get();
        assertTrue(token.isNumber());
        assertEquals(123, token.value());
        //
        assertFalse(token.isEnd());
        assertFalse(token.isSymbol("+"));
    }

    @Test
    public void testSymbol() {
        String line = " *   ";
        Tokenizer tokenizer = new Tokenizer(line);

        Token token = tokenizer.get();
        assertTrue(token.isSymbol("*"));
        //
        assertFalse(token.isNumber());
        assertFalse(token.isEnd());
    }

    @Test
    public void testSequence() {
        String line = "12+(34)";
        Tokenizer tokenizer = new Tokenizer(line);

        Token token = tokenizer.get();
        assertTrue(token.isNumber());
        assertEquals(12, token.value());

        token = tokenizer.get();
```

```
        assertTrue(token.isSymbol("+"));

        token = tokenizer.get();
        assertTrue(token.isSymbol("("));

        token = tokenizer.get();
        assertTrue(token.isNumber());
        assertEquals(34, token.value());

        token = tokenizer.get();
        assertTrue(token.isSymbol(")"));

        token = tokenizer.get();
        assertTrue(token.isEnd());

    }
}
```

Si ces tests échouent, corrigez de toute urgence les erreurs de vos classes `Tokenizer` et `Token`.

### 2.2 Fonctionnalités attendues.

Quand le `Calculator` analysera une ligne, il aura besoin de savoir si un `Token` représente un “mot” ou autre chose. Et si oui, de connaitre ce mot.

```
class Token {
    boolean isWord() { .... }
    String  word() {....}
    }
```

### 2.3 Tests supplémentaires du `Tokenizer`

Exemple de tests à ajouter :

```
    @Test
    public void testWord() {
        String line = " hello  ";
        Tokenizer tokenizer = new Tokenizer(line);

        Token token = tokenizer.get();
        assertTrue(token.isWord());
        assertEquals("hello", token.word());

        // etc.
    }

    @Test
    public void testSequenceWords() {
        // pour vérifier qu'on ne consomme pas un
        // caractère en trop
```

```
        String line = "abc=def";
        Tokenizer tokenizer = new Tokenizer(line);

        Token token = tokenizer.get();
        assertTrue(token.isWord());
        assertEquals("abc", token.word());

        token = tokenizer.get();
        assertTrue(token.isSymbol("="));

        token = tokenizer.get();
        assertTrue(token.isWord());
        assertEquals("def", token.word());

        token = tokenizer.get();
        assertTrue(token.isEnd());
    }
```

Netbeans fournira des “stubs” pour les méthodes manquantes.

## 2.4 Modification des classes existantes

### 2.4.1 `TokenType`, `Token`

Il ne vous faudra que quelques secondes pour

- dans l’énumération `TokenType`, ajouter `WORD`,
- dans `Token`, compléter `isWord()` et `word()`.

### 2.4.2 `Tokenizer`

Pour le `Tokenizer`, un peu plus de travail.

**Le fragment de code concerné** est

```
char first = line.charAt(next);
if (Character.isDigit(first)) {
    return getNumber();
} else if (SYMBOLS.indexOf(first) >= 0) {
    return getSymbol();
} else {
   ...
}
```

qui se base sur le premier caractère lu pour savoir si on est au début d’un nombre, sur un symbole reconnu, ou autre chose.

si c’est une lettre, ca sera un “mot”.

**Détection du début d’un mot**

```
char first = line.charAt(next);
if (Character.isDigit(first)) {
    return getNumber();
} else if (Character.isLetter(first)) {    // début de mot
    return getWord();                      //
} else if (SYMBOLS. .....) {
  ...
}
```

qui renvoie à une méthode `getWord` qui accumule les caractères du mot, comme dans `getNumber`.

**Les différences** avec `getNumber` :

- un "mot" commence par une lettre, suivie (conventions habituelles pour les identificateurs dans les langages de programmation) de lettres ou de chiffres, comme "test23".
- le type `TokenType.WORD`.

Éventuellement, vous pouvez admettre aussi le blanc souligné, comme dans "covid_19".

**Conseil** : définir deux méthodes pour caractériser les caractères :

```
static boolean isValidFirstCharForWord(char c);
static boolean isValidCharForWord(char c);
```

## 3 Structure de données pour mémoriser les valeurs

La calculatrice mémorise, dans une "table", la valeur associée aux variables qui sont identifiées par leur nom (une chaine de caractères).

```
Map<String,Integer> table = new HashMap<>();  // a no-brainer
```

## 4 Analyse et évaluation

Il suffit d'une légère modification dans la définition d'un facteur :

Le nom d'une variable peut être suivi du symbole "=" et d'une expression. Dans ce cas, il s'agit d'une affectation à la variable.

```
expr ::=  term    {  opAdd  term }
term ::=  factor  {  opMult factor }
factor ::=   number
        |    word [ = expr ]          // modification
        |    opUnary factor
        |    "(" expr  ")"
```

Notation : les crochets indiquent une partie facultative.

### 4.1 Ajout de tests unitaires pour `Calculator`

Ca ne peut pas faire de mal, et ça fait gagner beaucoup de temps…

```
public void testAssignments()
    throws SyntaxErrorException, EvaluationErrorException {
    Calculator c = new Calculator();
    assertEquals(12, c.evaluation("num = 3*4"));
    assertEquals(12, c.evaluation("num"));
    assertEquals(2, c.evaluation("den = 2"));
    assertEquals(2, c.evaluation("den"));
    assertEquals(6, c.evaluation("num / den"));

    assertEquals(10, c.evaluation("(a = 2+1) + (b = 2*3 + 1)"));
    assertEquals(3, c.evaluation("a"));
    assertEquals(7, c.evaluation("b"));
}
```

### 4.2 Modification de `get_factor_value`

Indication :

```
si le token est un nombre
  | ...appeler get_number_value()
sinon si c'est un mot                    *
  | c'est une variable !  noter son nom   *
  | avancer au token suivant              *
  | si c'est "="                          *
  |  |  avancer au token suivant          *
  |  |  appeler get_expr_value()          *
  |  |  affecter la valeur à la variable  *
  | retourner la valeur de la variable    *
sinon si c'est une parenthèse ouvrante
  |  ...
```

**Rappel** : si la variable n'est pas dans la table quand on veut obtenir sa valeur, il faut lever une `EvaluationErrorException`.