Héritage en java : Calculatrice SDC *

novembre 2015

Travail à rendre : le code complet du projet SDC sous forme d'une archive `tar.gz`. L'archive comportera trois répertoires :

— un répertoire `sdc` comportant le code complet ; ce code devra être exécutable.
— un répertoire `test_sdc` comportant l'ensemble des tests unitaires ;
— un répertoire `modif` comportant une copie de l'ensemble des fichiers que vous avez modifiés (ou créés).
— un rapport décrivant *i)* l'architecture générale de l'application et explicant l'intérêt de celleci ; *ii)* les modifications que vous avez faites

1 Introduction

1.1 Contexte

Vous êtes nouvel embauché dans une entreprise dont le département de Recherche et Développement a récemment développé une petite calculette du nom de SDC. Il vous demande de modifier ce logiciel pour y ajouter de nouvelles fonctionnalités.

Votre équipe dispose d'un ensemble de tests de régression qui vous permettront non seulement de vérifier que les nouvelles fonctionnalités sont correctement réalisées mais qu'elles ne remettent pas en cause les autres services fournis par la calculette. Naturellement ces tests ne sont que partiels et comportent peut-être (sûrement !) des erreurs.

1.2 Objectifs

Ce TP a deux objectifs :

1. Vous allez devoir vous plonger dans un projet existant composé d'environ 500 lignes de code afin de l'enrichir. Vous allez donc devoir apprendre rapidement à vous former sur un environnement nouveau et une architecture logicielle nouvelle, cas typique d'un nouvel embauché.
2. Vous familiarisez avec une architecture objet « complexe » (c.-à-d. avec plus de trois classes).

*Guillaume Wisniewski, `guillaume.wisniewski@limsi.fr` sur une idée originale de L. Pautet

2 Description de la calculatrice SDC

2.1 Principe

SDC est une calculette simple et interactive qui est capable (pour le moment) d'effectuer les 4 opérations usuelles sur des entiers. La notation acceptée par SDC est la notation polonaise inversée (comme une calculette HP). Dans cette notation au lieu d'écrire :

$$2 + (3 \times 4)$$

on écrit :

$$2\ 3\ 4\ \times\ +$$

Pour bien comprendre la notation polonaise inversée, il faut considérer la structure en arbre des expressions arithmétiques. Par exemple, l'expression précédente $(2 + (3 \times 4))$ est représentée par l'arbre de la Figure 1.

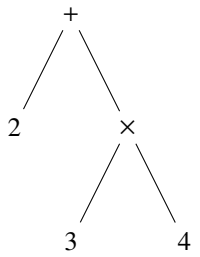

FIGURE 1 – Arbre d'analyse de l'expression $2 + (3 \times 4)$

Il y a 3 manières de linéariser cette représentation qui correspondent aux 3 manières de traverser un arbre binaire :

- traversée nœud-enfant-enfant ;
- traversée enfant-nœud-enfant ; ;
- traversée enfant-enfant-nœud ;

ce qui donne pour l'arbre de l'arbre de la Figure 1 :

- traversée nœud-enfant-enfant : (+ 2 (×3 4)) (notation utilisée dans le langage de programmation LISP) ;
- traversée enfant-nœud-enfant : (2 + (3 × 4)) (notation conventionnelle) ;
- traversée enfant-enfant-nœud : (2 (3 4 ×) +) (notation polonaise inversée utilisée dans les calculettes HP).

La notation polonaise inversée est particulièrement appréciée puisqu'elle facilite grandement le calcul d'une expression arithmétique et nécessite, pour cela, simplement une pile. Étant donné une suite de jetons (*token* en anglais) représentant une expression arithmétique en notation polonaise inversée, la procédure suivante implémente une calculette telle que SDC.

```
while (true) {
  String token = readNextToken();

  if token.isNumber() {
    stack.push(token.toInt())
  } else if token.isOperation() {
```

```
      int value1 = stack.pop();
      int value2 = stack.pop();

      int resultat = doOperation(token, value1, value2);

      stack.push(resultat);
    } else {
      // autre commande
    }
  }
```

2.2 Code fourni

Le code fourni correspond à une version simple de la calculatrice capable d'effectuer des opérations sur les nombres entiers et de reconnaître les commandes `quit` et `clear` (pour vider la pile).

Il est (fortement) conseillé de ne pas chercher à comprendre la totalité du code, mais uniquement les interfaces utiles pour réaliser les modifications demandées. La calculatrice est construite autour de 5 classes « principales » :

- la classe `SDC` qui comporte la boucle principale (exécution des commandes entrées) ;
- la classe `Factory` qui permet de décrire l'ensemble des opérations et des types de nombres utilisables par la calculatrice ;
- l'interface `Symbol` qui décrit un token (nombre ou commande). Cette interface est mise en œuvre au travers de deux classes abstraites, `Value` qui décrit une valeur (un nombre) manipulée par la calculatrice et `BinaryOperation` qui décrit une opération binaire (sur deux valeurs).

Les classes (concrètes) `IntegerValue`, `AddOperation` et `ClearSymbole` vous montrent comment ces classes abstraites peuvent être implantées pour modéliser des nombres entiers et réaliser des additions ou une ré-initialisation de la pile.

3 Questions

1. Implanter la commande view qui imprime le contenu de la pile à l'écran. Le format d'impression est donné dans l'exemple ci-dessous :

   ```
   [wisniews@m177] ...nement2014/java/td/sdc > java SDC
   Welcome to sdc. Go ahead type your commands ...
   > 2 4 6 8 10 12
   > view
   6 ----> 2
   5 ----> 4
   4 ----> 6
   3 ----> 8
   2 ----> 10
   1 ----> 12
   ```

2. Ajouter des valeurs de type rationnel ainsi que les quatre opérations correspondantes plus, moins, multiplier, diviser (+ - * /). Un nombre rationnel de SDC

s'écrira N#D où N désigne le numérateur et D le dénominateur, # représentant la barre de fraction.

Lors de cette mise en œuvre s'assurer qu'il n'est pas possible de mélanger un entier et un nombre non-entier. Plus exactement, si une opération trouve deux opérandes de type différent, l'opération est avortée, et le message *types incompatibles* est affiché.

```
[wisniews@m177] ...nement2009/java/td/sdc > java SDC
Welcome to sdc. Go ahead type your commands ...
> 12#5
> 2#3
> view
2 ----> 12#5
1 ----> 2#3
> +
> view
1 ----> 46#15
> 3 +
IncompatibleType
```

3. Implanter l'opération valeur absolue `||` sur tous les nombres. Cette instruction prend un nombre X comme unique opérande et retourne X si X est positif, sinon -X. Pour implanter cette opération, il faut permettre la mise en œuvre des opérations unaires. Exemple :

```
[wisniews@m177] ...nement2009/java/td/sdc > java SDC
Welcome to sdc. Go ahead type your commands ...
> -1 ||
> 2 ||
> view
2 ----> 1
1 ----> 2
>
```

4. Ajouter le type de données booléen et implanter les trois opérations suivantes :
   — & (et logique)
   — | (ou logique)
   — ~ (négation logique)
5. Rajouter aux types de nombres, les opérations de comparaison suivantes :
   — < (inférieur)
   — > (supérieur)
   — = (égal)

   Ces opérateurs dépilent les deux opérandes en haut de la pile (qui doivent être du même type), effectuent la comparaison et empilent le résultat qui est une valeur booléenne.

   Par exemple :

```
[wisniews@m177] ...nement2014/java/td/sdc > java SDC
Welcome to sdc. Go ahead type your commands ...
> 3 2 = ~
> view
1 ----> true
```

```
> 3 10 <
> 8 2 >
> &
> view
1 ----> true
```

Conseil : Il peut être intéressant de noter que l'opération « < » se déduit automatiquement de « > » par l'inversion des arguments et que l'opération « = » est vraie dès que ni « < », ni « > » ne sont vraies.

6. Rajouter la possibilité d'utiliser des variables. Le nom d'une variable correspond à toute séquence de caractères commençant par la lettre $ Par exemple : `$X` ou `$aXb`. Les noms de variables ne sont pas sensitifs à la casse, c'est-à-dire que `$Xyz`, `$xyz` et `$XYZ` dénotent tous la même variable.

   Avec les variables il faut implanter l'instruction d'affectation =>. Cette commande, qui doit être suivie du nom d'une variable, dépile l'opérande en haut de la pile et l'affecte à la variable (voir l'exemple). Si le => n'est pas suivi d'une variable, sdc émet une erreur. Lors d'une affectation le nom de la variable n'est pas précédé du signe $:

   ```
   [wisniews@m177] ...nement2009/java/td/sdc > java SDC
   Welcome to sdc. Go ahead type your commands ...
   > 3 => x
   ```

   Si une variable est utilisée dans une expression, sa valeur est utilisée dans le calcul de l'expression. Toute utilisation d'une variable non initialisée provoque l'émission du message suivant : opération illégale

   Une variable peut prendre au cours de l'exécution d'une session sdc des valeurs de plusieurs types.

   Exemple :

   ```
   Welcome to sdc. Go ahead type your commands ...
   > 3 => x
   > view
   > $x
   > view
   1 ----> 3
   > $y
   Illegal operation: unknown variable. Ignore last command line
   > 4 $x * => y
   > $x $y < => z
   > $z
   > view
   2 ----> 3
   1 ----> true
   > $x => z
   ```

7. Implanter les instruction `if ... endif` et `if ... else ... endif`. L'instruction `if` dépile la valeur en haut de la pile qui doit être une valeur booléenne. Si la valeur n'est pas booléenne, les instructions if ne sont pas exécutées.

Par contre, si la valeur booléenne est `true`, alors les opérations à la suite du `if` et jusqu'au `else` sont exécutées. Sinon, seules les opérations entre le `else` et le `endif` sont exécutées. S'il n'y a pas de `else`, aucune opération est exécutée. De même, si la valeur booléenne est `false`. Bien entendu il est possible d'avoir des `if ... endif` imbriqués.

Exemple : le « programme » suivant permet de déterminer le nombre de racines d'un polynome du seconde degré. Les point-virgules indiquent les commentaires.

```
Welcome to sdc. Go ahead type your commands ...

;  petit programme pour determiner le nombre de racines d'un polynome
;  du 2eme degre: A X^2 + B X + C
;
;  entrez ci dessous les 3 cœfficients A B C dans cet ordre

 7.0  5.0  -6.5

; le programme commence ici

=> C   ;  C vaut -6.5
=> B   ;  B vaut  5.0
=> A   ;  A vaut  7.0

$B $B * 4.0 $A $C * * - => $Delta  ;  Delta = B*B - 4*A*C

0.0 $Delta =  if
   1  ; Le polynome a une seule racine

else
   0.0 $Delta < if
      2  ; Le polynome a 2 racines distinctes
   else
      0  ; Le polynome n'a pas de racines reelles
   endif
endif

view
 -> 2
```