Labo 08 Calculatrice - Rapport

Sacha Bron - Valentin Minder // HEIG-VD // POO // 12.12.2014

Introduction

Le but de ce laboratoire est de créer un programme capable de gérer une calculatrice fonctionnant en notation polonaise inverse (Reverse Polish Notation). Dans ce mode de calcul, les opérateurs sont postfixés, c'est-à-dire placés après leur(s) opérande(s)).

Exempls: pour effectuer `(1 + 2) * 3` en notation classique, on note `1 2 + 3 *` en notation polonaise. L'avantage est l'absence d'ambiguité sans utilisation de parenthèses.

Méthodologie et implémentation

Comme pour beaucoup de problèmes en informatique, il y a plusieurs manière d'aborder ce problème. Voici comment nous avons raisonné et procédé.

Classes principales

Notre programme est constitué de plusieurs classes ayant des buts précis. Ce programme illustre le modèle de conception réutilisable MVC (Design Pattern Model-View-Controler).

- le modèle `State` : structure de donée représentant l'état interne de la calculatrice.
- les contrôlleurs `Operator` : une hiérarchie de différents opérateurs interagissant avec le modèle.
- les vues à choix `JCalculator` (mode graphique) et `Calculator` (mode console) : fournissent une interaction avec l'utilisateur, et appellent les contrôlleurs lors d'actions utilisateurs.

Par ailleurs, la classe `Pile` est reprise telle quelle du labo 7 pour représenter la stack du modèle.

Structure de données interne (modèle)

L'état de la calculatrice peut être défini comme suit: la valeur actuelle, l'état de la pile des valeurs, la valeur en mémoire, la présence d'une éventuelle erreur, et le caractère mutable de la valeur actuelle.

La valeur actuelle est manipulée sous forme de `String` afin de faciliter la modification (ajout et retrait en fin de chaine). Elle est transformée en `double` pour calcul au moment d'une opération. La valeur actuelle est modifiable (mutable) lors d'une insertion avec des digits p.ex. En revanche, elle ne l'est pas lors qu'il s'agit d'un résultat calculé ou provenant de la mémoire. Dans ce cas, une tentative de modification placera la valeur actuelle sur la pile et on travaillera avec une nouvelle valeur.

Comme les valeurs des opérandes en attente ne peuvent être mises ou enlevées que si elles sont les dernières entrées, les piles (*stack*) semblent être la structure de données la plus favorable à cette donnée.

La mémoire est une simple `String` qui peut être chargée (`MS` Memory Store, la valeur actuelle reste en copie) et déchargée (`MR` Memory Recall, la valeur actuelle est écrasée et devient immutable).

Gestion d'erreurs

Les erreurs arithmétiques possibles sont:

- division par 0 (lors de "division" de `x/y` avec `y = 0` ou lors de "1 sur x" de `1/x` avec `x = 0`)
- racine carrée d'un nombre inférieur à 0 (`sqrt(-x)` avec `x > 0`)
- valeurs `+/- Infinity` (dépassement de capacité, lorsqu'il n'est plus possible de représenter le nombre)
- valeurs `NaN` (Not a Number). Toutefois cela ne devrait pas se présenter, car les cas sont déjà présents avant (p.ex. `0/0`, `0 * Inf`, `Inf-Inf`, `sqrt(-x)`, etc.. sont déjà arrêtés par les cas précédent)

De plus, lors de l'appel à un opérateur utilisant deux opérandes, la pile doit contenir au moins un élément (le premier opérande, le deuxième étant la valeur courante). Dans le cas contraire, une erreur est produite.

Dans ces cas la valeur courante est inutilisable et les seules touches valides sont `CE` (clear error) et `C` (clear) afin de nettoyer l'erreur sur la valeur courante. Toutes les autres sont sans effet. *Toutefois notre implémentation tolère le rappel d'une valeur avec MR pour remplacer l'erreur (car cela efface la valeur courante en erreur)*

Hiérarchie d'opérateurs

La hiérarchie complète est disponible dans le diagrammes des classes en fin de rapport. La logique est de regrouper des operateurs similaires.

Exemples:

- les opérateurs à 2 opérandes peuvent être éxécutés aux mêmes conditions (valeur courante valide et une valeur sur la pile).
- les opérateurs à 1 opérande peuvent être éxécutés aux mêmes conditions (valeur courante valide).
- les digit sont regroupés en un seul opérateur qui contient sa valeur à utiliser lors de son utilisation.
- les number operateurs (digit et point), avant de s'éxécuter, doivent sauver la valeur courante dans la stack si elle est immutable (càd le résultat d'un précédent calcul)

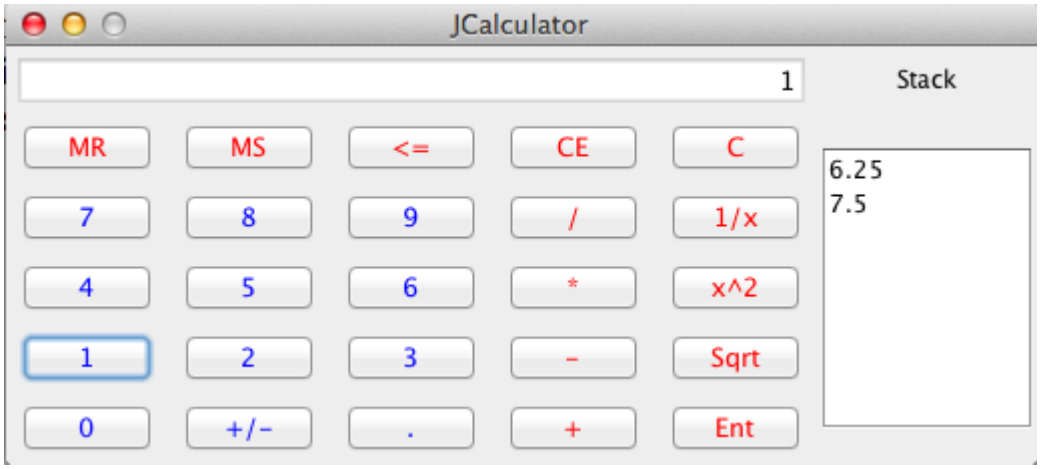
`Operator` est la racine et la seule à posséder une méthode `execute()`. Elle teste si l'operateur peut etre executé (méthode `check()`) avant de l'executer (méthode `exec()`). Ces méthodes sont définies dans les sous-classes, au besoin. En fait, `exec()` est défini pour chaque opérateur (chaque feuille de la hiérarchie) alors que `check()` est défini à des niveaux supérieurs (par groupe d'opérateurs, lorsque la condition de validité est la même).

Programmes et Tests

Les clases `JCalculator` et `Calculator` contiennent une méthode `main` et permettent de lancer la calculatrice en mode graphique respectivement console.

Exemple d'éxécution

En mode graphique.



En mode console.

```
<terminated> Calculator [Java App
> 1
1 []
> 2
2 [1.0]
> 3
3 [2.0, 1.0]
> +
5.0 [1.0]
> sqrt
2.23606797749979 [1.0]
> +
3.23606797749979 []
> exit
```

Conclusion

Il a été observé que la Notation Polonaise Inverse nécéssite moins de touches et d'opérations pour effectuer les mêmes calculs. Ce laboratoire illustre de plus les notions d'héritage, de liaison dynamique des liens et de pile.

Annexes

Le code imprimé est annexé. Pour simplifier, les sous-classes d'Operator ont été imprimées à la suite sur les mêmes feuilles.

Schéma des classes

Voici le schéma UML des classes.