Rapport

> *Auteur Changanaqui Yoann & Duruz Florian*

Points clés de l'implémentations

les opérations sont des classes distinctes héritant tout de la classe abstraite `Operation`, Les `Operator` lié aux opérations mathématiques ont été regroupéees dans une classe abstraite héritant de `Operator` cette dernière implémente une version constante de la méthode `execute()` la séquence étant identique, cette structure permet de définir et redéfinir uniquement le nécessaire; les classes dérivées doivent implémenter:

- `calculate()` : l'opération mathématique
- `hasError()` : Redéfinition quand d'autre erreur doivent être pris en compte(division par 0 et valeur hors domaine de définition.

Il en va de même pour les action lié à l'ajout de valeurs (numérique, point, etc...), ces dernière ont été regroupées dans la classe `DigitOperator` permettant de regrouper les variables et méthodes communes comprenant:

- `checkBeforeCommit()` : Guard clause qui définit si on rentre ou non dans la fonction
- `commitDigit()` : Action permettant d'ajouter un caractère la redéfinition permet de faire des actions avant ou après.

Concernant la structure de `State`, en accord avec une discussion avec l'assistant du cours, il à été décidé que les éléments de `State` ne doivent en aucuns cas être manipulé directement, les éléments sont manipulé via des fonctions précises permettant d'intéragir avec les attributs permettant un meilleur contrôle des intéractions et permettant aussi les exceptions lié au conversion de valeurs en retournant des valeurs par défaut.

Concernant le comportement vis-a-vis de la stack: si cette dernière est vide, on prend la valeur 0 par défaut.

Quelque cas particuliers:

Afin de rendre les valeurs intermédiaire immuable:

- Backspace ne fait rien si la valeur affiché est intermédiaire
- Le changement de signe est bloqué si la valeur affichée est intermédiaire

la Touche `Clear` efface aussi la valeur en mémoire (selon consigne de l'assistant)

L'implémentation de la pile repose sur une structure de nœuds chaînés, chaque élément référant au suivant. Ajout d'une méthode `clear()` pour réinitialiser la Stack facilement. Les méthodes `hasNext()` et `next()` sont implémentée directement dans la classe Node.

Tests effectués

Pour les tests nous nous sommes concentrés sur les cas ne pouvant pas être factorisé car lié à une opération particulières

pour les inputs type `digits` :

- vérifier qu'on ne peut pas mettre une suite de zéro avant un chiffre si ce dernier est avant la décimal (exemple: 03).
- Mettre plus d'un point (`PointOperator`) provoque une erreur "SYNTAX_ERROR" quand on tente de la mettre en stack via l'opérateur d'entrée (`EnterOperator`).

Pour les opérations mathématiques :

- Vérifier que les opérations liés aux divisions (division + réciproque) affiche une erreur "DIVID_BY_ZERO" si le dénominateur est égal à 0;
- Vérifier que l'opération SquareRoot lance une erreur "DOMAIN_ERROR" quand l'input est hors du domaine de définition de cette opération.

Tests pour la pile :

Pile vide :

- Vérifier qu'une pile nouvellement créée est vide, avec une taille de 0.
- Vérifier que les opérations sur une pile vide (comme pop ou top) retournent null sans provoquer d'erreur.

Empilement et dépilement :

- Vérifier que la méthode push ajoute correctement des éléments à la pile.
- Vérifier que la méthode pop suit la logique LIFO (Last In, First Out) en retirant le dernier élément ajouté.
- Vérifier que la taille de la pile se met à jour correctement après chaque opération.

Ordre des éléments :

- Vérifier que les éléments sont dépilés dans l'ordre LIFO (le dernier entré est le premier sorti).

Vidage de la pile :

- Vérifier que la méthode clear vide entièrement la pile.
- Vérifier qu'après un appel à clear, les opérations pop et top retournent null.

Conversion en tableau :

- Vérifier que la méthode toArray retourne un tableau contenant les éléments de la pile dans le bon ordre (le sommet en premier).
- Vérifier que le tableau résultant a la bonne taille et contient les bonnes valeurs.