Projet : calculatrice évoluée

Ce projet est à réaliser en binôme et à rendre selon les modalités et avant une date qui vous seront communiquées ultérieurement.
Le projet devra être original et signaler explicitement tout emprunt de code (toutefois, les imports de bibliothèques existantes sont à favoriser par rapport au copier/coller).
Il est à coder en Java. Les sources sont à rendre, accompagnées d'une documentation et de scripts permettant une compilation facile du projet (il est conseillé d'utiliser un outil tel que Gradle, qui, en plus de gérer la compilation et l'exécution, permet de télécharger automatiquement les dépendances déclarées).

Attention : la « belle programmation orientée objet » qui prendrait en compte les conseils donnés dans ce cours (bonnes pratiques, patrons de conception, ...) sera un critère primordial dans l'évaluation du projet.

I) Description brève

Le sujet consiste à programmer une calculatrice. Un classique ?... pas tout à fait !
En effet, il ne s'agit pas d'une calculatrice simulant une calculatrice physique avec des boutons : cela aurait peu d'intérêt par rapport à l'ergonomie qu'on peut avoir avec un clavier et un écran d'ordinateur.
Il s'agit plutôt d'un *shell*, c'est-à-dire une boucle d'entrée/évaluation/affichage (REPL : *read, eval, print, loop*).
Qui plus est, les extensions proposées feront de cette calculatrice un outil bien plus puissant que la plupart des calculatrices (qu'elles soient physiques ou bien des applications pour ordinateur).

Requis : Le logiciel que vous rendrez sera une calculatrice gérant au moins les 4 opérations usuelles (+, -, *, /) sur les nombres décimaux, disposant des caractéristiques décrites dans les 2 sections qui suivent et augmentée d'au moins 3 extensions parmi celles proposées.

Conseil : commencez par faire tranquillement les sections I-III, en suivant un *design* évolutif, mais sans pour autant vous encombrer la tête avec ce qui est écrit en IV-X. En un second temps, posez-vous et lisez le reste. Toutes les extensions ne rapporteront pas forcément autant de points et la même extension ne rapportera pas autant de points selon la façon dont elle a été traitée (cf. remarque sur la « belle POO »). Ainsi, privilégiez les extensions pour lesquelles vous avez une idée claire de l'architecture à utiliser.

II) L'interface utilisateur : la REPL

Comme dit plus haut, l'UI doit être une boucle de lecture d'entrée de l'utilisateur, évaluation et affichage. Un affichage typique après quelques entrées ressemblerait à cela :

```
> 1
1

> 4
4
```

```
> +
5
```

où les entrées de l'utilisateur sont écrites après le symbole « > » (l'« invite »), et le résultat est affiché à la ligne d'en dessous (sans invite).
Vous remarquerez l'ordre particulier des entrées pour cette addition. Il s'agit de la notation polonaise inversée. Voir la section suivante.
Il n'est pas demandé de faire forcément une interface graphique si ça ne se traduit pas par une amélioration franche de l'ergonomie. Cela dit, pour certaines extensions, cela se justifierait tout à fait.
Si vous faites une interface graphique, vous pouvez considérer Swing, JavaFX, voire un affichage sous forme de page web.

III) La syntaxe polonaise inversée (RPN).

Pour simplifier l'analyse des opérations entrées, on utilise souvent la « syntaxe polonaise inversée », pour laquelle on écrit les opérandes avant les opérations. Par exemple, l'opération $(15 + 4) \times 3$ s'écrit $15\ 4\ +\ 3\ *$.
La calculatrice que vous programmerez pourra en un premier temps ne gérer que la RPN, puis, via import d'une bibliothèque, gérer la syntaxe arithmétique usuelle (extension 2).
Le mode RPN fonctionne grâce à une pile (LIFO) d'opérandes, alimentée par les entrées successives de l'utilisateur : quand on entre une opérande (i.e. habituellement un nombre), celle-ci est ajoutée en sommet de pile ; quand on entre une opération, des opérandes en quantité nécessaire sont dépilées (sorti du sommet de la pile), le calcul est effectué, puis le résultat est empilé.
D'un point de vue interface utilisateur, on peut entrer les opérandes et les symboles d'opération une par une à chaque ligne entrée par l'utilisateur, ou bien en mettre plusieurs dans une seule ligne (séparées par des espaces).
Du point de vue des calculs, les deux modes d'entrée reviennent au même, mais pour l'affichage, ce sera différent : en effet, un affichage n'est demandé qu'après chaque ligne entrée. L'affichage sera typiquement le contenu actuel de la pile (si la calculatrice est programmée avec une interface graphique, on pourra se contenter de la dernière valeur empilée et une autre zone de la fenêtre sera chargée d'afficher l'état actuel de la pile.)

IV) Extension 1 : calculatrice multi-type

a) Ce dont il s'agit

Une extension rendant les choses très intéressantes (d'un point de vue de la conception objet), est de permettre la manipulation d'opérandes de types multiples.
Pour l'instant, dans les exemples, les opérandes étaient de simples nombres (entiers), mais on pourrait distinguer entre nombres entiers et nombres décimaux. Cela aurait notamment un impact pour le traitement de la division : euclidienne si les deux opérandes sont des entiers, décimale sinon ; mais cela aurait aussi un impact quand aux opérations disponibles : ainsi des opérations telles que le reste de divsion euclidienne, ou bien la factorielle, n'ont de sens que pour les entiers.
Dans cet exemple, il y a une relation de sous-typage (inclusion) assez claire : une opération implémentée pour les décimaux est automatiquement implémentée pour les entiers (il n'y a

qu'à considérer l'entier comme un nombre décimal). Mais rien n'empêche d'ajouter des types complètements différents :

- les fractions (sur lesquels les calculs devront être exacts ; les opérations arithmétiques devront réduire les fractions avant d'afficher le résultat)
- les booléens VRAI/FAUX, munis des opérations NON, ET, OU ;
- les ensembles, munis des opérations COMPLÉMENT,INTERSECTION, UNION ;
- langages réguliers sur un alphabet, munis de l'union, la concaténation et de l'étoile de Kleene.
- ...

On peut imaginer quelques autres types (les sections qui suivent vous en inspireront d'autres), avec parfois des conversions évidentes entre eux, parfois non.

b) La conception objet

Pourquoi ce genre de calculatrice est-il un défi ?
Typiquement, on aurait tendance à vouloir donner aux opérandes une interface commune dans laquelle toutes les opérations seraient déclarées, puis laisser, à l'exécution, la liaison dynamique s'occuper de choisir l'implémentation de l'opération à utiliser (par exemple choisir entre division euclidienne et division décimale), en fonction du type dynamique des opérandes.
Ici cette approche se heurte à plusieurs obstacles, notamment :

- chaque type d'opérande a des opérations différentes, au point que peut-être même aucune opération n'est commune à tous les types d'opérandes et que l'interface commune serait vide (comment alors faire opérer la liaison dynamique ?)
- une opération a généralement plusieurs opérandes (par exemple, pour faire « + », il faut deux opérandes), or la liaison dynamique n'effectue, à l'exécution, le choix d'une implémentation que par rapport au type d'un seul paramètre (le récepteur, `this`). On parle de *single dispatch*.

Tout ça pour dire qu'ici, inclure les opérations en tant que méthodes d'instance des types de données des opérandes n'apporte pas les avantages habituellement escomptés. La sélection d'une implémentation en fonction du type dynamique des opérandes devra être implémentée « à la main » pour simuler le *multiple dispatch*.
Une possibilité : constituer un dictionnaire des opérations, indexées par leurs noms et signatures (nombres et types d'opérandes), le choix de l'opération à effectuer se fera en fonction du type des opérandes en sommet de pile (et bien sûr du nom d'opération qui a été entré).
Ainsi chaque type de données implémenté se devra de remplir le dictionnaire avec les opérations qui lui appartiennent à son chargement (reste à décider d'une politique pour savoir où sera écrite l'implémentation d'une opération prenant comme opérandes, par exemple un entier et un booléen, ou bien une chaîne et un ensemble, etc.)

V) Extension 2 : syntaxe algébrique traditionnelle

Pour simplifier l'analyse des entrées, le sujet suggère jusqu'ici d'utiliser la syntaxe polonaise inversée, dans laquelle on entre une succession d'opérandes et d'opérations sans besoin de structurer. Ainsi, par exemple, aucune parenthèse n'était nécessaire.
Cependant nous sommes plus habitués à entrer des calculs via des expressions algébriques de la forme $3 * (4 - 5)$. Malheureusement, une telle expression, donnée par une grammaire dite « hors contexte », requiert une analyse plus poussée, demendant de bonnes connaissance en analyse syntaxique... et pas mal de réflexion si on veut implémenter cette analyse à la main.

Mais heureusement, dans le monde de Java, nous avons rarement besoin de réinventer la roue et il existe justement des bibliothèques dont le but est d'analyser les expression algébriques (en entrée : une chaîne de caractères, en sortie, un objet représentant l'expression).
Voici une liste de bibliothèques fournissant ce service :

- exp4j : `https://lallafa.objecthunter.net/exp4j/`
- CogPar : `http://cogitolearning.co.uk/docs/cogpar/index.html`
- ParserNG : `https://github.com/gbenroscience/ParserNG`
- mXparser : `http://mathparser.org/`

Essayez ces différentes bibliothèques et importez-en une dans votre projet.
Les objets-expressions utilisés devront soit être convertis dans vos propres types de données, soit *adaptés* (patron adaptateur).

Remarque : vous pouvez maintenant soit vous passer de la pile dans laquelle vous enregistriez le résultat de chaque ligne (en RPN), soit continuer à l'utiliser. Dans ce dernier cas, prévoyez une syntaxe (opérateur 0-aire « `pop` ») pour pouvoir utiliser, dans une formule algébrique, une opérande qui sera dépilée, exemple : `(pop() + 3) * pop()`.
Notez que l'opérateur `pop` est inutile en RPN, puisque les opérandes nécessaires étaient automatiquement dépilées. Par ailleurs, appeler `pop` en RPN serait un « calcul » qui ne fait rien (en effet, le résultat de toute opération étant mis immédiatement en pile, la signification de `pop` serait : dépile une valeur, puis... remets-là tout de suite en pile).
Dans la suite, les autres extensions sont décrites comme si cette extension n'était pas implémentée (exemples donnés en RPN). Si vous faites cette extension, il faudra adapter leur syntaxe.

VI) Extension 3 : rappel de valeurs

Pour faire des calculs un peu complexes, il peut être utile de rappeler des valeurs qui ont déjà été précédemment calculées (autrement qu'en dépilant des résultats récents).
Une telle fonctionnalité donnerait des pouvoirs supplémentaires par rapport à ce qui était déjà possible :

- permettre d'utiliser de vieux résultats (pas seulement les derniers empilés)
- permettre de réutiliser plusieurs fois un résultat connu sans modifier la pile
- permettre de stocker et réutiliser des valeurs stockées dans des variables nommées

En réalité, on devrait pouvoir effectuer 3 types de rappels : un rappel depuis la pile, en fournissant un index (2 variantes : en comptant depuis le haut et en comptant depuis le bas) ; ou bien un rappel depuis l'historique des calculs (idem, depuis le début et la fin), ou encore un rappel depuis une variable nommée (dans laquelle la valeur aura été stockée au préalable). Dans les 3 cas, la valeur rappelée est mise en pile (si calculcatrice RPN ; sinon, si syntaxe algérbrique, la fonction rappel retourne la valeur rappelée).
La principale différence la pile et l'historique, c'est que l'historique contient la valeur de toutes les lignes calculées depuis le début de l'exécution de la calculatrice, alors que les valeurs stockées en pile (empilées) disparaissent (sont dépilées) quand elles sont utilisées.
Exemple d'exécution :

```
> 3
3

> 4
4
```

```
> + // (3 + 4) * 7 remplace 3 et 4 par 7 dans la pile
7

> 5
5

> hist(0) // rappel première valeur de l'historique
3

> pile(0) // rappel base de la pile... sachant que 3 et 4 ont disparu !
7

> +
10

> pile(-1) // rappel sommet de pile (résultat: on duplique le sommet)
10

> +
20

> hist(-5)  // rappel 5 valeurs en arrière dans l'historique
3

> !x // dépile la valeur au sommet (3) et la stocke dans "x"
20  // nouveau sommet de pile

> ?x // lit la variable "x" et empile sa valeur
3
```

VII) Extension 4 : Variables symboliques, expressions et calculatrice symbolique

Une vraie calculatrice symbolique doit pouvoir manipuler en tant qu'opérandes, non seulement des constantes, mais aussi des variables symboliques [1] et des fonctions.
Le cœur de cette mécanique est l'ajout de nouveaux types d'opérandes (au minimum ; on peut détailler un peu plus si nécessaire) :

- `Variable` : une variable symbolique, caractérisée par son nom (par exemple x ou y)
- `Operation` : caractérisée par le nom de l'opérateur ou de la fonction à appliquer et la liste des opérandes sur laquelle elle sera appliquée. Il s'agit donc d'une opérande complexe, qui sera stockée avec toute sa complexité, sans forcément essayer de l'évaluer vers une opérande concrète.

Pour cette extension, il faudra ajouter, au minimum, les constructions suivantes :

- `$x` : syntaxe pour écrire une opérande de type variable symbolique (qui sera immédiatement empilée, si RPN)

1. Sans valeur a priori, à la différence des variables proposées dans l'extension précédente.

– `subst` : substitue une expression s à une variable x dans une autre expression e (en RPN : il faut que le sommet de pile contienne la variable x, la valeur d'après l'expression s et celle encore après, l'expression e).

Exemple en RPN :

```
> $x
x

> $y
y

> +
(x + y)

> 3
3

> $x
x

> subst
(3 + y)

> 2
2

> $z
z

> *
(2 * z)

> $y
y

> subst
(3 + (2 * z))

> 5
5

> $z
z

> subst
13
```

Encore quelques remarques :

– Si vous faites aussi l'extension 3, prenez bien conscience de la différence entre `!x/?x` et `$x`. Dans le premier cas, `x` est le nom d'une case mémoire, l'entrée s'évalue comme le contenu de cette case. Dans le deuxième cas, la variable est symbolique : `x` est juste un nom qui n'a jamais de valeur et qui s'évalue vers... ce même nom, « `x` » ; en revanche, on peut demander à substituer ce nom par une autre expression.

– Les expressions seront automatiquement évaluées et remplacées par une valeur concrète dès lors qu'elles ne contiennent plus de variables symboliques.

– Les opérations définies pour les autres types sont automatiquement définis pour les expressions symboliques. Par exemple, les opérations « + » de l'exemple, bien qu'initialement définies pour les nombres uniquement, ont été appliquées à des variables et des expressions complexes. Le résultat est une expression encore plus complexe.

– Une conséquence, si votre calculatrice gère plusieurs types (extension 1), est qu'il est possible de constituer des opérations mal typées :

```
> $x
x

> 1
1

> +
(x + 1)

> "bonjour"
"bonjour"

> $x
x
```

```
> subst
Erreur: impossible d'évaluer ("bonjour" + 1) (expression mal typée)
```

Essayez de faire en sorte que cela soit impossible. Une piste serait que les variables soient typées lors de leur déclaration et que ce type soit vérifié au moment où on les intègres dans une expression. Une fausse piste serait de rendre les types `Variable` et `Operation` génériques (en effet, à cause de l'effacement de type, il sera impossible de récupérer la valeur du paramètre de type quand on dépile une opérande).

– Les nouvelles constructions permettent de définir de nouveaux objets mathématiques. Par exemple, tel est le cas des polynômes : pour ceux-ci, il serait intéressant de prévoir des opérations spécifiques (par exemple, pour un produit de polynômes, développer le résultat ; ou bien une opération pour factoriser un polynôme par un autre...).
Il y a probablement d'autres exemples d'expressions mathématiques pour lesquelles on peut prévoir des manipulations spécifiques (simplification de formules trigonométriques, dérivation des fonctions usuelles, ...)

VIII) Extension 5 : historique modifiable (« feuille de calcul »)

Dans cette extension, il s'agit de permettre de revenir sur ce qui a été entré dans l'historique, de façon à ce que tous les calculs dépendants soient réévalués par rapport à cette modification (comme dans un logiciel de tableur ; ce serait donc un tableur « 1D »).
Exemple :

Il y aurait plusieurs façons d'implémenter cette fonctionnalité (plus ou moins naïve, plus ou moins orientée objet). Privilégiez autant que possible les concepts objets et patrons de conception (notamment les variantes du patron observateur/observable).
Pour cette extension, il est conseillé que la calculatrice soit en mode graphique, afin que le panneau des calculs soit mis à jour sans qu'il ne reste aucune trace de l'ancienne version (en mode texte, on ne peut jamais réellement effacer l'historique des affichages).

IX) Extension 6 : Feuille de calcul partagée

Il s'agit maintenant d'ajouter un mode multi-utilisateur à la calculatrice, de telle sorte que plusieurs utilisateurs puissent agir simultanément sur la session en cours.
Cela peut être fait indépendamment de l'extension 5... mais c'est beaucoup plus intéressant quand celle-ci a été faite, car il faut gérer les modifications concurrentes de l'historique de façon cohérente.
Sinon, avec seulement l'extension 3, c'est intéressant aussi, si chaque utilisateur agit dans une REPL différente, mais que les valeurs peuvent être partagées d'un REPL à l'autre (un utilisateur stocke un résultat de sa REPL dans x, un autre rappelle le contenu de x dans sa propre REPL). Les REPL de tous les utilisateurs seraient visibles (au moins en lecture seule) pour tout le monde.
Enfin, dans tous les cas, pour des raisons d'utilisabilité, il serait très utile d'ajouter une zone de *chat* textuel (sans REPL) dans la fenêtre graphique (de sorte à ce qu'un utilisateur puisse expliquer à un autre ce qu'il est en train de faire).

Remarque : concrètement, on aimerait voir de la vraie concurrence, de la simultanéité. Pour mettre cette simultaneité en évidence, le mieux c'est d'implémenter la mise en réseau de plusieurs calculatrices : depuis la calculatrice, on peut « partager » la feuille en cours, après quoi, une autre calculatrice peut se connecter (en utilisant l'adresse IP de la première calculatrice) et voir cette feuille et ses évolutions.

X) Extension 7 : graphes de fonctions

Si l'extension 5 a été réalisée, on peut alors voir toute expression à valeurs dans les nombres et dépendant d'une seule variable numérique comme une fonction numérique, qui peut être tracée dans un graphe.
Ajouter le tracé de graphe à la calculatrice pourrait alors être très intéressant (je sais que cela peut être fait assez facilement en JavaFX, mais d'autres options techniques sont envisageables).