Interface, exceptions

Exercice 1 - Evaluateur d'expression

Nous avons lors d'une séance précédente représenté les expressions arithmétiques par leur **arbre syntaxique**. Nous avons pour ça défini des classes dont les instances représentent des opérateurs ou des constantes, qui implantent toutes une interface `Expr`, qui déclare une méthode `public double eval()`.
Utiliser la fonctionnalité `import`, pas de copier/coller.
Nous allons maintenant créer des analyseurs syntaxiques dont le rôle est de calculer un tel arbre à partir d'une chaîne de caractères.
Vous documenterez toutes les classes, interfaces et méthodes que vous écrirez en utilisant `javadoc`.
Toutes les classes de l'analyseur (et ses dépendances) devront être créées dans le paquetage `fr.umlv.analyzer`.
En fait, on souhaite créer deux analyseurs syntaxiques différents, un analyseur préfixe et un analyseur postfixe.
Nous allons donc dans un premier temps essayer de définir un interface (un type) permettant de manipuler indifféremment un analyseur préfixe ou un analyseur postfixe.

1 Écrire une classe d'exception `SyntaxException` qui représente les erreurs syntaxique qui pourront être éventuellement détectée par les analyseurs syntaxiques que nous allons écrire.
2 Écrire une interface `SyntacticAnalyzer` représentant le type des analyseurs syntaxiques. Cette interface doit déclarer une méthode `analyze` prenant en paramètre un itérateur sur chaine (`Iterator<String>`) et levant une exception `SyntaxException` s'il y a une erreur durant l'analyse.

Exercice 2 - Analyseur préfixe

On cherche à construire un évaluateur préfixe respectant l'interface. `SyntacticAnalyzer`.

- Détaillez le fonctionnement de l'analyse préfixe de `* + 1 2 - 3 4`, en construisant au fur et à mesure des opérations l'arbre syntaxique de l'expression.
  Chaque chaîne de caractère est analysée pour savoir s'il s'agit d'un opérateur où d'une valeur. L'expression est construite par un parcours récursif, puis évaluée.
- Écrire une classe `PrefixAnalyzer` dont la méthode `analyze` prend des expressions en notation préfixe et implantant l'interface `SyntacticAnalyzer`.
- Ecrire le code de la classe `Evaluator` qui permet de tester l'évaluateur préfixe.

```
Java fr.umlv.analyzer.Evaluator "+" "7" "*" "3" "2"
=13
```

Exercice 3 - Calculette

Modifier la classe `Evaluator` pour que si des arguments sur la ligne de commandes sont présents alors l'analyse s'effectue sur la ligne de commande sinon que l'analyse s'effectue

sur l'entrée standard.

Exercice 4 - Analyseur postfixe

On cherche maintenant à construire l'évaluateur postfixe respectant lui-aussi l'interface. `SyntacticAnalyzer`.

- Détaillez le fonctionnement de l'analyse postfixe de `1 2 + 3 4 * /`, en construisant au fur et à mesure des opérations l'arbre syntaxique de l'expression. Pour faire cela, vous devrez surement utiliser une strcuture de donnée classique. Laquelle est-ce ?
- Écrire une interface pour cette structure de donnée. Vous aurez le soucis d'être le plus général possible dans les profils des méthodes qu'elle déclare.
- Écrire une classe implantant l'interface de la structure de donnée en stockant les éléments dans un conteneur de type `ArrayList`. Donnez la complexité de chacune des méthodes de la classe que vous venez d'écrire.
- Écrire une classe `PostfixAnalyzer` implantant l'interface `SyntacticAnalyzer`.
- Changer la méthode `main` pour utiliser un analyseur préfixe ou postfix suivant l'option de la ligne de commande et respectant la syntaxe ci-dessous :

```
java fr.umlv.analyzer.Evaluator
(-prefix|-postfix) expression
```