concevoir des tests. Comment vérifier expérimentalement que l'implémentation d'une structure de données ou un algorithme a bien la complexité temporelle théorique attendue ?

Les réponses aux questions et les parties d'implémentation individuelles doivent être soumises sur INGInious ***avant*** la séance ***intermédiaire***[a]. Cela suppose une étude individuelle et une mise en commun en groupe (sans tuteur) préalablement à cette séance (l'utilisation du repository git est fortement conseillée). Un document (au format PDF) reprenant les réponses aux questions devra être soumis sur INGInious ***au plus tard*** pour le vendredi 22 septembre à ***8h00AM***. Les réponses seront discutées en groupe avec le tuteur durant la séance intermédiaire. Ces réponses ne doivent pas explicitement faire partie des produits remis en fin de mission. Néanmoins, si certains éléments de réponse sont essentiels à la justification des choix d'implémentation et à l'analyse des résultats du programme, ils seront brièvement rappelés dans le *rapport de programme*.

*a*. à l'exception, le cas échéant, de question(s) spécifiquement liée(s) au problème traité.

# 9 Problème

Vous êtes en charge de concevoir et d'implémenter en Java un interpréteur du mini langage `PostScript` décrit ci-dessous.

## 9.1 Le langage PostScript

Dans le langage `PostScript`, le programme suivant calcule $1 + 1$ :

```
1 1 add pstack
```

Chaque élément est un *token* numérique, symbolique ou booléen. Dans l'exemple ci-dessus, deux tokens numériques **`1 1`** sont suivis de deux tokens symboliques **`add pstack`**. L'évaluation de ce programme se passe comme suit. Chaque token numérique est empilé sur la pile du langage. Le token **`add`** dépile deux opérandes de la pile, calcule leur somme et empile le résultat. Le token **`pstack`** provoque l'impression du contenu de *toute* la pile.Lors de l'exécution, une pile peut contenir des identifiants (**`/pi`** par exemple), des entiers, des flottants ou des booléens.

Les opérations suivantes doivent être implémentées : **`pstack`**, **`add`**, **`sub`**, **`mul`**, **`div`**, **`idiv`**, **`dup`**, **`exch`**, **`pop`**, **`true`**, **`false`**.

### 9.1.1 Les opérations de pile

| Avant | Après | Description |
|---|---|---|
| $any$ **pstack** | $any$ | Imprime la pile sur la sortie. Le premier élément de la pile est le premier élément imprimé. Les éléments de la pile sont séparés par des espaces. |
| $any_1$ $any_2$ **exch** | $any_2$ $any_1$ | Echange les deux derniers éléments de la pile. |
| $any$ **pop** | $-$ | Supprime le dernier élément de la pile. |
| $any$ **dup** | $any$ $any$ | Duplique le dernier élément de la pile. |

Pour l'impression à l'écran, le booléen *true*, *false*, est imprimé `true`, `false` respectivement. Par exemple,

— **1 2 pstack** imprime **1 2**,
— **1.1234 pstack** imprime **1.1234**,

Les autres opérations servent à manipuler directement la pile. Par exemple,

— **3 4 exch pstack** imprime **4 3**,
— **2.0 0 1 pop pstack** imprime **2.0 0**,
— **1 dup pstack** imprime **1 1**,

### 9.1.2 Les opérations arithmétiques

| | |
|---|---|
| **add** | Retire les deux derniers éléments de la pile, calcule la somme et ajoute le résultat sur la pile. |
| **sub** | Retire les deux derniers éléments de la pile, calcule la différence du second par le premier et ajoute le résultat sur la pile. |
| **mul** | Retire les deux derniers éléments de la pile, calcule le produit et ajoute le résultat sur la pile. |
| **div** | Retire les deux derniers éléments de la pile, calcule la division en nombre flottant du second par le premier et ajoute le résultat sur la pile. |
| **idiv** | Retire les deux derniers éléments de la pile, calcule la division en nombre entier du second par le premier et ajoute le résultat sur la pile. |

Les opérations **add**, **sub**, **mul** retourne un entier si les deux opérandes sont des entiers, dans le cas contraire, elles retournent des flottants. La précision des flottants est celle des Double de Java. Par exemple

— **1 1 add pstack** imprime **2**,
— **1.0 2.5 sub pstack** imprime **-1.5**,
— **1 4.0 mul pstack** imprime **4.0**,
— **200 5 div pstack** imprime **40.0**,
— **5 2 idiv pstack** imprime **2**.

Les deux programmes suivant calculent et impriment respectivement $1 + (3 * 4)$ et $(1 + 3) * 4$, sans que des parenthèses soient nécessaires (pouvez-vous dire pourquoi ?) :

```
1 3 4 mul add pstack
1 3 add 4 mul pstack
```

### 9.1.3 Les erreurs

Si un nombre insuffisant d'argument sont disponible sur la pile, le programme lance l'exception **`EmptyStackException`**.

Si une opération arithmétique n'est pas valide (lors d'une division par zéro par exemple), l'exception **`ArithmeticException`** est lancée.

Si tous les arguments ne sont pas valide (par exemple, un entier est attendu et un booléen est trouvé), l'exception **`IllegalArgumentException`** est lancée.

## 9.2 Analyse, conception et production d'une solution

Concrètement, vous devrez créer une classe `Interpreter` avec une méthode `interpret` qui prendra en entrée un `String` contenant les instructions (sur une seule ligne) et qui retournera un nouveau `String` contenant l'état de la stack lors de l'instruction `pstack`.

```
public interface PostScriptInterpreter {
    public void interpret(String instructions, PrintStream out);
}
```

Par exemple, lors de l'appel suivant : `myInterpreter.interpret("3 3 idiv pstack", System.out)`, votre méthode doit imprimer à la sortie standard `1`.

Ce problème sera divisé en trois étapes :

1. Création de tests unitaires avec JUnit pour tester le bon fonctionnement de votre stack et de votre interpréteur (tâche individuelle, à faire **avant** la phase d'implémentation) ;
2. Implémentation d'une stack générique (tâche individuelle) par une ***liste simplement chaînée***.
3. Implémentation de l'interpréteur de PostScript (par groupes).

Il serait particulièrement utile de vous poser un ensemble de questions ***avant*** d'aborder le travail de conception et de programmation dans ce problème. Parmi celles-ci, on peut citer :

— Quelles sont les fonctionnalités du programme demandé ? Comment répartir ces fonctionnalités dans différentes parties de votre programme et comment répartir le travail de programmation entre vous ?

— Quelles sont les fonctionnalités du programme demandé qui ne dépendent pas spécifiquement du problème traité ? Comment isoler ces fonctionnalités de manière à ce qu'elles soient réutilisables dans un autre programme résolvant un autre problème ?

— Que faudrait-il faire si l'on désirait changer d'implémentation de pile dans votre interpréteur ? Comment garantir que ce changement nécessite le moins de modifications possibles dans votre programme ? Quel est l'impact sur la complexité des opérateurs ?

Veillez à réfléchir sur la complexité algorithmique des opérateurs que vous devez implémenter. Certains opérateurs peuvent être implémentés de manière plus efficace qu'une première version naïve.

# 10 Indications méthodologiques

- Veillez à répondre aux questions de manière ***précise*** et ***concise***. Citez vos sources.
- Ne vous précipitez pas sur la résolution du problème mais veillez à répondre aux questions avant les phases d'analyse et de conception.
- Écrivez les tests unitaires avant de commencer à programmer.
- Demandez-vous comment un **diagramme de classes** peut aider à répartir efficacement la tâche de programmation entre vous.
- Demandez-vous comment la réponse aux questions et votre analyse initiale du problème peut influencer le travail de programmation.
- Demandez-vous comment répartir le travail de programmation pour que tous les membres du groupe aient l'occasion de perfectionner leur maîtrise de Java pendant ce quadrimestre.