Laboratoire 6 – Piles

ITI 1521. Introduction à l'informatique II

Semaine 8-12 Mars 2021

Dû en ligne après une semaine de votre Lab

/10

- Objectifs

• Implémentation d'une pile à l'aide d'un tableau
• Classe abstraite
• Appliquer le concept d'héritage pour l'implémentation de classes

I. Première partie – Piles

- Introduction

Cette partie du laboratoire porte sur l'implémentation de l'interface **Stack** à l'aide d'un tableau, ainsi qu'une application des piles.

Question 1 : (3 POINTS : 0.4 POINTS pour chaque méthode et 0.6 pour le programme Test)

Considérant l'interface `Stack` suivante :

```
public interface Stack<E> {
boolean isEmpty ( ) ;
E peek ( ) ;
E pop ( ) ;
void push (E item ) ;
void clear (E item ) ;
}
```

On désire définir la classe **`ArrayStack`** qui implémente l'interface **`Stack`**.
`ArrayStack` utilise un tableau de taille fixe pour stocker ses éléments et un attribut *top* qui désigne la première cellule libre de son tableau.
`ArrayStack` a donc un constructeur **`à un paramètre.`**
Puisque la classe **`ArrayStack`** réalise l'interface **`Stack`**, elle doit fournir une implémentation pour toutes les méthodes de l'interface. Ainsi, vous devez définir ces méthodes ainsi que le constructeur:
ArrayStack(int size) : constructeur
*isEmpty : renvoie true si la pile (**`ArrayStack`**) est vide*
E peek ( ) : Renvoie l'élément supérieur de la pile sans le supprimer
E pop ( ) : Supprime et renvoie l'élément supérieur de cette pile
void push (E item) : Place l'élément sur le dessus de cette pile
void clear (E item) : retire tous les éléments de cette pile. La pile sera vide suite à cet appel.

Écrire un programme `Test` pour tester votre pile :

- *Créer un objet de votre pile* ***`ArrayStack`*** *de type actuel String.*
- *Empiler votre pile de 10 éléments.*
- *Vider complètement votre pile*
- *Empiler à nouveau votre pile de 10 éléments.*
- *Dépiler votre pile tout en affichant l'élément retiré.*

/Exemple de sortie*/

Second time :18
Second time :16
Second time :14
Second time :12
Second time :10
Second time :8
Second time :6
Second time :4
Second time :2
Second time :0

Question 2 : (1 POINT)

Pour cette partie du laboratoire, il y a deux algorithmes pour la validation d'expressions contenant des parenthèses (les parenthèses, les accolades et les crochets).
La classe **Test** présente un algorithme simple (ci-bas) pour valider des expressions. On remarque qu'une expression bien formée est telle que pour chaque type de parenthèses (les parenthèses, les accolades et les crochets), le nombre de parenthèses ouvrantes est égal au nombre de parenthèses fermantes. D'où l'algorithme suivant :

```
public static boolean algorithm1 ( String str ) {
    int brace , square , bow ;
    brace = square = bow = 0 ;
    for ( int i =0; i<str.length ( ) ; i++) {
      char c ;
      c = str.charAt ( i ) ;
      switch ( c ) {
        case '{' :
                  brace ++;
                  break ;
        case '}' :
                   brace --;
                   break ;
        case '[' :
                  square++;
                   break ;
         case ']' :
                  square--;
                   break ;
```

```
        case '(' :
                bow ++;
                 break ;
         case ')' :
                bow --;
      }
    }
    return brace == 0 && square == 0 && bow == 0 ;
}
```

Compilez ce programme et expérimentez. D'abord, assurez-vous qu'il fonctionne pour des expressions valides, telles que " ()[]()", " ([][()])". Vous remarquerez que l'algorithme fonctionne aussi pour des expressions qui contiennent des opérandes et des opérateurs : " (14 * (47 - 2)) ".
Ensuite, vous devez trouver des expressions pour lesquelles l'algorithme retourne **true** bien que ces expressions ne sont pas bien formées.
Écrire un programme `Test` pour tester.

/Exemple de sortie*/
algorithm1("([][()])") returns true

Question 3 : (2 POINTS : POINT pour *algorithm2* et un pour *main*)

Vous avez trouvé des expressions qui font échec à cet algorithme. En effet, une expression bien formée est une expression telle que le nombre de parenthèses ouvrantes et fermantes est le même, et ce, pour chaque type de parenthèses. Mais aussi, lorsqu'on lit une telle expression de gauche à droite et que l'on rencontre une parenthèse fermante alors son type doit être le même que celui de la dernière parenthèse ouvrante rencontrée qui n'a pas encore été traitée (associée).
Vous devez implémenter un algorithme à base de pile afin de valider des expressions : retourne **true** si l'expression est bien formée et **false** sinon. De plus, l'analyse ne devrait parcourir la chaîne qu'une seule fois.
Vous devez créer votre implémentation dans la classe **`Test`**, nommez cette méthode **algorithm2**. Ci-joint l'algorithme que vous devez compléter :

```
public static boolean algorithm2(String str ) {
      Stack<Character> myStack;
       myStack = new ArrayStack<Character>( 100 );

     for ( int i=0; i<str.length(); i++ ) {
        char current = str.charAt( i );
        if ( current == '(' || current == '[' || current == '{' ) {
           myStack.push( new Character( current ) );
        }
             votre code vient ici

  }
```

Faites plusieurs tests à l'aide d'expressions valides et non valides. Assurez-vous que votre algorithme traite ce cas-ci : *"(())"*.
Écrire un programme `Test` pour tester.
/Exemple de sortie*/
algorithm2("()[]() ") returns true
*algorithm2("(14 * (47 - 2))) ") returns false*
algorithm2("(())") returns false

II. Deuxième partie – Héritage et classe abstraite

Vous allez implémenter une hiérarchie de classes afin de représenter des codes postaux de divers pays, sachant que :
• Tous les codes postaux ont une méthode *getCode* retournant le code (de type *String*) représenté par cette instance (*code*) ;
• Tous les codes postaux ont une méthode *isValid* retournant *true* si le code de cette instance est valide, et *false* sinon ;
• Un code postal canadien est valide si les positions 0, 2 et 5 sont occupées par des lettres, les positions 1, 4 et 6 sont des chiffres, et la position 3 est un caractère blanc ;
• Un code postal américain (Zip code) valide est constitué de deux lettres, suivies d'un espace blanc, suivi de 5 chiffres.

Question 4 : (4 POINTS : 1 POINT pour chaque classe)

Concevoir une implémentation pour les classes `PostalCode` (classe mère **abstraite**, la méthode *isValid* est abstraite), `CanadaCode` et `USCode` qui sont les sous classes concrètes de `PostalCode`. Assurez vous d'y include les variables d'instance ainsi que les constructeurs. L'appendice ci-bas présente un résumé des méthodes des classes `String` et `Character`.

Créez une classe `Test` pour tester vos classes :
• Déclarez un tableau, nommé *codes*, pouvant contenir 100 codes postaux ;
• Créez $n = 10$ codes postaux dont certains sont des codes américains alors que d'autres seront des codes canadiens. De même, certains codes sont valides et d'autres pas. Finalement, sauvegardez ces codes dans les *n* premières cellules de gauche du tableau ;
• Sachant qu'exactement n codes postaux se trouvent dans la partie gauche du tableau, écrivez une boucle *for* afin de compter le nombre de codes valides et l'afficher.

/Exemple de sortie*/
2 codes sont valides

Appendix

La classe **`String`** contient les méthodes suivantes.
char charAt(int pos) **:** retourne le caractère se trouvant à la position spécifiée par l'index pos ;
int length() **:** retourne la longueur de cette chaîne.

La classe **`Character`** contient les méthodes suivantes :
***static boolean isDigit(char ch)* :** détermine si le caractère passé en paramètre est un nombre;
***static boolean isLetter(char ch)* :** détermine si le caractère passé en paramètre est une lettre;
***static boolean isWhitespace(char ch)* :** détermine si le caractère passé en paramètre est un espace blanc.

Créer et soumettre un fichier zip comme d'habitude (Q1,…,Q4)