NFP121 : Programmation avancée
TP n° 3 - Les piles

Une pile d'entiers primitifs

Soit la classe Pile d'entiers ci-dessous

```
public class Pile {

  public final static int TAILLE_PAR_DEFAUT = 5;

  private int[] zone;
  private int ptr;

  public Pile(int taille) {
    if (taille < 0)
      taille = TAILLE_PAR_DEFAUT;
    this.zone = new int[taille];
    this.ptr = 0;
  }

  public Pile() {
    this(TAILLE_PAR_DEFAUT);
  }

  public void empiler(int i) throws PilePleineException {
    if (estPleine())
      throw new PilePleineException();
    this.zone[this.ptr] = i;
    this.ptr++;
  }

  public int depiler() throws PileVideException {
    if (estVide())
      throw new PileVideException();
    this.ptr--;
    return zone[ptr];
  }

  public boolean estVide() {
    return ptr == 0;
  }

  public boolean estPleine() {
    return ptr == zone.length;
  }

  @Override
  public String toString() {
    StringBuffer sb = new StringBuffer("[");
    for (int i = ptr - 1; i >= 0; i--) {
      sb.append(Integer.toString(zone[i]));
      if (i > 0)
        sb.append(", ");
    }
    sb.append("]");
    return sb.toString();
  }
}
```

Une pile d'"Object"s

Les éléments de la classe Pile sont maintenant des instances de la classe java.lang.Object (classe racine de toute classe Java).

Donc **private Object**[] zone; remplace **private int**[] zone;

Modifiez le code de la classe pile d'entiers pour obtenir une nouvelle version de la classe Pile et de la classe ApplettePile

un exemple de la "nouvelle" ApplettePile *(Essayez d'empiler des nombres, des mots, ...)*

Développez à nouveau une classe "UneUtilisation"

Soit la classe UneUtilisation ci-dessous

```
public class UneUtilisation {

  public static void main(String[] args) throws Exception {
    Pile p1 = new Pile(6);
    Pile p2 = new Pile(10);
    // p1 est ici une pile de polygones réguliers PolygoneRegulier.java,
    // entre autres
    p1.empiler(new PolygoneRegulier(4, 100));
    p1.empiler(new PolygoneRegulier(5, 100));
    p1.empiler("polygone");
    p1.empiler(new Integer(100));
    System.out.println(" la pile p1 = " + p1);

    p2.empiler(new Integer(1000));
    p1.empiler(p2);
    System.out.println(" la p1 = " + p1);

    try {
      p1.empiler(new PolygoneRegulier(4, 100));
      // ....
      String s = (String) p1.depiler(); // vérifiez qu'une exception est
                        // levée à l'exécution
    } catch (Exception e) {
      e.printStackTrace();
    }
  }
}
```

vérifiez l'affichage produit et l'exception résultante.
Soumettez cette première question à l'outil d'évaluation.

Spécification : Interface

Soit l'interface spécifiant le comportement d'une pile (fichier PileI) :

```
package question2;

import question1.PilePleineException;
import question1.PileVideException;

public interface PileI {

  public final static int CAPACITE_PAR_DEFAUT = 6;

  public void empiler(Object o) throws PilePleineException;
  public Object depiler() throws PileVideException;

  public Object sommet() throws PileVideException;
  public int capacite();
  public int taille();
  public boolean estVide();
  public boolean estPleine();
  public boolean equals(Object o);
  public int hashCode();
  public String toString();

}
```

Modifiez la classe Pile pour qu'elle implémente maintenant l'interface PileI

Attention : *5 nouvelles méthodes sont spécifiées dans PileI (capacité, taille, sommet, equals, et hashCode), et donc toute implémentation de cette interface doit implanter aussi ces 5 méthodes.*

Notes :

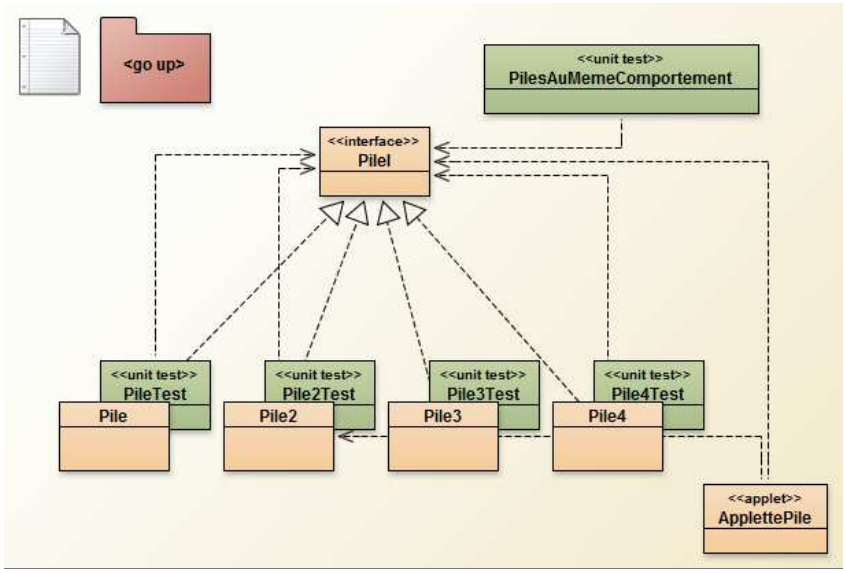
La méthode *sommet* retourne le sommet de la pile (sans dépiler)
La méthode *equals* teste l'égalité de deux Piles : Deux piles sont égales si elles ont la même taille, même capacité, et les éléments contenus identiques.
La méthode *capacité* indique le nombre maximal d'éléments que l'on peut empiler.
La méthode *taille* retourne le nombre d'éléments présents dans cette pile.
La méthode *hashCode* est fournie, et cette note intitulée "How to avoid traps and correctly override methods from java.lang.Object" peut vous être utile pour comprendre sa présence ainsi que la javadoc de la classe Object

Implémenter une interface

PLUSIEURS implémentations UN SEUL comportement :

Proposez, 4 autres implémentations des Piles (donc implémentant l'interface PileI), attention, pour l'utilisateur ici les piles doivent toujours être bornées , elles ne peuvent avoir un comportement différent...

- Pile : utilise la Pile de la question 1,
- Pile2 : utilise la classe prédéfinie java.util.Stack‹object›,
- Pile3 : utilise la classe prédéfinie java.util.Vector‹object›,
- Pile4 : utilise une liste chaînée créée en interne



Remarques :

Pile2 et Pile3 seront composées d'une instance d'une classe prédéfinie comme le suggère les extraits de code suivants :

```
import java.util.Stack;
public class Pile2 implements PileI {
  private Stack<Object> stk; // La classe Pile2 est implémentée par délégation
  // ...
}

import java.util.Vector;
public class Pile3 implements PileI {
  private Vector<Object> v;
  // ...
}
```

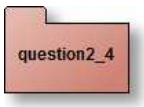
notes :

La méthode equals teste l'égalité de deux Piles ... de capacité et contenu identiques

Ci dessous "ApplettePile" avec une implémentation de la classe Pile par délégation à la classe java.util.Stack<Object>

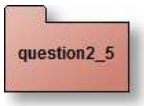
Les classes de tests unitaires

Proposez une classe de tests unitaires à chaque implémentation.



Tests Unitaires "PilesAuMemeComportement"

Complétez la classe de Tests Unitaires nommée "PilesAuMemeComportement" , qui "vérifie" que toutes les piles ont bien le même comportement.



Le couple <equals, hashCode>

Que pensez vous de cette réponse pour les classes Pile, Pile2, Pile3 et Pile4 ?

```
  public boolean equals(Object o) {
    if (o instanceof PileI) {
      PileI p = (PileI) o;
      return this.capacite() == p.capacite()
          && this.hashCode() == p.hashCode();
    } else
      return false;
  }
```

Est-elle correcte ?, quelle est la règle à retenir ? (à toutes fins utiles la javadoc de java.lang.Object)

Cette autre réponse, élégante, n'est pas correcte

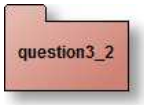
```
  public boolean equals(Object o) {
     return this.toString().equals(o.toString());
    }
```

pourquoi ?

Généricité : un premier usage

L'interface pileI est désormais paramétrée par le type des éléments.

```
package question3;

import question1.PilePleineException;
import question1.PileVideException;

public interface PileI<T> {

  public final static int CAPACITE_PAR_DEFAUT = 6;

  public void empiler(T o) throws PilePleineException;
  public T depiler() throws PileVideException;
  public T sommet() throws PileVideException;
  // ...

} // PileI
```



La classe Pile2<T> et l'IHM ApplettePile

Vérifiez le source de la classe Pile2<T> et modifier l'IHM ApplettePile, ce type de message obtenu à la compilation ne doit plus apparaître

La classe UneUtilisation

Dans la méthode main de la classe UneUtilisation, proposez toutes les déclarations correctes afin que la compilation de cette classe s'effectue sans aucun message d'erreur ou alertes

i.e. proposez les déclarations telles que p1 contienne des éléments "de type PolygoneRegulier", p2 que des Piles de Polygone régulier , etc ...

```
import question1.PolygoneRegulier;

public class UneUtilisation {

  public static void main(String[] args) throws Exception {
    PileI ... p1 = new Pile2 ... (10); // p1 ne contient que des polygones réguliers
    PileI ... p2 = new Pile2 ... (10); // p2 ne contient que des piles de polygones réguliers

    etc ...
```

Vérifiez ensuite que ces lignes extraites de la question 1 ne se compilent plus !

```
    try {
      p1.empiler(new polygoneRegulier(5,10)); // vérifiez qu'une exception est levée à la compilation

      String s = (String)p1.dépiler();
    } catch(Exception e ) {
      e.printStackTrace();
    }
```
