Université de Montpellier / Faculté Des Sciences

Assertions, Exceptions et mise en œuvre du type abstrait `Pile`

Structures de données — HMIN 215

8 février 2021

1 Objectifs du cours

Java possède deux principales techniques de mise en place de contrats pour les classes : les assertions et les exceptions. Nous les étudions dans ce cours au travers de l'étude du type abstrait Pile.

Le plan du cours est le suivant :

— Le type abstrait *Pile* et une structure de données qui le met en œuvre,
— Les assertions qui servent à la mise au point du type abstrait.
— Les exceptions qui servent à sécuriser son usage.

2 Le type abstrait *Pile*

Le type abstrait *Pile* se définit ainsi.

— le type défini est *Pile*
— il est paramétré par un autre type T

Opérations Les opérations autorisées sur une pile sont :

— initialiser()
— empiler(T element)
— depiler() : T
— sommet() : T
— estVide() : boolean

Préconditions On peut a priori toujours initialiser, empiler et savoir si la pile est vide. Par contre on ne peut dépiler ou consulter le sommet d'une pile que si elle est non vide. Donc si p est une pile :

— précondition (p.depiler()) : $\neg\ p.estVide()$
— précondition (p.sommet()) : $\neg\ p.estVide()$

Axiomes si p est une pile, et t un élément de type T, les axiomes indiquent le fonctionnement des opérations.

1. Juste après l'instruction $p.empiler(t)$, on a $p.sommet() = t$
2. Juste après $p.initialiser()$, on a $p.estVide() = true$

3. Juste après $p.empiler(t)$, on a $p.estVide() = faux$
4. Juste après $p.empiler(t)$, on a $p.depiler() = t$

Nous proposons de représenter le type abstrait *Pile* par une interface.

```
public interface IPile<T>
{
   void initialiser();
   void empiler(T t);
   T depiler();
   T sommet();
   boolean estVide();
   // et nous ajoutons une méthode pratique
   int nbElements();
}
```

Puis cette interface est implémentée en Java par une classe. Cette classe contient une fonction *main* qui illustre le fonctionnement de la pile.

```
public class Pile<T> implements IPile<T>{
   private ArrayList<T> elements;

   public Pile(){initialiser();}

   public T depiler() {
     T sommet = elements.get(elements.size()-1);
     elements.remove(sommet);
     return sommet;
   }

   public void empiler(T t) {elements.add(t);}

   public boolean estVide() {return elements.isEmpty();}

   public void initialiser() {elements = new ArrayList<T>();}

   public T sommet() {return elements.get(elements.size()-1);}

   public String toString(){return "Pile = "+ elements;}

   public int nbElements(){return elements.size();}

   public static void main(String[] a)
   {
      Pile<String> p = new Pile<String>();
      System.out.println(p);
      p.empiler("a"); p.empiler("b"); p.empiler("c");
      System.out.println(p);
      p.depiler();
      System.out.println(p);
```

```
      p.depiler(); p.depiler();
      System.out.println(p);
    }
}
```

3 Assertions

Les assertions permettent de vérifier des propriétés sur une classe. Le programme est interrompu en cas de non respect de ces propriétés. Elles peuvent être activées ou inhibées et sont principalement une aide au débogage et vont servir à vérifier la plupart des axiomes (post-conditions). Pour les activer (resp. les désactiver), il faut exécuter l'interprète *java* avec l'option $-ea$ (resp. $-da$). Sous eclipse, il faut changer les arguments de la configuration d'exécution, comme le montre la figure 1.

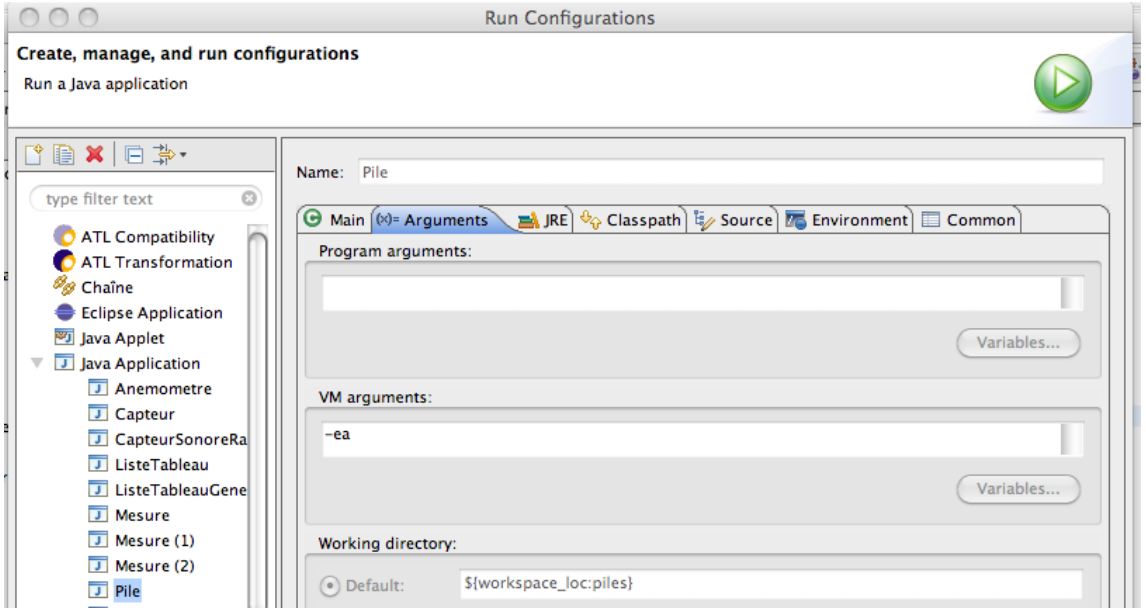

FIGURE 1 – Configuration sous eclipse pour activer les assertions

Deux syntaxes sont possibles :

```
assert condition ;
assert condition : objet ;
```

Par exemple, pour vérifier le premier axiome qui indique qu'après avoir empilé un élément t dans une pile, cet élément apparaît au sommet, on peut écrire l'assertion en fin de la méthode.

```
public void empiler(T t) {
   elements.add(t);
   assert this.sommet()==t;
}
```

Supposons que l'on écrive une version erronée de la méthode *empiler*, telle que l'ajout de *null* à la fin du conteneur des éléments de la pile :

```
public void empiler(T t) {
   elements.add(t); elements.add(null);
   assert this.sommet()==t ;
}
```

L'exécution du *main* précédent s'arrêtera et vous obtiendrez le message suivant :

```
Exception in thread "main" java.lang.AssertionError
at pile.Pile.empiler(Pile.java:20)
at pile.Pile.main(Pile.java:48)
```

Si l'on veut être plus précis, on peut mettre un objet dans l'assertion, l'erreur affichée mentionnera le résultat de l'appel de *toString*() sur cet objet.

```
public void empiler(T t) {
   elements.add(t);
   assert this.sommet()==t : "dernier empile ="+this.sommet();
}
```

Cette fois l'erreur affichée sera la suivante.

```
Exception in thread "main" java.lang.AssertionError: dernier empile =null
at pile.Pile.empiler(Pile.java:20)
at pile.Pile.main(Pile.java:48)
```

4 Exceptions

On utilisera le mécanisme de gestion des exceptions pour signaler mais aussi pour rattraper des erreurs pouvant se produire pendant l'exécution du programme. A l'inverse du mécanisme d'assertions qui sert principalement à la mise au point, le mécanisme de gestion des exceptions sert à éviter des arrêts brutaux du programme, et à revenir dans un fonctionnement normal, ou à stopper le programme proprement, après avoir éventuellement sauvegardé des données, fermé des fichiers, etc.

En Java, les exceptions sont des objets qui représentent une erreur à l'exécution :

- une méthode appelée hors de son domaine de définition, par exemple dépiler une pile vide, ou donner une mauvaise valeur de paramètre, cela correspond souvent aux préconditions des opérations des types abstraits,
- une erreur de saisie (comme saisir "aa" quand on attend un entier, ou une date mal formée),
- un problème matériel (plus de place lors de l'écriture d'un fichier, ce qui empêchera le respect d'une post-condition du fait de l'environnement d'exécution),
- certains événements de l'exécution qui vont se produire car le programme est mal prouvé ou mal testé (sortir des bornes d'un tableau, une division par zéro).

Définition d'une classe d'erreur sur une pile Dans le cas de la classe pile telle que nous l'avons esquissée, une erreur peut ainsi se produire lors de l'appel des opérations *dépiler* ou *sommet* sur une pile vide. Cette erreur va se représenter à l'aide d'une instance de la classe `PileVideException` que nous définissons. Cette classe spécialise la classe de l'API Java `Exception`. C'est nécessaire pour bénéficier du mécanisme d'exceptions. On note que cette classe n'est pas très compliquée dans le cas présent, elle pourrait avoir en plus des attributs décrivant la localisation de l'erreur ou l'état des objets au moment où l'erreur s'est produite, et des opérations de traitement de l'erreur.

```
public class PileVideException extends Exception {
    public PileVideException() {  }
    public PileVideException(String message) { super(message); }
}
```

Déclaration et signalement d'une erreur Cette classe d'exception nous permet de réécrire une version plus sécurisée de la méthode *dépiler*. Il faut noter dans la signature (première ligne) la **déclaration** du fait que la méthode risque de s'interrompre à cause de l'erreur $PileVideException$ et le **signalement** effectué dans le code lorsque l'on détecte que l'on essaie de dépiler une pile vide.

```
public T depiler() throws PileVideException{
    if (this.estVide())
        throw new PileVideException("en dépilant");
    T sommet = elements.get(elements.size()-1);
    elements.remove(sommet);
    return sommet;
}
```

Notez que cela oblige à prévoir le signalement dans l'interface également qui sera modifiée en conséquence.

```
public interface IPile<T>
{
....
   T depiler() throws PileVideException;
...
}
```

Récupération d'une erreur Maintenant le programme suivant va tout de même s'interrompre brutalement avec une erreur et n'atteindra pas la fin.

```
public static void main(String[] a)
{
    Pile<String> p = new Pile<String>();
    System.out.println(p);
    p.empiler("a"); p.empiler("b"); p.empiler("c");
    System.out.println(p);
    p.depiler();
    System.out.println(p);
    p.depiler(); p.depiler();  p.depiler();
    System.out.println(p);
}
```

On utilise alors un bloc de contrôle particulier, le bloc

```
try{BLOC-0}
catch (Class-1 e1){BLOC-1}
...
```

```
catch(Class-n en){BLOC-n}
finally{BLOC-n+1}
```

pour rattraper l'erreur, éventuellement la traiter, puis continuer l'exécution. Dans ce bloc `try / catch`, les `BLOC-i`, i entre 1 et $n+1$ sont des sous-blocs.

Lorsqu'une instance d'exception est générée dans un bloc *try*, l'exécution de `BLOC-0` s'interrompt. Les clauses *catch* sont examinées dans l'ordre, jusqu'à en trouver une (`Class-i`) qui déclare une classe à laquelle appartient l'instance d'exception. S'il en existe une, le bloc du *catch* (`BLOC-i`) est exécuté, et l'exécution reprend juste après les clauses *catch*. Sinon, l'exception n'est pas capturée, elle est donc transmise à l'appelant (une autre méthode et/ou un autre bloc try/catch), et le reste de la méthode n'est pas exécuté. On passe dans tous les cas par le bloc $finally$.

Le code ci-dessous vous montre une utilisation simple d'un tel bloc.

```
public static void main(String[] a)
{
     try{
         Pile<String> p = new Pile<String>();
         System.out.println(p);
         p.empiler("a"); p.empiler("b"); p.empiler("c");
         System.out.println(p);
         p.depiler();
         System.out.println(p);
         p.depiler(); p.depiler();  p.depiler();
         System.out.println(p);
    }
    catch (PileVideException p){System.out.println(p.getMessage());}
    System.out.println("Fin de la méthode main");
}
```

Nous examinons maintenant un cas plus complexe de l'utilisation de ce bloc de récupération des erreurs. Nous considérons 4 classes d'exception avec une relation d'héritage entre certaines de ces classes.

```
public class E1 extends Exception {}
public class E2 extends Exception {}
public class E3 extends E2 {}
public class E4 extends Exception {}
```

Nous allons nous préoccuper de ce qu'affiche le programme suivant suivant l'exception signalée dans meth1.

```
public class TestException {

    public void meth1() throws E1, E2, E3, E4
    {  ..........}

    public void meth2() throws E4
    {
        try{
```

```
            System.out.print("(1)");
            meth1();
        }
        catch(E1 e){System.out.print("(2)");}
        catch(E2 e){System.out.print("(3)");}
        finally {System.out.print("(4)");}
        System.out.println("(5)");
    }

    public static void main(String[] arg) throws E4
    {
        TestException t = new TestException();
        t.meth2();
    }
}
```

- s'il n'y a rien de signalé dans meth1 : (1)(4)(5)
- avec throw new E1() ; dans meth1 : (1)(2)(4)(5)
- avec throw new E2() ; dans meth1 : (1)(3)(4)(5)
- avec throw new E3() ; dans meth1 : (1)(3)(4)(5)
- avec throw new E4() ; dans meth1 : (1)(4)Exception in thread "main" exerciceCours2011.E4
  at exerciceCours2011.TestException.meth1(TestException.java :7)
  at exerciceCours2011.TestException.meth2(TestException.java :14)
  at exerciceCours2011.TestException.main(TestException.java :25)

Exceptions en Java Il existe plusieurs catégories d'exceptions dans l'API Java. Les deux classes *RunTimeException* et *Error* (et leurs sous-classes) correspondent à des erreurs que l'on n'a pas besoin de déclarer dans la signature des méthodes. Les *RunTimeException* sont nombreuses, vous en avez déjà rencontré plusieurs en programmant ($NullPointerException$, $ArrayIndexOutOfBoundsException$, $ClassCastException$, etc.). On peut chercher à les capturer pour sécuriser le programme. Les *Error* correspondent à des erreurs de la machine virtuelle sur lesquelles nous ne travaillerons pas dans le cadre de ce cours.

5 Synthèse

Il existe des bonnes et des mauvaises pratiques pour l'usage des exceptions et des assertions que nous détaillons ci-après. Il est conseillé d'utiliser les assertions pour exprimer les propriétés suivantes :

- invariant d'algorithme, par ex. à la fin de l'itération i dans la recherche du minimum la variable min contient la plus petite valeur rencontrée entre 0 et i
- la plupart des postconditions des opérations et des axiomes des types abstraits de données, par ex. à la fin d'une fonction de tri le tableau est trié
- invariants de classe, par ex. une voiture a 4 roues

Il est conseillé d'utiliser les exceptions pour :

- vérifier les paramètres d'une méthode car ces paramètres viennent d'ailleurs (de l'extérieur de la méthode) et peuvent vraisemblablement être faux (donc il vaut mieux ne pas interrompre brutalement l'exécution, et plutôt traiter le problème, avec des exceptions notamment),
- la plupart des préconditions des opérations,

— vérifier les résultats d'interactions avec un utilisateur (elles vont souvent comporter des erreurs, qu'il faut traiter),

Dans tous les cas, il faut absolument éviter pendant une vérification d'assertion de :

— créer des effets de bord, c'est-à-dire de modifier l'état du programme (par exemple ne pas empiler ou dépiler dans une assertion).