

Rappel de cours sur les Exceptions

The code in the **finally** block is executed under all circumstances, regardless of whether an exception occurs in the **try** block or is caught. Consider three possible cases :

If no exception arises in the **try** block, **finalStatements** is executed, and the next statement after the **try/catch/finally** statement is executed.

If one of the statements causes an exception in the **try** block that is caught in a **catch** block, the other statements in the **try** block are skipped, the **catch** block is executed, and the **finally** clause is executed. If the **catch** block does not rethrow an exception, the next statement after the **try** statement is executed. If it does, the exception is passed to the caller of this method.

If one of the statements causes an exception that is not caught in any **catch** block, the other statements in the **try** block are skipped, the **finally** clause is executed, and the exception is passed to the caller of this method.

The **finally** block executes even if there is a **return** statement prior to reaching the **finally** block.

```
1 try {
2     statements;
3 }
4 catch (TheException ex) {
5     handling ex;
6 }
7 finally {
8     finalStatements;
9 }
```

★ Exercice 1. On considère l'interface Stack et la classe Value suivantes :

```
1 package ex01;
2
3 public interface Stack {
4
5     public boolean isEmpty() ;
6
7     public void push(Value v) ;
8
9     public Value pop() throws EmptyStackException;
10
11     public Value peek() throws EmptyStackException;
12 }
```

```
1 package ex01;
2
3 public class Value {
4     private String name;
5     private int value;
6
7     public Value(String n, int v) {
8         this.name = n;
9         this.value = v;
10    }
11    public String toString() {
12        return "<"+this.name+";"+this.value+">";
13    }
14 }
```

- La classe **Value** décrit un couple <nom de la valeur, valeur entière stockée>.
- L'interface **Stack** décrit l'interface standard d'une pile de valeurs :
 - la méthode **boolean isEmpty()** indique si la pile est vide ou non,
 - la méthode **void push(Value v)** empile une nouvelle valeur,
 - la méthode **Value pop()** retourne et dépile une valeur empilée,
 - la méthode **Value peek()** retourne une valeur empilée, sans la dépiler.
- Les méthodes **Value pop()** et **Value peek()** ne peuvent pas retourner de valeur quand elles sont appelées sur une pile vide. Dans ce cas, une exception **EmptyStackException** doit être levée.

▷ **Question 1.** Écrire la classe **EmptyStackException** héritant de la classe **Exception**.

Réponse

```
1 package ex01;
2
3 // BEGINKILL
4 public class EmptyStackException extends Exception {
5
6     public EmptyStackException() {
7     }
8
9 }
10 // ENDKILL
```

Fin réponse

▷ **Question 2.** Écrire une classe `LIFOStack` implémentant l'interface `Stack` qui réalise une pile du type *Last In, First Out* (la dernière valeur empilée est la première valeur à être dépilée). Pensez à lever une exception quand c'est nécessaire.

Indication : Comme structure interne de votre pile, vous utiliserez par exemple un champ particulier qui sera de classe `ArrayList` (il faut importer `java.util.ArrayList` pour pouvoir utiliser cette classe). Les méthodes dont vous aurez besoin sont les suivantes (rappel) :

- connaître la taille de la collection : `coll.size()`
- ajouter un élément `elm` : `coll.add(elm)`
- lire l'élément à la position `pos` : `coll.get(idx)`
- retirer l'élément à la position `pos` : `coll.remove(idx)`

Réponse

```

1 package exo1;
2
3 import java.util.ArrayList;
4
5 public class LIFOStack implements Stack {
6 // BEGINKILL
7     private ArrayList<Value> values = new ArrayList<Value>();
8
9     public boolean isEmpty() {
10         return this.values.isEmpty();
11     }
12
13     public void push(Value v) {
14         this.values.add(v);
15     }
16
17     public Value pop() throws EmptyStackException {
18         if (isEmpty()) {
19             throw new EmptyStackException();
20         }
21         return (Value) this.values.remove(this.values.size()-1);
22     }
23
24     public Value peek() throws EmptyStackException {
25         if (isEmpty()) {
26             throw new EmptyStackException();
27         }
28         return (Value) this.values.get(this.values.size()-1);
29     }
30
31 // ENDKILL
32 }

```

Fin réponse

▷ **Question 3.** Écrire une classe `TestStack` qui crée une instance de la classe `LIFOStack`, qui empile une valeur et essaye de la dépiler. N'oubliez pas de capturer les exceptions qui peuvent être levées.

Réponse

```

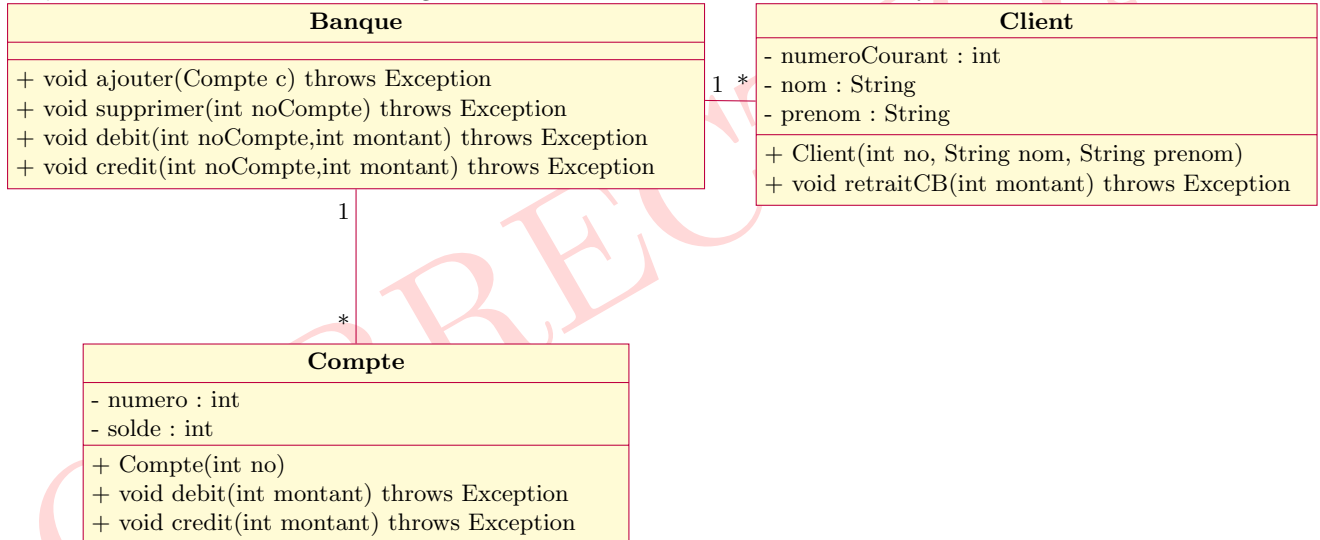
1 package exo1;
2
3 public class TestStack {
4 //BEGINKILL
5     public static void main(String args[]) {
6         Stack s = new LIFOStack();
7         s.push(new Value("bad", 666));
8         try {
9             System.out.println(s.pop());
10        } catch (EmptyStackException ex) {
11            ex.printStackTrace();
12        }
13    }
14 //ENDKILL
15 }

```

Fin réponse

★ Exercice 2.

▷ **Question 1.** On considère le diagramme de classes suivant modélisant un système bancaire.



Implanter ce diagramme sachant que l'appel à la méthode `retraitCB()` déclenche un appel à la méthode `debit()` de la classe `Banque` qui elle même fait appel à la méthode `debit()` de la classe `Compte`. Dans le cas où le solde ne permettrait pas d'effectuer le retrait d'argent, une exception du type `PasAssezArgentException` doit être levée et propagée.

Réponse

Cet exercice est optionnel.

La question est un peu draft, mais un jour je l'améliorerai. Promis ;-)

Pour expliquer aux étudiants, il est possible de dérouler l'exécution de la méthode `retraitCB()` en dessinant la pile des appels. On peut ainsi montrer comment "remonte" l'exception et l'endroit où elle est "attrapée".

Fin réponse

★ Exercice 3.

▷ **Question 1.** On considère que l'exécution `instruction2` lève une exception dans le bloc `try-catch`

```

1 try {
2     instruction1;
3     instruction2;
4     instruction3;
5 } catch (Exception1 ex1) {
6     // ...
7 } catch (Exception2 ex2) {
8     // ...
9 }
10 instruction4;
  
```

- L'instruction `instruction3` sera-t-elle exécutée ?
- Si l'exception n'est pas attrapée, l'instruction `instruction4` sera-t-elle exécutée ?
- Si l'exception est attrapée dans un des blocs `catch`, l'instruction `instruction4` sera-t-elle exécutée ?
- Si l'exception est passée à la méthode appelante, l'instruction `instruction4` sera-t-elle exécutée ?

Réponse

- NON, `instruction3` ne sera pas exécutée
- NON, l'exception est passée à la méthode appelante
- OUI, le code du bloc attrapant l'exception sera dans un premier temps exécuté, puis `instruction4`

(d) NON

Fin réponse

▷ **Question 2.** On considère que l'exécution `instruction2` lève une exception dans le bloc `try-catch`

```

1 try {
2     instruction1;
3     instruction2;
4     instruction3;
5 } catch (Exception1 ex1) {
6     // ...
7 } catch (Exception2 ex2) {
8     // ...
9 } catch (Exception3 ex3) {
10    throw ex3;
11 } finally {
12    instruction4;
13 }
14 instruction5;
```

- (a) L'instruction `instruction5` sera-t-elle exécutée si l'exception n'est pas attrapée ?
- (b) Si l'exception levée est de type `Exception3`, l'instruction `instruction4` sera-t-elle exécutée ? l'instruction `instruction5` sera-t-elle exécutée ?

Réponse

- (a) NON, l'exception est passée à la méthode appelante
- (b) OUI, `instruction4` sera exécutée, NON `instruction5` ne sera pas exécutée.

En question subsidiaire, on peut même leur demander l'ordre d'exécution.

Fin réponse

★ **Exercice 4.** On considère l'interface Java suivante :

```

1 package exo4;
2
3 public interface IterateurTabInt {
4     public int suivant();
5     public int indiceDuSuivant();
6     public boolean aUnSuivant();
7 }
```

On veut écrire une classe d'itérateurs qui parcourent uniquement les cases contenant un nombre pair dans un tableau quelconque d'entiers. Cette classe se nommera `IterateursDesPairs` et implantera l'interface `IterateurTabInt`.

Le sens du parcours est celui des indices croissants. La référence du tableau à parcourir est un attribut noté `tab` et est transmise lors de l'instanciation de l'itérateur. L'indice du tableau dont la valeur est retournée par la méthode `suivant()` est noté `pos`. Cet indice est mis à jour la première fois à l'instanciation, et, les fois suivantes, lors des appels à la méthode `suivant()`. Il n'y a plus de suivant quand cet indice est égal à la longueur du tableau.

Les premières déclarations de la classe sont donc :

```

1     private int[] tab;
2     private int pos;
3
4     public IterateurDesPairs(int[] tab) {
5         this.tab = tab;
6
7         // à compléter
8     }
```

▷ **Question 1.** Écrivez le code de la classe `IterateurDesPairs`.

Réponse

```

1 package exo4;
2
3 public class IterateurDesPairs implements IterateurTabInt {
4     private int[] tab;
5     private int pos;
6
7     public IterateurDesPairs(int[] tab) {
8         this.tab = tab;
9         // BEGINKILL
10        pos = -1;
11        // ENDKILL
12    }
13
14    public int suivant() {
15        // BEGINKILL
16        pos = indiceDuSuivant();
17        return tab[pos];
18        // REPLACE
19        // return -1;
20        // ENDKILL
21    }
22    public int indiceDuSuivant() {
23        // BEGINKILL
24        int indice = pos+1;
25        while ((indice < tab.length) && (tab[indice] % 2 != 0)) {
26            indice++;
27        }
28
29        return indice;
30        // REPLACE
31        // return -1;
32        // ENDKILL
33    }
34    public boolean aUnSuivant() {
35        // BEGINKILL
36        int indice = indiceDuSuivant();
37        return (pos < indice) && (indice < tab.length) ;
38        // REPLACE
39        // return false;
40        // ENDKILL
41    }
42 }

```

Fin réponse

▷ **Question 2.** Complétez le corps du constructeur de la classe `Test` pour avoir à l'exécution les affichages suivants (format : indice de la valeur entière paire → entier pair)

```

1 test1 : 1->2,2->6,4->8,7->12,8->14,
2 test2 : 0->2,
3 test3 :

```

```

1 package exo4;
2
3 public class TestIterateur {
4     public TestIterateur(String message, int[] t) {
5         IterateurDesPairs i = new IterateurDesPairs(t);
6         System.out.print(message);
7
8
9         /******
10        /* A VOUS DE JOUER */
11        /******
12    }
13
14    public static void main(String[] args) {
15        new TestIterateur("test1 : ", new int[] {1,2,6,7,8,9,11,12,14,13,5});
16        new TestIterateur("test2 : ", new int[] {2,79});
17        new TestIterateur("test3 : ", new int[] {});
18    }
19 }
20

```

Réponse

```
1 package exo4;
2
3 public class TestIterateur {
4     public TestIterateur(String message, int[] t) {
5         IterateurDesPairs i = new IterateurDesPairs(t);
6         System.out.print(message);
7
8         // BEGINKILL
9         while (i.aUnSuivant()) {
10             System.out.print(i.indiceDuSuivant()+"->" + i.suivant() + ",");
11         }
12         System.out.println();
13         // ENDKILL
14     }
15
16     public static void main(String[] args) {
17         new TestIterateur("test1 : ", new int[] {1,2,6,7,8,9,11,12,14,13,5});
18         new TestIterateur("test2 : ", new int[] {2,79});
19         new TestIterateur("test3 : ", new int[] {});
20     }
21 }
```

Fin réponse
