Errata 1

Les exceptions

« Attrapez-les tous, sans exception ! »

1.1 Motivation

Un programme ne tourne pas dans un monde idéal.

Il doit pouvoir *résister aux défaillances* de l'environnement, telles que :

- ▷ on tente d'ouvrir un fichier qui n'existe pas ;
- ▷ l'utilisateur entre des données incorrectes ;
- ▷ une connexion à un site web ne se fait pas ;
- ▷ etc.

Tous ces évènements *exceptionnels* viennent perturber le fonctionnement de votre programme, et doivent être gérés. Le langage Java offre un mécanisme, celui des exceptions.

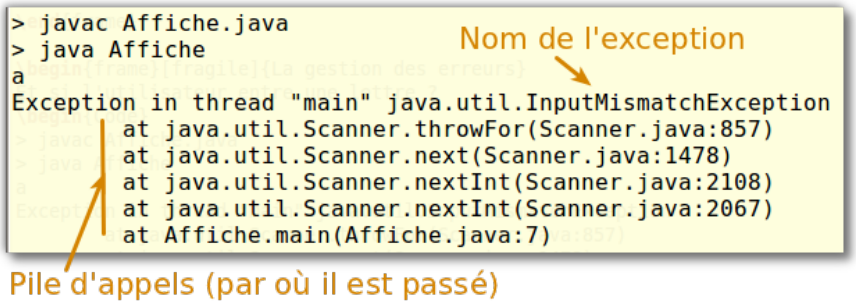
À titre d'exemple, considérons le programme suivant

```java
import java.util.Scanner;
public class Affiche {
  /**
   * Affiche un nombre entier lu au clavier.
   * @param args non utilisé
   */
  public static void main(String[] args) {
      Scanner clavier = new Scanner(System.in);
      int nb;

      nb = clavier.nextInt();
      System.out.println(nb);
  }
}
```

java

Que se passe-t-il lorsque l'utilisateur entre une lettre, ou un nombre décimal ?



On constate qu'une Exception a été levée (ou « lancée »), ce qui résulte en l'arrêt prématuré du programme.

Remarque

Ici, c'est une InputMismatchException qui a été lancée et pas une Exception. En fait, nous verrons en *Développement II* qu'il existe plusieurs exceptions différentes.

Celles que nous pouvons déjà retenir sont :

InputMismatchException
: par exemple lorsque l'utilisateur ou l'utilisatrice entre une valeur qui n'est pas du type attendu ;

ArrayIndexOutOfBoundException
: par exemple lorsque le programme essaie d'accéder à un élément d'un tableau hors de celui-ci ;

IllegalArgumentException
: par exemple lorsque l'argument passé à une méthode n'est pas dans la plage des valeurs attendues ;

NullPointerException
: par exemple lorsque l'on essaie d'accéder à un tableau qui n'a pas été créé.

1.2 L'instruction try-catch

Une Exception peut arrêter l'exécution du programme. Cependant, le mécanisme autorise aussi d'intercepter (ou « attraper ») les Exceptions, afin de traiter l'erreur de façon transparente pour l'utilisateur. On utilise à cet effet une instruction try-catch, constituée de deux blocs de code :

- ▷ try : contient les instructions qui *peuvent mal se passer* ;
- ▷ catch : contient le code qui est *en charge* de gérer le problème.

Voici la grammaire simplifiée (voir [GJS+17] page 463 pour la grammaire complète) :

TryStatement :
`try` *Block Catches*

Catches :
CatchClause {CatchClause}

CatchClause :
`catch (` *Type Identifier* `)` *Block*

Quand une exception est levée dans un try :

- ▷ le code du try est interrompu ;
- ▷ le code du catch est exécuté ;
- ▷ l'exécution reprend après le bloc try-catch.

À titre d'exemple, on peut ainsi intercepter les exceptions levées par nextInt() comme ceci :

```java
import java.util.Scanner;
public class Affiche {
  /**
   * Affiche l'entier lu au clavier, ou un message si ce n'est pas
    un entier.
   * @param args inutilisé.
   */
  public static void main(String[] args) {
      Scanner clavier = new Scanner(System.in);
      int nb;
      try {
        nb = clavier.nextInt();
        System.out.println(nb);
      }
      catch(Exception e) {
        System.out.println("Ce␣n'est␣pas␣un␣entier!");
      }
  }
}
```

java

Afin d'être plus spécifique, on peut nommer dans le catch l'exception précise à intercepter, par exemple InputMismatchException.

1.3 L'instruction throw

Imaginons la situation où un programme doit demander un *entier positif* à l'utilisateur. Si le programme ne vérifie pas immédiatement que le nombre donné est entier et positif, il s'expose à des problèmes tels que plantages, résultats erronés, perte ou corruption de données, etc. avec la condition aggravante que le problème pourrait ne se signaler que plus loin dans le code, et plus tard dans le temps, rendant le déboguage plus difficile. Cette constatation est à la base du mantra « Fail early, fail loudly. ».

Dès que le développeur ou la développeuse constate que son code pourrait être dans une situation instable — pourrait *planter* à court terme — à cause d'un argument par exemple, elle décide de lever une exception en utilisant l'instruction throw :

ThrowStatement :
`throw` *Expression*`;`

Cas pratique : vérifier les *arguments* reçus. Si un paramètre est supposé être un entier positif, une saine gestion consiste à vérifier que c'est le cas et à lever une exception sinon.

```java
/**
 * Calcule la racine carrée d'un nombre.
 * @param nb le nombre dont on veut la racine carée.
 * @return la racine carrée de <code>nb<\code>.
 * @throws IllegalArgumentException si <code>nb</code> est
  négatif.
 */
public static double racineCarrée(double nb) {
  if (nb<0) {
    throw new IllegalArgumentException("nb␣doit␣être␣positif!");
  }
  // Traitement normal. On est sûr que le paramètre est OK.
}
```

java

```java
try {
  System.out.println( racineCarrée( val ) );
} catch (Exception ex) {
  System.out.println( "Calcul␣impossible␣!" );
}
```

java

Comme annoncé précédemment, on peut aussi préciser qu'on n'intercepte *que* les IllegalArgumentException

```java
try {
  System.out.println( racineCarrée( val ) );
} catch (IllegalArgumentException ex) {
  System.out.println( "Calcul␣impossible␣!" );
}
```

java

Références

[GJS+17] James Gosling, Bill Joy, Guy Steele, Gilad Bracha, Alex Buckley, and Daniel Smith. *The Java Language Specification. Java SE 9 Edition.* Oracle America Inc., 2017.