Programmation Orientée Objet – Java

Cours 6 : Gestion des erreurs “Exceptions”

Viviane Pons

Master BIBS Université Paris-Saclay

2 sortes d'erreurs quand on programme

Compilation / excécution

Erreurs à la compilation

- erreur de syntaxe
- erreur de type

Ces erreurs peuvent être repérées de façon "statique" : sans exécuter le code. Elles sont repérées par Eclipse et vous empèchent de lancer l'application.

Erreurs à l'exécution

Ces erreurs se produisent au moment où on exécute le programme. Elles sont plus difficiles à repérer car elles dépendent du contexte dans lequel le programme est exécuté. Souvent, elles sont plus embêtantes.

Note : il est IMPOSSIOBLE d'écrire un programme qui pourrait décider dans tous les cas si un autre programme quelconque crée une erreur ou non. (C'est ce qu'on appelle un problème indécidable)

Remarque : Il existe aussi d'autres erreurs, les erreurs de logique ou fonctionnelles. Le programme s'exécute sans problème mais ne fait pas ce qu'on veut. C'est les plus pénibles !

Comment ça se passe en C

Quand un programme C rencontre une erreur à l'exécution . . . Il plante.

Exemple courant : `Segmentation Fault`

Une erreur d'exécution en C termine le programme et nous donne en général assez peu d'information (à quelle ligne s'est produite l'erreur ? Dans quel contexte ? Qui avait appelé cette fonction ?)

Comment ça se passe en Java (et en python)

Il y a une couche supplémentaire qui supervise l'exécution et permet "d'encapsuler" les erreurs d'exécution. C'est ce qu'on appelle les "Exceptions"

Les Exceptions en Java

Une `Exception` est un objet qui contient toute l'information sur l'erreur :

- le type d'erreur
- le message
- où dans le code a eu lieu l'erreur
- la série d'appel où a eu lieu l'erreur = la "trace"

```
Exception in thread "main" java.lang.NumberFormatException
    at java.base/java.lang.NumberFormatException.forInputSt
    at java.base/java.lang.Integer.parseInt(Integer.java:66
    at java.base/java.lang.Integer.valueOf(Integer.java:999
    at ExempleExceptions.main(ExempleExceptions.java:48)
```

Différents types d'erreurs

Il y a plusieurs classes qui représentent des erreurs d'exécution :

- `Throwable` (du verbe *throw*, lancer – en français, on dit "lever" une exception) : la superclasse de toutes les erreurs en Java
- `Error` : une sous-classe qui correspond aux erreurs "graves" où on ne peut "rien faire" (ça plante...). Exemple `StackOverflowError`
- `Exception` : les erreurs "normales"

En tant que développeur / développeuse, on doit s'occuper des "erreurs normales", c'est-à-dire des objets qui héritent de `Exception`

Plein, plein, plein d'Exceptions différentes

Chaque type d'erreur à sa propre classe qui hérite de `Exception` :
`NumberFormatException`, `FileNotFoundException`,
`IOException`
Elles se rangent en 2 catégories :

- celles qui héritent de `RunTimeException` : en général liées à une *erreur de conception* dans le programme. Exemple : `NumberFormatException`, `NullPointerException`, `IndexOutOfBoundException`
- toutes les autres : liées au contexte d'utilisation du programme. Par exemple, les interactions avec l'utilisateur. Ces erreurs là doivent être *gérées par le programme*.

Gestion des exception

try ... catch

On peut “attraper” les exceptions pour décider de la suite à donner.

```
File f = new File("IDontExist");
try {
    Scanner scan = new Scanner(f);
} catch (FileNotFoundException e) {
    e.printStackTrace();
}
System.out.println("I am still here");
```

Typiquement : si le programme veut charger un fichier (par exemple, une partie de Tétris) mais n'arrive pas à lire le fichier. On veut :

1. afficher un message à l'utilisateur "Le fichier ne marche pas !"
2. continuer à faire fonctionner le jeu (par exemple pour essayer avec un autre fichier)

Si on on ne les attrape pas, on doit les signaler

La méthode qui utilise une action “dangereuse” doit signaler l'exception qui peut être levée.

```
    public static void readNumbers(String filename) throws
        File f = new File(filename);
        Scanner scan = new Scanner(f);
        while(scan.hasNext()) {
            int v = scan.nextInt();
            System.out.println(v);
        }
    }
```

On peut aussi lever ses propres exceptions

```
try {
    int v = scan.nextInt();
    System.out.println(v);
} catch(InputMismatchException e)  {
    throw new IOException(e);
}
```

Un exemple plus complet

```
public static void readNumbers(String filename) throws
    File f = new File(filename);
    Scanner scan = new Scanner(f);
    while(scan.hasNext()) {
        try {
            int v = scan.nextInt();
            System.out.println(v);
        } catch(InputMismatchException e)  {
            throw new IOException(e);
        }
    }
    scan.close();
}

public static void safeRead(String filename) {
    try {
        readNumbers(filename);
    } catch(FileNotFoundException e) {
        System.out.println("I cannot open this file : "
```

Enfin : on peut créer ses propres exceptions

Le but est de lever l'exception appropriée pour chaque problème rencontré par le programme.
La librairie java offre déjà de nombreuses exceptions qu'on peut réutiliser quand on en a besoin.
Si on souhaite être plus précis : on peut aussi créer nos propres objets qui héritent de la classe `Exception` (ou d'une de ses sous classes)

Exemple

```
public class MyProblemException extends IOException {

    public MyProblemException()
        {
            super();
        }
        public MyProblemException(String message)
        {
            super(message);
        }
        public MyProblemException(Throwable cause)
        {
            super(cause);
        }
        public MyProblemException(String message, Throwable
        {
            super(message, cause);
        }
}
```