# Développement Orienté Objets
# **Exceptions**

Petru Valicov
petru.valicov@umontpellier.fr

https://gitlabinfo.iutmontp.univ-montp2.fr/dev-objets

2021-2022




## Problématique

Parfois un programme peut "planter" indépendamment de la volonté du programmeur :

- Problème matériel ou de connexion réseau
- Mauvaise saisie utilisateur
- Problème lié à l'exécution d'une IHM
- Accès à un fichier introuvable

... et ça serait bien de prévoir et maîtriser ces problèmes.

**Définition**

Une **exception** est un événement qui intervient à l'exécution, et qui interrompt le cours *normal* d'instructions.

- contraire à l'exécution classique du code (du début à la fin)
- une exception **ne doit pas** être considérée comme un bug
  - parce qu'elle est prévue
  - parce qu'elle est contrôlée



## Exemples de situations exceptionnelles

```
double diviseur = Math.random();
double division = 36 / diviseur; // est-ce sans danger ?
```

```
Scanner scanner = new Scanner(System.in);
System.out.println("Veuillez saisir un nombre svp :");

int nombre = scanner.nextInt(); // est-ce une instruction sûre ?

double calcul = nombre * 30 - 5;
```

```
Scanner scanner = new Scanner(System.in);
System.out.println("Veuillez saisir un nom de fichier svp :");
String nom = scanner.next();

FileReader lecteur = new FileReader(nom); // le fichier existe ?

int valeur = lecteur.read();
lecteur.close();
```



## Idéalement, que faire en cas d'erreur ?

- Envoi de message
- Sauvegarde du travail en cours, lorsque possible
- Retour à un état stable

**Première tentative (supposons que le code compile) :**

```
Scanner scanner = new Scanner(System.in);
System.out.println("Veuillez saisir un nom de fichier svp :");
String nom = scanner.next();
File file = new File(nom);
if(!file.isFile()) {
    System.out.println("Ceci n'est pas un fichier");
}
else {
    if (file.length() == 0) {
        System.out.println("Le fichier est vide");
    }
    else {
        FileReader lecteur = new FileReader(nom);
        int valeur = lecteur.read();
        lecteur.close();
    }
}
```

**Problème ?**

## Idéalement, que faire en cas d'erreur ?

- Envoi de message
- Sauvegarde du travail en cours, lorsque possible
- Retour à un état stable

**Deuxième tentative :**

```
Scanner scanner = new Scanner(System.in);
System.out.println("Veuillez saisir un nom de fichier svp :");
String nom = scanner.next();
try {
    FileReader lecteur = new FileReader(nom);
    int valeur = lecteur.read();
    lecteur.close();
}
catch (FileNotFoundException e) {
    System.err.println("Fichier non-trouvé : " + e.getMessage());
}
catch (IOException e) {
    System.err.println("La lecture du fichier est problématique : " + e.getMessage());
}
```

- séparation entre le code métiers, et le code de gestion des cas pathologiques
- meilleure lisibilité



## Les différentes *"erreurs"* en Java

- La hiérarchie `Error` :
  - Erreurs internes à la JVM, manque de ressources d'exécution, etc.
  - hors contrôle (*unchecked exceptions*) : pas de gestion explicite par les programmes
  - provoquent généralement l'arrêt brutal du programme.
- La hiérarchie `Exception` :
  1. les exceptions sous contrôle (*checked exceptions*) : les conditions exceptionnelles du programme que le développeur **doit traiter/gérer**
  2. les exceptions d'exécution de type `RuntimeException` : erreurs (bugs) de programmation (taille tableau, tentative d'accès à référence nulle, etc.).

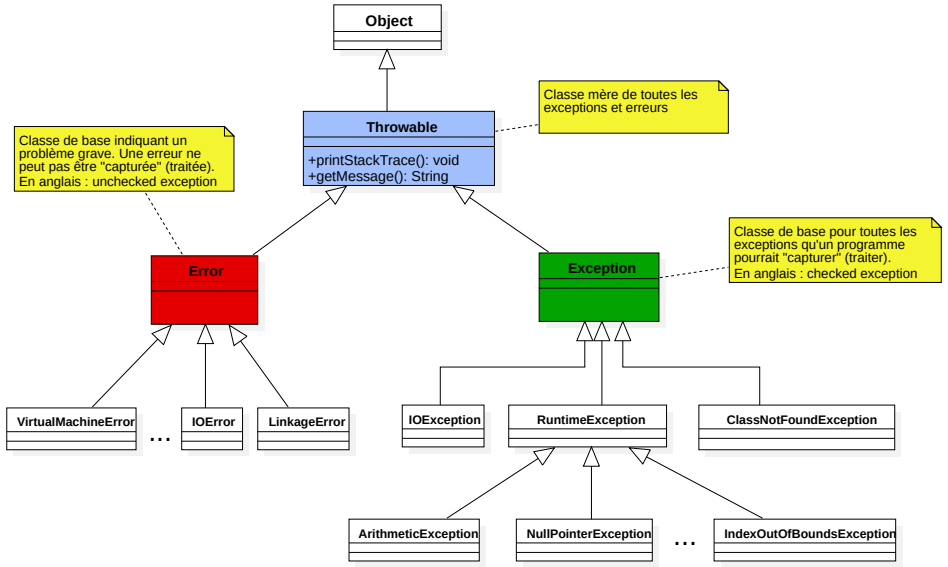
**Recommandation**

Ne traitez pas les `Error` et traitez les `RuntimeException` qu'en cas exceptionnels et avec discernement (par ex. lorsqu'**il ne s'agit pas** d'erreurs de programmation incontrôlées/incomprises)

**À votre avis, pourquoi ?**



## Hiérarchie des classes d'exceptions en Java






## Gestion des exceptions

Deux aspects à prendre en compte dans la gestion des exceptions :

- *signaler* un cas d'exception à un programme client (i.e. appelant) : **lever une exception**
- *intercepter* une exception signalée par un programme appelé : **traiter une exception**

**Important**

Tout appel de programme susceptible de lever une exception sous contrôle doit être traité dans un bloc spécifique.

Lorsqu'une exception est levée, une exécution anormale (ou exceptionnelle) du programme doit se produire.

## Les exceptions - schéma d'exécution

Quand une exception se produit :

1. l'exécution normale du code s'arrête
2. un objet de type `Exception` (ou une de ses sous-classes) est créé
3. cet objet est transféré au runtime

} **levée** d'exception (*throw*)

4. le runtime Java cherche un bloc de traitement d'exception (*exception handler*) dans la pile d'appel

} **interception** (*catch*)

```
// du code fonctionnant correctement

try{
   // du code susceptible de poser problème (i.e. lever une exception)
} catch (IOException e){
   // traitement correspondant aux entrées/sorties
} catch (NullPointerException e){
   // traitement correspondant aux problèmes d'accès à des références null
} finally{
   // code à exécuter dans tous les cas
}

// du code fonctionnant correctement
```



## Levée d'exception

```
Scanner scanner = new Scanner(System.in);
System.out.println("Veuillez saisir un nombre svp :");
try{
   // code susceptible de lever une exception
   int nombre = scanner.nextInt();
}
catch (InputMismatchException exc) {
   // Code exécutée uniquement en cas d'exception levée
   System.out.println("Pardon, mais ce que vous venez de saisir n'est pas un nombre");
   System.err.println(exc.getMessage());
   exc.printStackTrace();
}
```

Deux scénarios possibles :

- soit aucune exception n'est levée, auquel cas après le bloc `try` l'exécution continue normalement ;
- soit une exception est levée, auquel cas l'exécution cherche un bloc `catch`, qui spécifie le traitement à effectuer.



## Les exceptions : capture (`catch`)

Une fois l'exception levée :

1. le runtime Java cherche dans la pile des appels le bloc `catch` **le plus profond** pour ce type d'exception :
   - ce bloc `catch` définit le traitement approprié à l'exception
   - les blocs `catch` doivent être les plus spécifiques possible

     Exemple horrible (il vaut mieux ne rien faire plutôt que faire ça !) ⟶

     ```
     try{ ... }
     catch(Exception e){...}
     ```

   - Les actions standards (minimales) à exécuter :
     - `e.getMessage()`
     - `e.printStackTrace()`
2. l'exécution reprend ensuite après ce bloc

**Remarques**

- le fonctionnement est implicite pour les exceptions de sous-type `RuntimeException`
- le bloc `finally` permet s'assurer qu'un traitement sera exécuté dans tous les cas après le bloc `try` (pour faire un nettoyage de ressources utilisées)



## Les exceptions : transfert (`throws`)

Il est possible de ne pas lever & traiter le problème immédiatement,

**mais** on doit indiquer au compilateur qu'on accepte les "risques" et demander que l'exception soit *transférée* :

```
public void lire() throws IOException{
    System.in.read();
}
```

Du coup, effet boule de neige : toutes le méthodes appelant `lire()` doivent soit inclure un `try/catch` soit avoir une clause `throws`

```
public void parser() throws IOException{
    lire();
}
```

ou

```
public void parser(){
   try{
      lire();
   }catch(IOException e){
      System.out.println("Quelque chose de pas très cool vient de se passer !");
      System.out.println(e.getMessage()); //On affiche au moins le message détaillé
      System.out.println("Au secours !!!");
   }
}
```

## Levée d'exception : l'instruction `throw`

- **Rappel** : toutes les classes d'exceptions/erreurs héritent de la classe `Throwable`
- Toute exception/erreur est levée grâce à l'instruction `throw` :
  - de manière explicite
  - de manière implicite dans un bloc **try**

```
public class AppExpression {
   public static void main(String[] args) {
      // exception hors contrôle - pas de "try/catch", ni de "throws"
      throw new RuntimeException("Méthode non-implémentée");
   }
}
```

```
// l'exception sera transférée à l'appelant si problème
void lireFichier(File fichier) throws FileNotFoundException {
     if (!fichier.exists())
        throw new FileNotFoundException();

     Scanner scan = new Scanner(fichier);
     // du code utilisant scan
  }
```

```
void lireFichierBis(File fichier) {
    try {
        Scanner scan = new Scanner(fichier);
        // du code utilisant scan
    } catch(FileNotFoundException e) {
        e.printStackTrace();
    }
}
```

**Attention aux mots-clés** : `throw` $\neq$ `throws`



```
public class MauvaisNombreException extends Exception {
    private String message;
    public MauvaisNombreException(String s) { message = s; }

    public String recupererMessage(){
        return message;
    }
}
```

```
public class ExemplesExceptions {
    public int diviser(int divise, int diviseur) throws MauvaisNombreException {
        if (diviseur == 0)
            throw new MauvaisNombreException("Ne peut pas diviser par 0");
        return divise / diviseur;
    }

    public void appelDiviseur() {
        try {
            int resultat = diviser(2,1);
            System.out.println(resultat);
            resultat = diviser(2,0);
            System.out.println(resultat); //pas exécuté
        } catch (MauvaisNombreException e) {
            //traitement approprié
            System.out.println(e.recupererMessage());
        }
        System.out.println("Essai de la division terminé");
    }

    public void appelDiviseur2() throws MauvaisNombreException {
        int resultat = diviser(2,1);
        System.out.println(resultat);
        resultat = diviser(2,0);
        System.out.println(resultat); //pas exécuté
    }
}
```



## *try-with-ressources*

Plusieurs librairies Java contiennent des ressources qu'il faut fermer manuellement. Exemples :

- `InputStream`, `OutputStream` – flux d'octets d'entrée et sortie
- `InputStreamReader`, `BufferedReader` – flux de caractères

Il faut toujours penser à fermer un flux de donnée après utilisation :

```
// déclaration d'un lecteur de fichier "bufférisé" et d'un rédacteur
BufferedReader lecteur = new BufferedReader(new FileReader("fichier.in"));
BufferedWriter rédacteur = new BufferedWriter(new FileWriter("fichier.out"));
try {
    // du code ici utilisant le lecteur et le rédacteur
    String s = lecteur.readLine();
    rédacteur.write(s);
    // du code ici utilisant le lecteur et le rédacteur
}
finally {
    lecteur.close();
    rédacteur.close(); // et si l'instruction précédente lève une exception ???
}
```

**Problèmes avec la fermeture manuelle :**

- c'est fastidieux
- les choses se compliquent si plusieurs ressources à gérer...



## *try-with-ressources*

Ne pas utiliser le bloc *try-finally* pour fermer les ressources.

Si une seule ressource :

```
// déclaration d'un bloc try-with-ressource avec une seule ressource
try (BufferedReader lecteur = new BufferedReader(new FileReader("chemin d'accès"))) {
    // du code ici utilisant la variable lecteur
    lecteur.readLine();
    // du code ici utilisant la variable lecteur
}
```

Si plusieurs ressources :

```
// la déclaration d'un bloc try-with-ressource avec une plusieurs ressources
try (
    BufferedReader lecteur = new BufferedReader(new FileReader("fichier.in"));
    BufferedWriter rédacteur = new BufferedWriter(new FileWriter("fichier.out"))
) {
   // du code ici utilisant la variable lecteur
   lecteur.readLine();
   // du code ici utilisant la variable lecteur
  }
```

Le bloc *try-with-ressources* va fermer correctement les ressources quoiqu'il arrive.

## Qu'est-ce qu'on en fait ?

- Traiter les exceptions à l'endroit utile :
  - annuler l'opération en cours
  - afficher un message à l'utilisateur
  - rectifier la situation et relancer le traitement
- Ne **jamais** lever une exception et ne rien faire !
- Spécifier le type le plus précis possible :

  pas de `catch (Exception e)`

  pas de `throws Exception`
- N'hésitez pas à définir des exceptions personnalisées
- Utiliser le bloc "*try with ressources*" pour libérer les ressources (fichiers, connexion réseau, ...) ouvertes dans le `try` correspondant
- Une bonne gestion des exceptions indique généralement un projet solide et bien structuré

Plus de documentation sur le site d'Oracle :

https://docs.oracle.com/javase/tutorial/essential/exceptions/index.html



## Exceptions : bêtisier

Qu'en pensez-vous ?

```
try {
    int taille = (int) (Math.random()*1000);
    double tab[] = new double[taille];
    int index = 0;
    double min = tab[index];
    while (true){
        if (min > tab[index])
            min = tab[index];
        index++;
    }
} catch (ArrayIndexOutOfBoundsException e) {
    System.out.println("La recherche est terminée, le minimum est " + min);
}
```

```
ArrayList<String> texte;

/* du code écrit ici et que vous n'avez pas eu le temps de regarder... */

String motif = "motRecherché";
try {
    Iterator<String> it = texte.iterator();
    while (!motif.equals(it.next())) ; // attention au ; !

    System.out.println(motif + " trouvé !!!");

} catch (NullPointerException e) {
    System.out.println("Le tableau n'a pas été correctement instancié");
} catch (NoSuchElementException e) {
    System.out.println("Fin de l'itération, je n'ai pas trouvé le mot");
}
```