Informatique - Java

Nicolas Rousset

Gestion des exceptions

Java - exceptions

Rappel sur le Runtime VS Compilation

Il y a deux étapes à bien distinguer en java, la **compilation** et le **runtime**.

Simplement la première désigne le fait de transformer votre code source en code exécutable, le deuxième le moment où votre programme est effectivement exécuté.

Les erreurs dans la première partie sont liées à des problèmes de syntaxe ou de typages incohérents; elles sont levées directement par le compilateur, et sont faciles à traiter (si votre code n'est pas trop mal structuré).

Les erreurs dans la deuxième partie sont liées aux valeurs des variables : elles ne sont donc pas détectable à priori par le compilateur. Le compilateur ne peut, par exemple, détecter systématiquement si une variable est bien initialisée avant que l'on ne fasse appel à elle.

Ici, erreur doit bien s'entendre comme une opération qui n'est pas possible, par exemple appeler une méthode sur un objet **null* accéder à un élément qui n'existe pas dans une liste, diviser par 0, etc ...

Les opérations qui ne produisent pas le bon résultat sont encore plus problématiques, car plus difficiles à détecter. C'est pour cela que l'on choisira parfois de lever une exception (voir plus loin), ie que la fonction se comporte comme si elle ne pouvait pas produire le résultat

Définition d'une exception

Une exception, comme son nom l'indique, est créée lorsqu'une opération impossible du fait des valeurs des variables est tentée.

On interrompt alors le code exécutant, et on gère cette anomalie.

Les exceptions ne doivent pas servir à cacher des erreurs de programmation

Au contraire, on rajoutera des exceptions pour tenir compte de ce qui a été supposé, afin de mieux détecter certains comportements incohérents.

Cas d'usage des exceptions

Les principaux cas d'usage des exceptions concernent la lecture de données externes ; lorsque vous lisez un fichier supposé être d'un certain format, que se passe-t-il si le fichier ne correspond pas ? Si il y a des lettres là où il devrait y avoir des chiffres ?

Le programme lève alors une exception.

Normalement si votre programme ne dépend pas du tout des données externes (ce qui serait en fait un peu étrange, il renverrait toujours le même résultat à chaque exécution) vous n'avez pas besoin d'exception.

Exception usuelles

Quelques exemples d'exception usuelles (voir dans le project de code java JavaCodeSamples exception/ExceptionsUsuelles.java)

java.lang.NullPointerException

Levée lorsque l'on fait n'importe quelle opération ou traitement sur un objet de valeur **null** (autre que **!=** ou **==**)

java.lang.ArithmeticException

Lorsqu'une opération arithmétique impossible est tentée (division par 0, souvent)

java.lang.IndexOutOfBoundsException

Lorsque l'on cherche à accéder à un élément qui n'existe pas (dans un tableau ou une liste)

java.lang.NumberFormatException

Lorsque l'on essaie de convertir en nombre (avec Integer.parse) une chaîne de caractère incompatible

java.lang.ClassCastException

Lorsque l'on fait une conversion explicite avec une valeur incompatible

java.nio.file.NoSuchFileException

Lorsque l'on cherche à accéder, avec les méthodes du package java.nio, à un fichier qui n'existe pas

Caractéristiques d'une exception

Avant de rentrer dans les détails du système spécifique de propagation d'une exception, notons qu'une exeption hérite nécessairement de la classe concrère java.lang.Exception, qui elle-même hérite de la classe Throwable, qui définit les caractéristiques d'une exception;

Une exception possède :

- un message; une String à destination de l'utilisateur
- une cause; ie une autre exception si l'exception a été lancée à cause d'une autre
- une classe; comme tout objet java, mais dans le cas des exceptions on l'utilise particulièrement
- une stackTrace; ie la liste des lignes de code qui ont mené à l'exécution de la ligne problématique

A noter que l'on peut créer directement des objets de la classe Exception, sans avoir à en hériter obligatoirement (il s'agit d'une classe concrète)

On accède à ces éléments avec les méthodes définies dans la classe Throwable
https://docs.oracle.com/javase/7/docs/api/java/lang/Throwable.html

getMessage()
getCause()
getClass()
getStackTrace() ou printStackTrace()

On définit ces valeurs dans le constructeur, en surchargeant les méthodes get mentionnées ci-dessus ou avec les setters (il n'y en a pas pour le message)

Système de propagation d'une exception

Regardons le résultat d'un main qui provoque une exception :

```
Exception in thread "main" java.lang.NullPointerException
    at exception.ExceptionsUsuelles.nullPointerException(ExceptionsUsuelles.java:46)
    at exception.ExceptionsUsuelles.callNullPointerException(ExceptionsUsuelles.java:23)
    at exception.ExceptionsUsuelles.callCallNullPointerException(ExceptionsUsuelles.java:19)
    at exception.ExceptionsUsuelles.main(ExceptionsUsuelles.java:15)
```

Ce qui s'affiche est la stackTrace, ie la liste des appels de fonction qui a mené à l'exception. Elle se lit de bas en haut : la fonction **main** a appelé la fonction **callCallNullPointerException** qui a appelé la fonction **callNullPointerException** qui a appelé la fonction **nullPointerException** qui a elle-même levé l'exception.

Une fonction qui lève une exception s'arrête donc instantanément

De la même façon que le **return**, sauf qu'ici on voit que cela se propage, toutes les fonctions appelantes s'arrêtent les unes après les autres. A noter que l'exception est le seul mécanisme qui permette ainsi d'interrompre la fonction appelante à partir de la fonction appelée.

Système de propagation d'une exception - le try / catch

La propriété d'une exception est donc de se propager, mais jusqu'à quand ?

En fait, si rien ne l'arrête, elle remonte jusqu'au **main** et interrompt donc le programme.
Le block qui est conçu pour l'arrêter est ce que l'on appelle le try / catch, qui ressemble à ceci :

```
List<Integer> liste = new ArrayList();
try
{
    Integer un = liste.get(0);
}
catch( IndexOutOfBoundsException ex )
{
    System.err.println( "Oups, quelque chose n'a pas marché" );
    System.err.println( ex.getLocalizedMessage() );
    ex.printStackTrace(System.err);
}
System.out.println("Mais on survit quand même");
```

On "essaie" un code et si celui-ci ne fonctionne pas, on gère l'exception.
Puis le code continue à être exécuté normalement.

Système de propagation d'une exception - le try / catch (2)

Le point remarquable du catch est que l'on spécifie l'exception que l'on veut attraper. Ainsi, la structure ci-dessous n'arrêtera pas la propagation de l'exception :

```
List<Integer> liste = new ArrayList();
try
{
    Integer un = liste.get(0);
}
catch( ArithmeticException ex )
{
    System.out.println( "Tout va bien !!" );
}
System.out.println("Mais on survit quand même");
```

En effet l'exception levée ne correspond à un type indiqué dans les catchs.

Système de propagation d'une exception - le try / catch (3)

On peut même ajouter plusieurs **catch** les uns derrière les autres pour attraper différentes exceptions :

```
...
catch( ArithmeticException ex )
{
    System.out.println( "J'ai attrapé une exception arithmétique" );
}
catch( IndexOutOfBondException ex )
{
    System.out.println( "J'ai attrapé une exception d'index hors domaine" );
}
catch( Exception ex )
{
    System.out.println( "J'attrape toutes les exceptions qui restent" );
}
```

Un seul catch sera activé. Comme toutes les exceptions héritent de **Exception**, un catch(Exception ex) attrapera toutes les exceptions.
Il n'est donc pas possible de rajouter de nouveau catch après le **catch(Exception ex)**
Un catch attrape l'exception mentionnée et toutes les exceptions héritées.

Système de propagation d'une exception - le try / catch / finally

On peut ajouter une clause **finally** après les catch. Celle-ci sera **toujours** exécutée que l'on catche ou pas une exception, et quelque soit l'exécution catchée, et y compris si l'exception n'est pas catchée.

```
...
catch( ArithmeticException ex )
{
    System.out.println( "J'ai attrapé une exception arithmétique" );
}
catch( Exception ex )
{
    System.out.println( "J'attrape toutes les exceptions qui restent" );
}
finally
{
    System.err.println("Erreur détectée, le reste de la liste à traiter est supprimé");
    liste.clear();
}
```

Système de propagation d'une exception - le try / catch / finally (2)

Typiquement, on l'utilisait avant pour fermer les fichiers :

```
BufferedReader br = new BufferedReader(new FileReader(path));
try
{
    return br.readLine();
}
finally
{
    if(br != null)
    {
        br.close();
    }
}
```

A noter qu'il existe une nouvelle syntaxe pour cela dans Java 7, plus élégante :

```
try( BufferedReader br = new BufferedReader(new FileReader(path)) )
{
    return br.readLine();
}
```