Afpa
se former, avancer

Java

POO / OPP : traitement des exceptions

Afpa

LES EXCEPTIONS
TRY...CATCH...FINALLY

LES EXCEPTIONS

La gestion des cas d'erreur représente un travail important dans la programmation. Les sources d'erreur peuvent être nombreuses dans un programme.

La robustesse d'une application est souvent comprise comme sa capacité à continuer à rendre un service acceptable dans un environnement dégradé, c'est-à-dire quand toutes les conditions attendues normalement ne sont pas satisfaites.

En Java, la gestion des erreurs se confond avec la gestion des cas exceptionnels. On utilise alors le mécanisme des exceptions.

Une exception est une classe Java qui représente un état particulier et qui hérite directement ou indirectement de la classe Exception.

Par convention, le nom de la classe doit permettre de comprendre le type d'exception et doit se terminer par Exception.

- NullPointerException
- NumberFormatException
- IndexOutOfBoundsException

Une exception est un événement inattendu survenant pendant l'exécution du programme.

- Entrée utilisateur non valide
- Échec d'un périphérique
- Perte de connexion réseau
- Limites physiques (mémoire insuffisante)
- Erreurs de code
- Ouvrir un fichier indisponible

HIÉRARCHIE DES EXCEPTIONS

la classe Throwable est la classe racine de la hiérarchie.

La liste des messages d'erreur après chaque tentative d'exécution est appelée stacktrace. La liste montre chaque méthode appelée lors de l'exécution du programme.

- Les programmes capables de gérer les exceptions de manière appropriée sont plus tolérants aux pannes et robustes.
- Les applications à tolérance de pannes sont conçues pour continuer à fonctionner, éventuellement à un niveau réduit, en cas de défaillance d'une partie du système.

La robustesse représente la capacité d'un système à résister aux contraintes et à continuer à fonctionner.

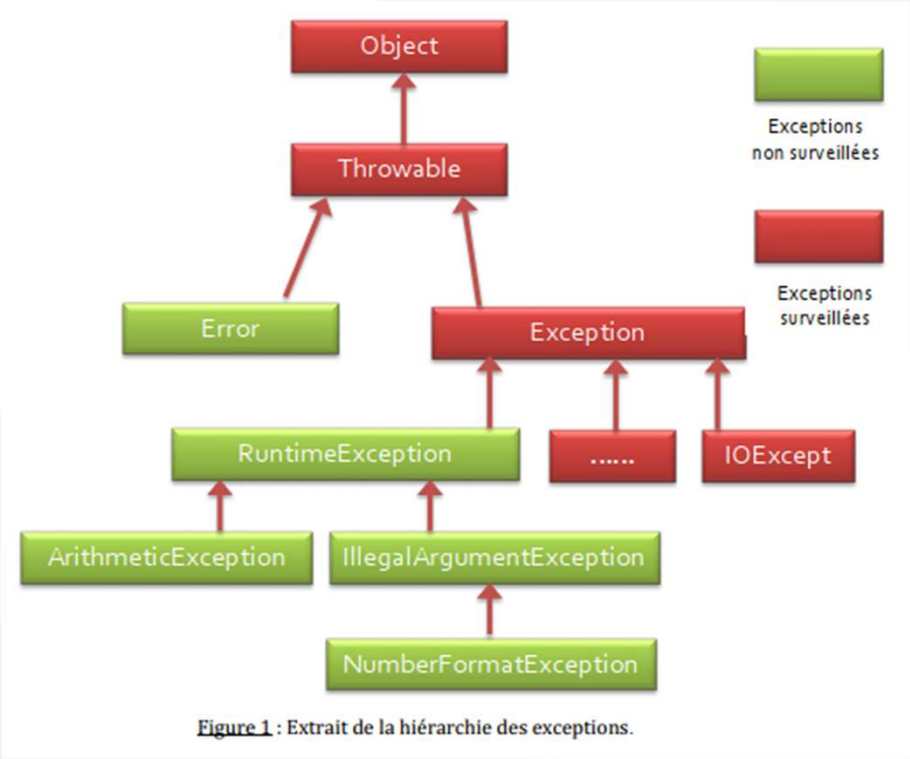

Figure 1 : Extrait de la hiérarchie des exceptions.

GESTION D'EXCEPTIONS EN JAVA

La classe Error

La classe Error représente des erreurs plus graves que votre programme ne peut généralement pas récupérer.

Par exemple, la mémoire.

La classe Exception

La classe Exception comprend les erreurs moins graves qui représentent des conditions inhabituelles survenant pendant l'exécution d'un programme et à partir desquelles le programme peut être restauré.

- Par exemple, l'indice d'un tableau dépassé.

Liste des exceptions Java

https://programming.guide/java/list-of-java-exceptions.html

HIÉRARCHIE DES EXCEPTIONS

Exceptions non surveillées

- Le compilateur Java ne vérifie pas les exceptions non surveillées.
- Elle se se produisent pendant l'exécution sont dues à des erreurs de logique du développeur (indice d'un tableau hors limites, division par zéro, méthode sur une référence null, …) ou par exemple atteinte des limites des ressources du système.
  1. Elles n'ont pas l'obligation d'être gérées par un catch.
  2. Les méthodes qui peuvent en lever ne doivent pas (mais peuvent) l'indiquer dans leur entête.
  3. Elles sont les exceptions du type RuntimeException et ses sous-classes

Exceptions surveillées

- Les exceptions surveillées sont vérifiées par le compilateur de Java.
- Les méthodes qui peuvent lever une exception surveillée doivent l'indiquer dans leur entête en précisant la clause throws.
- Toutes les exceptions surveillées doivent explicitement être contrôlées avec un bloc catch.
- L'exception remonte toute la pile d'appel de méthodes jusqu'à ce qu'un gestionnaire d'exception soit trouvé.
- Les exceptions surveillées comprennent toutes les exceptions du type Exception et ses sous-classes, *sauf la classe RuntimeException et ses sous-classes.*

GESTION D'EXCEPTIONS EN JAVA

Quelques méthodes utiles

Méthode	Description
public String getMessage()	Renvoie un message détaillé sur l'exception qui s'est produite. Ce message est initialisé dans le constructeur Throwable.
public String toString()	Renvoie le nom de la classe concaténée avec le résultat de getMessage ().
public Throwable getCause()	Renvoie la cause de l'exception représentée par un objet Throwable.

Capture et traitement des exceptions

Dans la terminologie orientée objet, vous essayez (**try**) une procédure pouvant provoquer une erreur. Une méthode qui détecte une condition d'erreur lève une exception (**throws**) et si vous écrivez un bloc de code qui traite l'erreur, ce bloc est dit intercepte l'exception (**catch**).

```
try{
    // traitements
}
catch(TypeException var){
    // gérer la condition d'erreur
}
```

TRY.. CATCH ET THROW

Try

Lorsque vous créez un segment de code dans lequel une exception peut survenir, vous placez le code dans un bloc try {…}

- ce bloc de code que vous essayez d'exécuter tout en reconnaissant qu'une exception pourrait se produire.

```
try{
    // traitements
}
catch(TypeException var){
    // gérer la condition d'erreur
}
```

Catch… Throw

Un bloc catch {…} est un segment de code qui peut gérer une exception, peut être levée par le bloc try qui le précède.

L'exception pourrait être celle qui est lancée automatiquement, ou vous pourriez écrire explicitement une instruction throw.

- Une instruction throw est une instruction qui envoie un objet Exception à partir d'un bloc ou d'une méthode afin qu'il puisse être géré ailleurs.

EXEMPLE

Dans l'exemple ci-dessus,

- si la variable heros vaut null alors le traitement du bloc try est interrompu par une NullPointerException.
- Sinon le bloc continue à s'exécuter.
- Si la condition est vraie, le traitement du bloc est interrompu par le lancement d'une FinDuMondeException et le traitement reprend dans le bloc catch…

```
try {
  if (heros == null) {
    throw new NullPointerException("Le heros ne peut pas être nul !");
  }

  boolean victoire = heros.combattre(espritDuMal);
  boolean planDejoue = heros.desamorcer(machineInfernale);

  if (!victoire || !planDejoue) {
    throw new FinDuMondeException();
  }

  heros.setPoseVictorieuse();

} catch (FinDuMondeException fdme) {
  // ...
}
```

EXEMPLE

b étant un entier, lorsqu'une valeur illégale est tentée, une exception **InputMismatchException** est créée automatiquement et le bloc catch est exécuté.

Pour rappel, c'est une exception qui n'est pas capturé automatiquement par Java et donc qui nous incombe de traiter.

Ici, la demande de saisie s'effectue avec des String et donc si l'utilisateur envoie autre choses que des nombres alors l'addition ne sera pas possible, générant une exception de type InputMismatchException

```
import java.util.Scanner;
import java.util.InputMismatchException;

public class Test {
    public static void main(String args[]) {
        int a, b;
        Scanner clavier = new Scanner(System.in);

        try {
            System.out.print("Saisir a : ");
            a = clavier.nextInt();

            System.out.print("Saisir b : ");
            b = clavier.nextInt();

            System.out.println("a+b = " + (a + b));
        } catch (InputMismatchException e) {
            System.out.println(e);
        }

        // fermer les ressources
        clavier.close();
    }
}
```

GESTION D'EXCEPTIONS EN JAVA

Multiples blocs catch

Vous pouvez placer autant d'instructions que nécessaire dans un bloc try et intercepter autant d'exceptions que vous le souhaitez.

Si vous essayez plusieurs instructions, seule la première instruction générant une erreur lève une exception.

Dès que l'exception se produite, la logique est transférée vers le bloc catch, ce qui laisse le reste des instructions du bloc try non exécutées.

Exemples

Exemple 1 : catch générique

```
try{
    // traitements
}
catch(Exception e){
    System.out.println(e);
}
```

Exemple 2 : catch spécifique

```
try{
    // traitements
}
catch(ArithmeticException, InputMismatchException e)
{
    System.out.println(e);
}
```

LE BLOC FINALLY

Le code dans un bloc finally s'exécute que le bloc try précédent identifie une exception ou non. En règle générale, vous utilisez un bloc finally pour effectuer des tâches de nettoyage qui doivent être effectuées, que des exceptions se soient produites ou non.

```
try
{
    // instructions
}
catch(Exception e) {
    // actions si une exception a été lancée
}
finally
{
    // nettoyage
}
```

```
java.io.FileReader reader = new java.io.FileReader(filename);
try {
  int nbCharRead = 0;
  char[] buffer = new char[1024];
  StringBuilder builder = new StringBuilder();
  // L'appel à reader.read peut lancer une java.io.IOException
  while ((nbCharRead = reader.read(buffer)) >= 0) {
    builder.append(buffer, 0, nbCharRead);
  }
  // le retour explicite n'empêche pas l'exécution du block finally.
  return builder.toString();
} finally {
  // Ce block est obligatoirement exécuté après le block try.
  // Ainsi le flux de lecture sur le fichier est fermé
  // avant le retour de la méthode.
  reader.close();
}
```

TRY-WITH-RESSOURCES

La gestion des ressources peut également être réalisée par la syntaxe du try-with-resources.

Après le mot-clé try, on déclare entre parenthèses une ou plusieurs initialisations de variable.

Ces variables doivent être d'un type qui implémente l'interface AutoCloseable ou Closeable.

Ces interfaces ne déclarent qu'une seule méthode : close.

Le compilateur ajoute automatiquement un bloc finally à la suite du bloc try pour appeler la méthode close sur chacune des variables qui ne valent pas null.

```
try (java.io.FileReader reader = new java.io.FileReader(filename)) {
  int nbCharRead = 0;
  char[] buffer = new char[1024];
  StringBuilder builder = new StringBuilder();
  while ((nbCharRead = reader.read(buffer)) >= 0) {
    builder.append(buffer, 0, nbCharRead);
  }
  return builder.toString();
}
```

PROPAGATION D'UNE EXCEPTION

Throws :

Si une méthode lève une exception mais qu'une méthode différente interceptera, vous devez créer une clause throws suivi du type d'exception dans l'entête de la méthode. (spécification d'exception)

```
public class listPrix {
    private static final double[] prix = {15.99, 27.88, 34.56, 45.89};
    public static void afficherPrix(int elem) throws IndexOutOfBoundsException
    {
        System.out.println("le prix est : " + prix[elem]);
    }
}
```

- Si vous générez **une exception vérifiée** à partir d'une méthode, vous devez effectuer l'une des opérations suivantes soit :
  - Attraper l'exception "catch" dans la méthode.
  - Spécifier l'exception dans la clause throws dans l'en-tête de la méthode.
- Si vous écrivez une méthode avec une clause throws dans l'en-tête, toutes méthodes utilisant votre méthode doit effectuer l'une des opérations suivantes soit :
  - Intercepter et gérer l'exception possible.
  - Déclarer l'exception dans sa clause throws. La méthode appelée peut alors renvoyer l'exception à une autre méthode qui pourrait l'attraper ou la lancer à nouveau.
- Si vous écrivez une méthode qui lève explicitement **une exception vérifiée** qui n'est pas interceptée dans la méthode, Java requiert que vous utilisiez la clause throws dans l'en-tête de la méthode.
  - Vous incluez la clause throws dans l'en-tête de méthode afin que les applications qui utilisent vos méthodes soient informées du risque d'exception.

GESTION D'EXCEPTIONS EN JAVA

Créez vos propres classes d'exception

Java : 40 catégories d'exceptions mais on ne peut pas tout prédire.

➢ La solution : Création de sa propre classe d'exceptions.

Votre classe **MaClasseException** doit étendre la classe Exception

La classe Exception

4 constructeurs :

- **Exception()** - Construit un nouvel objet Exception avec la valeur null comme message de détail.
- **Exception(String message)** - Construit un nouvel objet Exception avec le message de détail spécifié.
- **Exception(String message, Throwable cause)** - construit un nouvel objet Exception avec le message de détail spécifié et la cause.
- **Exception(Throwable cause)** - construit un nouvel objet Exception avec la cause spécifiée et un message détaillé de cause.toString(), qui contient généralement la classe et le message détaillé de cause, ou null si l'argument de la cause est null.

EXEMPLE

Création d'une classe CompteException qui contient une seule instruction qui transmet la description d'une erreur au constructeur Mère Exception. Ce message sera récupéré au travers de la méthode getMessage() avec un objet CompteException.

Conseils :

Vous ne devez pas créer un nombre excessif de types d'exception spéciaux pour vos classes, en particulier si l'environnement de développement Java contient déjà une classe Exception qui interceptera l'erreur.

Toutefois, lorsque cela est approprié, les classes Exception spécialisées constituent un moyen élégant de gérer les situations d'erreur.

```
class CompteException extends Exception {
    public CompteException() {
        super("le solde de votre compte est négatif");
    }
}
```

```
public class Gestion {
    public Gestion(double salaire) throws CompteException {
        if (salaire < 0) {
            // lever une exception
            throw (new CompteException());
        }
    }
}
```

```
public class Test {
    public static void main(String args[]) throws CompteException {
        Scanner clavier = new Scanner(System.in);
        System.out.print("Saisir votre salaire horaire : ");
        double salaire = clavier.nextDouble();
        try {
            Gestion cpt = new Gestion(salaire);
            System.out.println("le salaire est " + salaire);
        } catch (CompteException e) {
            System.out.println("Erreur : " + e.getMessage());
        }

        // sinon
        clavier.close();
    }
}
```

EXEMPLE DE GESTION DES EXCEPTIONS
PAR L'EXEMPLE

```java
public static void main(String[] args) {
    // TODO Auto-generated method stub

    DAO<Login> loginDAO = new LoginDAO();
    Login login;

    try {
        login = new Login(1, "Formateur", "@form");

        if ( loginDAO.create(login) ) {
            System.out.println("Login enregistré");
        } else {
            System.out.println("login non enregistré");
        }

    } catch (ReportException e) {
        // TODO Auto-generated catch block
        e.printStackTrace();
    }
eTogether...

}
```

Comme c'est dans notre Main que nous avons positionnés le try … catch, le catch capture l'exception et la traite selon nos instructions (affichage, log etc…)

1

```java
public class Login {

    private int log_user;
    private int pro_id;
    private String log_nom;
    private String log_pass;

    /**
     * Constructeur
     * @param pro_id
     * @param log_nom
     * @param log_pass
     * @throws ReportException
     */
    public Login(int pro_id, String log_nom, String log_pass) throws ReportException {
        super();
        setPro_id(pro_id);
        setLog_nom(log_nom);
        this.log_pass = log_pass;
    }
CodeTogether...
```

2

7

Le constructeur renvoie l'exception qui a été levée à celui qui l'a appelé. Ceci du fait que le constructeur dans sa déclaration de méthode contient throws ReportException

TESTS SUR MES SETTER de la saisie avec possibilité de déclencher une Exception

3

```java
/**
 *
 * @param log_nom
 * @throws ReportException
 */
public void setLog_nom(String log_nom) throws ReportException {

    if (log_nom == null || log_nom.isEmpty() ) {
        throw new ReportException("Problème de Saisie : Le nom ne peut être vide");
    }
    this.log_nom = log_nom;
}
```

```java
/**
 *
 * @return
 */
public String getLog_pass() {
    return log_pass;
}
```

5

En cas de non-respect des règles, alors je lève une exception en la créant par Throw new ReportException(…) classe que j'ai créé pour gérer mes propres messages d'erreurs.

Cette exception créée est alors "lancée" par le setter à l'appelant donc ici le constructeur

6

4

```java
/**
 *
 * @param log_pass
 * @throws ReportException
 */
public void setLog_pass(String log_pass) throws ReportException {
    if (log_pass == null || log_pass.isEmpty() ) {
        throw new ReportException("Problème de Saisie : Le mot de passe ne peut être vide");
    } else if (!checkLogPass(log_pass) ) {
        throw new ReportException(
                "Problème de Saisie : les critères du mot de passe ne sont pas bonnes");
    }

    this.log_pass = hashPass(log_pass);
}
```