Exceptions

Table des matières

Définition

Nous avons déjà eu l'occasion de rencontrer des exceptions lors de l'exécution de nos programmes : `ArrayIndexOutOfBoundsException`, `ArithmeticException`, `NumberFormatException`, `NullPointerException`, …

Les exceptions sont des classes comme les autres, si ce n'est qu'elles héritent toutes de la classe `Exception`.

La première chose à savoir faire est de :

Traiter les exceptions

Quand traiter ?

Avant de traiter une exception, il faut toujours se poser la question de savoir si on est à un endroit où on est en mesure de le faire. Si ce n'est pas le cas on laissera l'appelant le faire, voire même l'appelant de celui-ci et ainsi de suite, jusqu'à `main()`. Si à ce moment, l'exception n'est pas traitée le programme se plantera en affichant le contenu de la pile des appels.

Quand faut-il traiter une exception ? Il n'y a pas de réponse globale à ce problème. On traitera l'exception dès qu'on est à un endroit où on sait comment le faire. S'il faut afficher une erreur on ne la traitera pas avant d'être dans main ou dans une classe faisant partie d'une interface graphique.

Si, à un moment, on sait que les circonstances sont telles que l'exception à traiter ne peut pas avoir lieu, on est au bon endroit pour la traiter.

Comment traiter ?

Pour traiter des exceptions on emploie l'instruction `try` … `catch` … `finally`.

```
String s = ...;
int n = 0;
try {
      n = Integer.parseInt(s);
} catch (NumberFormatException nfe) {
      System.out.println("il fallait un nombre entier");
      System.exit(1);
}
```

On place dans le bloc `try` ce qu'on essaie de faire et qui pourrait provoquer une exception.

On met dans le bloc `catch` le traitement à effectuer en cas d'erreur.

Normalement, si une exception survient, l'instruction en cours est interrompue et le contrôle passe immédiatement au bloc `catch` permettant de traiter l'exception survenue. Après le programme se poursuit normalement et le contrôle passe à la fin du `try` … `catch` … `finally`. Sauf bien sûr, si le programme est terminé par un `System.exit()` ou si le contrôle est redirigé ailleurs par un `return` ou un `break` par exemple.

Si un traitement doit être effectué dans tous les cas, on peut donc le placer après le bloc `try` … `catch`. Mais si dans le bloc `catch`, on sort de la méthode par un `return` ou d'une boucle englobant le `try` … `catch` par un `break`, ce qui suit le `try` … `catch` ne sera pas exécuté. Si un traitement commun doit être fait dans ce cas, on utilisera le bloc `finally`.

Attention, si un `System.exit()` a lieu, le bloc `finally` ne sera pas exécuté.

```
String fileName = ...;
BufferedReader f = null;
int n = 0;
try {
   try {
      f = new BufferedReader(new FileReader(fileName));
      n = Integer.parseInt(f.readLine());
   }
```

```
      catch (NumberFormatException nfe) {
         System.out.println("il fallait un nombre entier");
         return;
      }
      catch (FileNotFoundException fnfe) {
         System.out.println(fnfe.getMessage());
         return;
      }
   finally {
      if (f != null)
         f.close();
   }
}
   catch (IOException ioe) {
      System.out.println(ioe.getMessage());
      return;
   }
```

Plusieurs blocs `catch` sont permis, comme on peut le voir ci-dessus, mais on est obligé de respecter la hiérarchie des classes d'exceptions : un code comme ci-dessous est interdit car `FileNotFoundException` est une sous-classe d'`IOException`.

```
    catch (IOException ioe) {
       System.out.println(ioe.getMessage());
       return;
    }
    catch (FileNotFoundException fnfe) {
       System.out.println(fnfe.getMessage());
       return;
    }
```

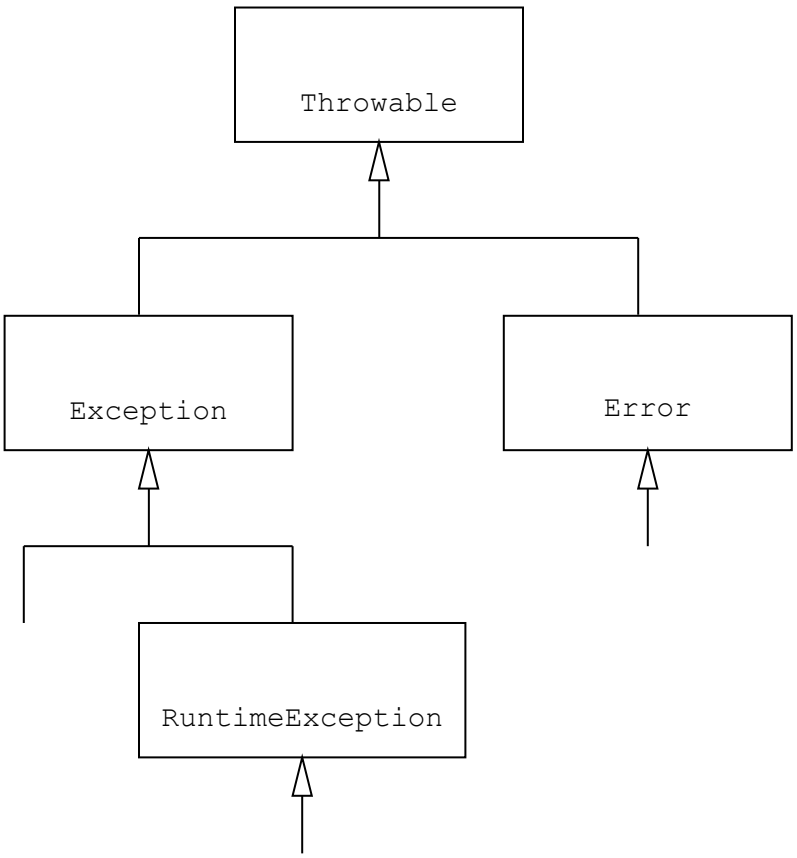
Le compilateur ne vous permettra pas cette construction, il émettra un message d'erreur du style : `exception java.io.FileNotFoundException has already been caught`. On écrira donc plutôt :

```
    catch (FileNotFoundException fnfe) {
       System.out.println(fnfe.getMessage());
       return;
    }
    catch (IOException ioe) {
       System.out.println(ioe.getMessage());
       return;
    }
```

Postposer le traitement d'une exception :

Si on ne traite pas une exception immédiatement le traitement devra se faire dans l'appelant. Si lui ne la traite pas, dans son appelant et ainsi de suite jusqu'à vider la pile des appels. A ce moment le programme termine brutalement et affiche le contenu de la pile des appels depuis le moment où l'exception s'est produite.

Si on désire postposer le traitement de l'exception, que faut-il faire? Tout dépend du type d'exception. Java distingue en effet deux types d'exceptions : les "checked exceptions" et les "unchecked exceptions". La hiérarchie des exceptions se présente ainsi :



Les classes `Error`, `RuntimeException` et toutes leurs sous-classes sont des unchecked exceptions (exemples : `ArithmeticException`, `ArrayIndexOutOfBoundsException`, `NumberFormatException`, `NullPointerException`,…). Les autres sont checked (exemples : `IOException`, `FileNotFoundException`, …).

Si une exception est checked et qu'on désire postposer le traitement de l'exception, il est obligatoire d'annoncer dans l'entête de la méthode que cette exception pourrait se produire. On utilise pour cela une clause `throws`.

```
public void méthode() throws IOException {
```

Plusieurs exceptions peuvent être postposées :

```
public void méthode() throws IOException,CloneNotSupportedException{
```

Pour les checked exceptions, il est donc obligatoire de les traiter (à l'aide de `try … catch`) ou de les postposer à l'aide de `throws`. Ce n'est par contre pas obligatoire pour les unchecked exceptions.

Exceptions et héritage :

Si une exception est postposée, toute exception d'une sous-classe de celle-ci est également postposée, en vertu du principe de substitution. Ainsi si dans `méthode()` survient une `FileNotFoundException`, celle-ci ne doit pas nécessairement être traitée puisque la clause throws `IOException` postpose aussi une `FileNotFoundException`.

D'autre part, la redéfinition d'une méthode a une influence sur les exceptions indiquées dans la clause `throws`. Si une méthode redéfinie doit postposer une exception `X` via une clause `throws X`, la méthode parent devra également postposer l'exception `X` via une clause `throws X` ou `throws Y` où `Y` est une superclasse pour `X`.

Autrement dit, une méthode redéfinie n'a pas le droit de postposer des checked exceptions qui ne le serait pas par la méthode qu'elle redéfinit dans la classe parent.

Lancer une exception : l'instruction `throw`.

Dans une méthode on peut décider de lancer soi-même une exception en utilisant l'instruction `throw` :

```
if (i >= tableau.length)
   throw new ArrayIndexOutOfBoundsException(i + " est trop grand");
```

Le paramètre passé au constructeur de l'exception pourra être récupéré dans un bloc `catch` via la méthode `getMessage()` héritée de la classe `Exception`.

Il va de soi que, si on lance une exception, c'est parce qu'on n'est pas à même de la traiter. Si cette exception est checked, il faudra donc l'indiquer dans la clause `throws` de la méthode.

Définir soi-même ses propres exceptions

Ecriture d'une classe d'exception :

Il est très aisé de définir ses propres exceptions : Il suffit de définir une sous-classe `d'Exception`.

Si on ne désire pas passer de message d'erreur lors du lancement de cette exception, on peu se contenter de faire :

```
public class MonException extends Exception {
}
```

Bien souvent, on désire utiliser la possibilité de passer un message d'erreur lors du lancement. Pour cela, on a besoin d'un constructeur admettant une String en paramètre. Il faut alors définir l'exception comme suit, puisque si on écrit un constructeur, Java ne génère plus de constructeur par défaut :

```
public class MonException extends Exception {
      public MonException() {
            super();
      }
      public MonException(String message) {
            super(message);
      }
}
```

C'est ce qu'on fait généralement.

Il est bien sûr possible de définir d'autres méthodes dans la classe `MonException`, mais c'est rarement utile.

Les méthodes héritées de la classe `Exception` :

`String getMessage();` affiche le message d'erreur passé lors du lancement de l'exception

`String toString();` renvoie une String formée du nom de la classe d'exception, de `": "` et du message d'erreur (s'il y en a un).

`void printStackTrace();` affiche le contenu de la pile des appels lors du lancement de l'exception.

`void printStackTrace(PrintStream p);` affiche le contenu de la pile des appels lors du lancement de l'exception, sur le `PrintStream p`.

`void printStackTrace(PrintWriter p);` affiche le contenu de la pile des appels lors du lancement de l'exception, sur le `PrintWriter p`.

Exemple

Donnons une version orientée exceptions des classes de comptes vues précédemment. Ici les méthodes ne renverront plus un booléen pour indiquer leur réussite mais provoqueront une exception en cas d'échec :

```
public abstract class Compte {

    private final String titulaire;
    private final String numéroDeCompte;
    private double solde = 0;

    public Compte( String titulaire, String numéroDeCompte ) {
        this.titulaire = titulaire;
        this.numéroDeCompte = numéroDeCompte;
```

```
}

public Compte( String titulaire, String numéroDeCompte,
                                            double montantInitial ) {
    this(titulaire, numéroDeCompte);
    this.solde = montantInitial;
}

private void transaction( double montant ) {
    this.solde += montant;
}

public final void  depot( double montant ) throws
                                            MontantNegatifException {
    if ( montant > 0 ) {
        transaction(montant);
        return;
    }
    throw new MontantNegatifException("depot : " + montant + " BEF");
}


public final void retrait( double montant) throws
                SoldeInsuffisantException, MontantatifException {
    if ( montant > 0) {
        if (getSolde() - montant >= getSeuil()){
            transaction(-montant);
            return;
        }
        throw new SoldeInsuffisantException("retrait : solde = " +
                 getSolde() + " BEF, seuil = " + getSeuil() + " BEF");
    }
    throw new MontantNegatifException("retrait " + montant + " BEF");
}

public void virement(double montant, Compte c)
        throws SoldeInsuffisantException, MontantNegatifException,
              VirementInterditException {
    throw new VirementInterditException(
              "virement interdit sur un compte" + getType());
}

public final String getTitulaire() {
    return this.titulaire;
}

public final String getNuméroDeCompte() {
    return this.numéroDeCompte;
}
```

```
    public final double getSolde() {
        return this.solde;
    }

    public double getSeuil() {
        return 0;
    }

    public void setSeuil(double seuil)
                              throws SeuilNonModifiableException {
        throw new SeuilNonModifiableException();
    }

    public abstract String getType();
}
```

Comme on peut le voir la méthode `virement() throws` bien plus d'exception que la seule qui s'y produit. C'est parce que ces exceptions seront lancées dans la redéfinition de cette méthode dans la sous-classe ci-après.

```
public final class CompteAVue extends Compte {
    public static final String VUE = " à vue";

    private double seuil = 0;

    public double getSeuil(){
        return this.seuil;
    }

    public void setSeuil( double seuil ){
        this.seuil = seuil;
    }

    public String getType() {
        return CompteAVue.VUE;
    }

    public void virement(double montant, Compte c) throws
               SoldeInsuffisantException, MontantNegatifException {
        retrait(montant);
        c.depot(montant);
    }

    public CompteAVue( String titulaire, String numéroDeCompte ){
        super(titulaire, numéroDeCompte);
    }
```

```
    public CompteAVue( String titulaire, String numéroDeCompte,
                                                double montantInitial ){
        super(titulaire, numéroDeCompte, montantInitial);
    }

    public CompteAVue( String titulaire, String numéroDeCompte,
                                double montantInitial, double seuil ){
       super(titulaire, numéroDeCompte, montantInitial);
       this.seuil = seuil;
    }
}

public class CompteDEpargne extends Compte {
    public static final String EPARGNE = " d'épargne";

    public String getType() {
        return CompteDEpargne.EPARGNE;
    }

    public CompteDEpargne( String titulaire, String numéroDeCompte ){
        super(titulaire, numéroDeCompte);
    }

    public CompteDEpargne( String titulaire, String numéroDeCompte,
                                               double montantInitial ){
        super(titulaire, numéroDeCompte, montantInitial);
    }
}
```

Les classes d'exceptions sont triviales :

```
public class MontantNegatifException extends Exception {
    public MontantNegatifException() {
        super();
    }

    public MontantNegatifException(String message) {
        super(message);
    }
}

public class SeuilNonModifiableException extends Exception {
    public SeuilNonModifiableException() {
        super();
    }

    public SeuilNonModifiableException(String message) {
        super(message);
    }
}

public class SoldeInsuffisantException extends Exception {
    public SoldeInsuffisantException() {
        super();
    }

    public SoldeInsuffisantException(String message) {
        super(message);
    }
}

public class VirementInterditException extends Exception {
    public VirementInterditException() {
        super();
    }

    public VirementInterditException(String message) {
        super(message);
    }
}
```