

2i002 Exceptions Personnalisées

Vincent Guigue - vincent.guigue@lip6.fr



Le cours est inspiré de sources diverses: L. Denoyer, F Peschanski, E. Chailloux...

Une **exception** est une rupture de calcul.

Elle est utilisée :

- pour éviter les erreurs de calcul
 - division par zéro
 - accès à la référence `null`
 - ouverture d'un fichier inexistant
 - ...
- comme style de programmation

En Java une exception est un objet

DÉFINIR SES EXCEPTIONS PERSONNALISÉES

Idée

Une exception est un objet... On peut définir ses propres exceptions

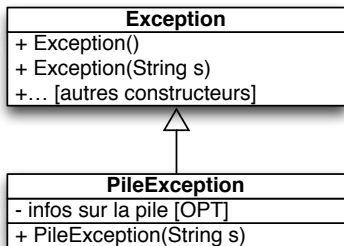
```
1 // Exception de base pour tous les problèmes de pile
2 public class PileException extends Exception {
3     public PileException(String message) {
4         super("Problème de pile: " + message);
5     }
6 }
```

On peut même définir une hiérarchie d'exception :

```
1 // Exception spécifique pour la pile pleine
2 public class PilePleineException extends PileException {
3     public PilePleineException() {
4         super("Pile pleine");
5     }
6 }
```

- Utile pour le transport d'informations (Attributs de l'exception personnalisée)

RÉCUPÉRATION DES INTERRUPTIONS AVEC DES HIÉRARCHIES D'EXCEPTIONS 1/3

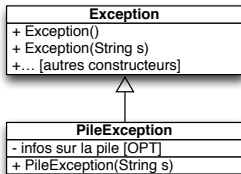


Une `PileException` EST UNE `Exception`... Le principe de subsomption s'applique.

```
1 try{
2     // Pile p vide...
3     System.out.println(p.depiler()); // lève une PileException
4 }
5 } catch (Exception e){ // PileException est attrapée
6     // car c'est UNE Exception
7     ...
8 }
```

RÉCUPÉRATION DES INTERRUPTIONS AVEC DES HIÉRARCHIES D'EXCEPTIONS 2/3

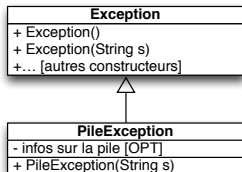
- Possibilité de raffiner le traitement en fonction du type de l'exception



Possibilité de faire plusieurs `catch` : traitement séquentiel, le premier qui correspond est utilisé (et les autres non)

```
1 try{
2     // Pile p vide...
3     System.out.println(p.depiler()); // lève une PileException
4 }
5 } catch (PileException e){ // PileException est attrapée ici
6     ...
7 } catch (Exception e){ // On ne passe pas ici
8     ...
9 }
```

RÉCUPÉRATION DES INTERRUPTIONS AVEC DES HIÉRARCHIES D'EXCEPTIONS 3/3

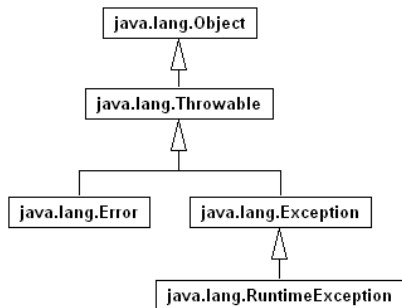


- On définit obligatoirement les Exception les plus spécialisée en premier
- Sinon erreur de compilation
- Car code non accessible

Le programme suivant ne compile pas...

```
1 try{
2     // Pile p vide ...
3     System.out.println(p.depiler()); // lève une PileException
4 }
5 }catch (Exception e){
6     ...
7 }catch (PileException e){ // Code non accessible
8     ...
9 }
```

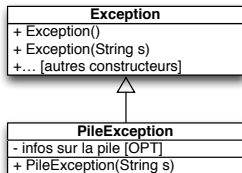
DISTINCTION ENTRE EXCEPTION ET RUNTIMEEXCEPTION



Les **Exception** susceptibles d'être levées dans une méthode **doivent être déclarée** dans la signature de la méthode via le mot clé `throws`.

Les **RuntimeException** sont des Exception particulières qui **ne requièrent pas cette déclaration**.

DÉCLARATION D'UNE EXCEPTION



Dans le cas d'un héritage à partir d'Exception, la déclaration est obligatoire :

- Sinon le code ne compile pas
- Signature de la méthode :

```
public <Retour> fonction(<args>) throws <typeException>
```

Sur un exemple :

```
1 public void empiler(int item) throws PileException {
2     if(niveau >= capacite)
3         throw new PileException("La pile est pleine : impossible d'
4         items[niveau++] = item;
5 }
```


Attention

Si l'exception est traitée au niveau local, la fonction n'est pas susceptible de la lever : pas de déclaration dans ce cas.

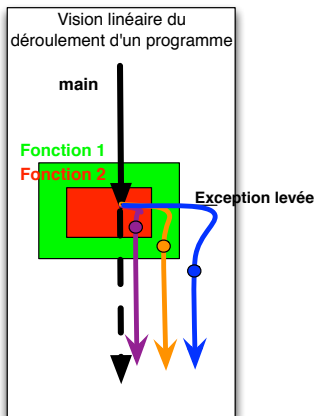
```
1 // avec MonException extends Exception
2
3 public void maFonction(Type args) { // pas de déclaration
4     ...
5     try{
6         ... // traitements
7         if(test)
8             throw new MonException("MonMessage");
9         ... // reste des traitements
10    }catch(MonException e){
11        ... // faire qqch
12    }
13 }
```

Dans ce cas, il est impossible que maFonction soit interrompue par une MonException non traitée

USAGE : MÉTHODE CAUSANT UNE INTERRUPTION

Idée (logique)

Si une méthode a utilise une méthode b susceptible de lever l'Exception Exc, alors a est susceptible de lever Exc également.



- 1 Exception traitée dans Fonction2 : aucune déclaration nulle part
- 2 Exception traitée dans Fonction1
 - Fonction2 doit déclarer `throws MonException`
 - Délégation du traitement
 - Fonction1 ne déclare rien
- 3 Exception traitée dans main
 - Les fonctions déclarent `throws`

NB : suivant les cas, on ne reprend pas l'exécution au programme au même endroit... Comportement différent.

CASCADE D'EXCEPTION

Une Exception peut déclencher d'autres Exception

Mécanisme : Exception $e \rightarrow$ catch \rightarrow Exception $e2$

- Pour certains cas particuliers
- Traduction d'exceptions générales vers les exceptions d'un projet particulier

```
1 public void maFonction throws DepassementCapaciteException{
2     try{
3         // gestion d'une structure tabulaire de données
4         // risque de dépassement d'indice
5         ...
6     }catch(IndexOutOfBoundsException e){
7         // traduction d'une Exception Java standard
8         // -> Exception perso
9         throw new DepassementCapaciteException();
10    }
11 }
```

Idée

Simplifier la syntaxe des Exception

Instruction assert

- Syntaxe : `assert expr1 [: expr2]`
 - `expr1` est une expression booléenne
 - `expr2` de type quelconque
- Exécution :
 - `expr1` est évaluée
 - si `false` \Rightarrow une `AssertionError` est lancée
 - `expr2`, convertie en chaîne, est utilisée comme message

`java.lang.Object`
 extended by `java.lang.Throwable`
 extended by `java.lang.Error`
 extended by `java.lang.AssertionError`

Alternative avec des exceptions :

```
1 if (! expr1) {throw new MyException();}
```

UTILISATIONS [E. CHAILLOUX]

```
1 if (x == 0) {
2     ...
3 } else { // x est égal à 1 ...
4     assert x == 1 : x;
5 }
6 ...
7 switch(x) {
8     case -1 : return INF;
9     case 0  : return EGAL;
10    case 1  : return SUP;
11    default : assert false; // impossible si x=-1, 0 ou 1
12 }
13 ...
14 public int f f() {
15     for (...) {
16         ... return ...;
17     }
18     // on ne passe pas ici
19     assert false;
20     return 0;
21 }
```