POLYTECH
PARIS-SUD

Année 2018-2019

Polytech'Paris-Sud –
Département Informatique

« Les exceptions en Java »

Frédéric Voisin
Département Informatique

Pourquoi des « exceptions » ?

Gérer des « situations exceptionnelles »

- Cas d'erreurs liées
  - à l'environnement (JVM : `OutOfMemoryException`)
  - à des erreurs de programmation (référence nulle, indice incorrect)
  - à une application/classe particulière

    `next()` sur un itérateur épuisé (`NoSuchElementException`)

    `new MesDate(2013, 2, 29)  // 29 février 2013 : date incorrecte`
- ... ou tout autre événement impliquant une rupture (partielle) du déroulement normal du programme

Mécanisme utile

- si on ne sait pas que faire à l'endroit où le problème est découvert (le signaler à l'appelant)
- Si on ne sait pas forcément à quel niveau le problème pourra être traité (propagation automatique jusqu'à récupération)

Pourquoi des « exceptions » ? (suite)

Signaler explicitement des cas « exceptionnels » ?

- Éviter de procéder par « retour de valeur particulière » qui ne sont pas toujours testées par l'appelant
- Éviter que les cas « erronés » soient ignorés : une exception non traitée interrompt le programme
- Ne pas obscurcir les cas « normaux » avec les cas « exceptionnels » (tester toute référence à `null` avant usage ?)

Autoriser des traitements de rattrapage arbitrairement complexes : correction locale et/ou signalement aux appelants.

Autoriser des traitements sélectifs : chaque méthode décide quels cas exceptionnels elle sait traiter et quels cas doivent être propagés.

« Gérer » une exception ?

Signaler une situation exceptionnelle (« lever une exception »)

`throw` *instance d'exception ;*

Rattraper une exception : récupération d'une situation exceptionnelle, **si** on est capable de la traiter !

```
try { liste d'instructions }      // bloc d'instructions protégé
catch(ClasseException1 variable){ traitement1 }
catch(ClasseException2 variable){ traitement2 }
```

Propager explicitement ou implicitement une exception qu'on ne sait pas traiter complètement

Souvent on combine traitement local + propagation à l'appelant (la même exception ou une autre !)

Annoncer syntaxiquement ce qu'on est susceptible de lever

Qu’est-ce qu’une (classe) d’exception ?

- Une exception = une instance d’une classe membre de la sous-hiérarchie de racine `Throwable`

  `Error, Exception, RunTimeException, IOException, ...`

- Une classe d’exception est une classe « ordinaire », avec constructeurs, attributs, méthodes...

  Méthodes héritées de `Throwable` :

  `getMessage(), toString()`

  `printStackTrace()` *// historique de propagation*

  \+ une **description du contexte d’exécution** (trace d'appels) au moment de la levée de l’exception

- Les règles de typage classiques s’appliquent !

- On peut organiser ses exceptions en hiérarchie.

Un exemple

```
public class MesDates {
  private int annee, quantieme;
  public MesDates(int an, int quant) throws ErreurDate {
    if (quant >= 366 || (quant == 366 && ! bissextile(an)) {
        throw new ErreurDate("quantieme trop grand");
    } else ...
  }

  public MesDates(int a, int m, int j) throws ErreurDate {
      this(a, quantieme(a, m, j));
 }

  public static MesDates aujourdhui() {
   return new MesDates(...);      // OK ou pas ?
  }
}
```

Exemple de classe d'exception

```
public class ErreurDate extends Exception {

 /* prend une String en paramètre pour fournir une indication sur la
  * nature du problème. La chaîne pourra être récupérée en
  * appliquant getMessage à l'exception.
  */
   public ErreurDate(String msg) {
      super(msg);
   }

  /* Version sans argument : getMessage() renverra null */
   public ErreurDate() { super() ; }
}
```

On peut avoir aussi des variables d'instances ou des méthodes !

class java.lang.**Throwable** *sous-classe directe de `Object` !*

class java.lang.**Error** ***Difficilement rattrapables !***

class java.lang.VirtualMachineError

class java.lang.OutOfMemoryError

…

class java.lang.Exception

class java.lang.**RuntimeException** ***Trop fréquentes pour qu'on oblige***

class java.lang.ArithmeticException ***à les signaler à l'avance !***

class java.lang.**ClassCastException** ***Elles peuvent survenir partout***

class **java.lang**.IllegalArgumentException

class java.lang.IllegalThreadStateException

class java.lang.NumberFormatException

class java.lang.**IndexOutOfBoundsException**

class java.lang.ArrayIndexOutOfBoundsException

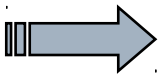
class java.lang.StringIndexOutOfBoundsException

class java.lang.NegativeArraySizeException

class java.lang.**NullPointerException**

class java.**util**.**NoSuchElementException**

class java**.io.IOException**



Ici prochainement, vos exceptions !

Spécification des exceptions

Une méthode doit déclarer explicitement les exceptions qu'elle est susceptible de lever :

```
public void f(…) throws E1, E2 { … }
```

- le corps de `f` ne peut lever que les exceptions de classe (au sens large) `E1` ou `E2`
- `f` doit soit rattraper, soit propager (conformément à son en-tête) les exceptions des méthodes qu'elle appelle.

L'appelant de `f` sait donc contre quelles exceptions se prémunir…

Toute redéfinition de `f` devra respecter ce profil… mais peut lever dynamiquement une exception plus spécifique (sous-classe)

La règle s'applique aussi à la méthode `main` !

Spécification des exceptions (suite)

A titre dérogatoire, il est facultatif de spécifier la levée des exceptions des hiérarchies `Error` et `RunTimeException`

```
public void f(…) {
      …
}
```

ne peut donc lever/propager que des exceptions de classe `Error` ou `RunTimeException`

```
public void f(…) throws Exception {
      …
}
```

peut lever des exceptions arbitraires mais complique trop la tâche de ses appelants

Normalement on ne travaille pas directement au niveau de `Exception`.

Spécification des exceptions (exemple)

```
public class MesDate {
  public MesDate(int an, int quantieme) throws ErreurDate {
    // exception pour Date(2003, 366) ou Date(2003, -1) !
    …
  }

  // KO : ErreurDate n'est ni rattrapée, ni « déclarée »
  // Incorrect, même si ici l'exception ne peut pas se produire !
  public static MesDate aujourdhui() {
     Calendar date = Calendar.getInstance();
     return new MesDate(date.get(Calendar.YEAR),
                        date.get(Calendar.DAY_OF_YEAR));
  }
}
```

Comment corrige-t-on ?

Levée d'exceptions

Levée d'exception = rupture du déroulement normal du programme :

- arrêt du déroulement normal du programme pour rechercher **dynamiquement** un traite-exception adapté
- Ce traitement **remplace** le traitement normalement effectué, sans possibilité de retour...

Levée implicite par la JVM ou une classe existante:

```
Object[] tab = new Object[0];
Integer I; … ; System.out.println(Integer.intValue(I));
new Arraylist().iterator().next();
```

Levée explicite par le code du programmeur:

```
throw new monException(...);
```

Récupération d'exceptions

`try` { f_1(); … f_N(); } // *bloc protégé suivi des traites-exceptions*

`catch` (E_1 v) { *instructions* }

… // les E_i *sont des classes d'exception,* v *référence l'instance reçue.*

`catch` (E_p v) { *instructions* }

Les `catch` sont examinés dans l'ordre jusqu'au premier qui correspond (modulo héritage).

- Le compilateur vérifie que tout `catch` est accessible.
- Le traite-exception **remplace** la fin du traitement du bloc `try`, avec les mêmes contraintes : type de la valeur retournée, exceptions levables, etc.
- A vous de conserver assez d'information sur le contexte pour traiter l'exception : quel appel f_i() a levé l'exception, i.e. quelle partie du traitement a bien pu être effectuée…
- En l'absence de `catch` adapté, **propagation automatique** de l'exception au bloc englobant

Récupération d'exceptions (suite)

Un `catch` peut lui-même lever une exception

la même :

```
catch (E1 e) {
     traitement local
     throw e;
}
```

ou une autre :

```
   catch (E1 e) {
     traitement local éventuel
      throw new E2(...);
   }
```

Les `catch` peuvent englober des blocs `try/catch` **mais** une exception levée dans un `catch` n'est **jamais** rattrapable dans le même bloc `try/catch`

```
try { … }
catch (E1 e) { throw new E2(); }
catch (E2 e2) { … }
```

Le second `catch` ne rattrapera pas une exception levée dans le premier `catch`, qui sera donc propagée au bloc englobant le `try/catch`.

Éléments méthodologiques : qui rattrape quoi ? (suite)

Il est courant de rattraper une exception pour en lever une autre, éventuellement après un traitement local

Une classe `Intervalle` par jours entiers sur une seule année civile

```
public Intervalle(MesDates d, int q) throws ErreurIntervalle {
  private MesDates debut, fin;
  if (q < 1) throw new ErreurIntervalle("Durée negative ou nulle");
  try { this.deb = d; this.duree = q;
        // lève l'exception si la fin de l'intervalle est en dehors de l'année
         this.fin = new MesDates(d.annee(), d.quantieme() + q);
  catch(ErreurDate e) {
    throw new ErreurIntervalle("Durée trop longue");
  }
}
```

- Une ou plusieurs exception pour les intervalles ?
- Un attribut pour stocker les valeurs associées à l'exception levée?
- Quelle hiérarchie entre les exceptions des intervalles et des dates ?

Récupération d'exceptions (fin)

Attention à la portée des `try/catch` :

```
f(Iterator it) {
  try {
     while (true)
       { it.next(); … }
  }catch (…) { … }
}
```

```
f(Iterator it) {
   while (true)
      try {
             it.next();
       } catch (…) { … }
 }
```

Attention donc au fragment de programme que vous encapsulez dans un try/catch.

Propagation d'exceptions

Recherche dynamique, selon l'imbrication des `try/catch`, d'un traite-exception adapté

- Si on en trouve un : l'exécution continue à partir de ce point
- Sinon, l'exécution du bloc est interrompue et l'exception est signalée chez l'appelant, au point d'appel initial
  - On remonte **dynamiquement** la chaîne de blocs et d'appels de méthodes jusqu'à éventuellement la méthode `main`
  - Si l'exception n'est pas rattrapée : interruption du programme

On ne rattrape pas si on ne sait pas (vraiment) qu'en faire…

On ne rattrape pas si c'est juste pour re-propager **la même** exception

Exceptions et constructeurs

Attention aux exceptions dans les constructeurs :

Dans quel état serait l'instance concernée (initialisation partielle) ?

- Qu'est-ce que l'utilisateur pourrait en faire ?

```
MesDates d = new MesDates(2013, 366); ...
```

Une exception levée dans un constructeur n'est **jamais** récupérable dans un autre constructeur

```
public MesDate(int a, int quant) throws ErreurDate {
  …
}
public MesDate() {                  // Incorrect !
   this(year, today);               // Comment la rattraper ?
}
```

Même problème pour la liaison constructeurs sous-classe/super-classe

Éléments méthodologiques

On évite la levée d'exception ou on la rattrape ?

- Privilégier la correction du code et sa lisibilité
- La gestion des exceptions est un mécanisme coûteux

```
if (o instanceof E) { E monE = (E) o ; … }   vs
try { E monE = (E) o ; … }
catch (ClassCastException e) { … }
```

- On peut parfois s'en servir comme mécanisme de programmation pour revenir directement d'une longue séquence d'appels récursifs.

Une **partie intégrante de la conception** d'une classe

- L'usage de `RuntimeException` amène une fuite d'informations
- Qui lève quoi ? Aussi important que les types des paramètres !
- Quel niveau de détail :
  - Quel contenu pour la classe d'exception (attributs, méthodes) ?
  - Quelle hiérarchie entre les classes d'exceptions de l'application ?

Éléments méthodologiques : qui rattrape quoi ?

On ne rattrape une exception que si on sait quoi en faire.

- Halte aux rattrapages abusifs
- Halte aux messages d'erreur et `printStackTrace` intempestifs

Ex: une pile d'éléments de capacité fixée est pleine. L'utilisateur empile un nouvel élément. Vous faites quoi ?

En phase de mise au point : ne pas rattraper les exceptions qui ne devraient pas se produire (=> facilite la mise au point) ; la suite de l'exécution a-t-elle encore du sens ?

En production : on peut préférer un code plus « robuste ». On rattrape, à un endroit où on sait comment repartir du bon pied ou sortir proprement de l'application.

Éléments méthodologiques (fin)

Qu'est-ce qu'un cas exceptionnel ?

Ex : la méthode boolean `add(Object o)` *de* `Collection`

- Certaines collections admettent les doubles (`ArrayList`)
- D'autres collections n'ont jamais de doubles (`TreeSet, HashSet`)
- Certaines collections ont des contraintes supplémentaires
  - `TreeSet` : les éléments doivent pouvoir être comparés entre eux
  - D'autres refusent d'ajouter `null` comme élément de la collection.
  - D'autres refusent d'implémenter `add` (ou `remove`) !

Toutes sont des implémentations du `add` de `Collection` *et ont donc même* **profil d'exception**.

Quelles exceptions prévoyez-vous ? Pour quels cas de figure ?
A quoi sert le type de retour dans `add` *?*

Éléments méthodologiques : Exemple (fin)

Deux règles de conduite:

1. *Il n'est pas « exceptionnel » d'ajouter un élément à une collection qui le contient déjà, puisque certaines collections l'autorisent !*

2. - *add renvoie `true` si l'ajout est possible et `c` a été modifiée* (soit parce que `o` n'y était pas, soit que la collection accepte les doubles)

3. - *add renvoie `false` si l'ajout est possible et `c` n'a pas eu besoin d'être modifiée* (`o` déjà présent et doubles non autorisés)

Dans les deux cas `o` est bien présent dans la collection après le retour de `add` et, si on est intéressé, on peut savoir si `c` a été modifiée ou pas

Éléments méthodologiques : Exemple (fin)

Deux règles de conduite:

1. …

2. Si `c.add(e)` réussit, `e` doit être dans `c` ! *Donc a*`add` **doit** lever une exception si l'ajout est impossible :

`UnsupportedOperationException` pour celles qui n'implémentent pas `add`

`ClassCastException` pour celles qui ont des contraintes « de type »

`IllegalArgumentException` pour d'autres contraintes sur les éléments

`NullPointerException` : pour celles qui refusent `null` comme élément

`IllegalStateException` : …

Ces exceptions sont dans `RuntimeException` pour ne pas avoir à apparaître explicitement dans l'en-tête de `add`.

=> On peut jouer sur la valeur de retour **et** sur la levée d'exception !

Guide : quel invariant doit-on garantir sur l'effet de la méthode ?

La clause « finally »

```
try { instructions }
catch (E1 e1) { ... } // traites-exceptions optionnels
  ...          ...        ...
finally { instructions }     // clause optionnelle
```

- Rôle : garantir qu'un fragment de code est exécuté dans tous les cas, normaux ou avec des exceptions, rattrapées ou non !

- Précautions d'usage :
  - attention aux doubles traitements
  - attention à ne pas lever d'exceptions dans le bloc `finally` !

- OK aussi pour les boucles (forcer l'exécution d'un fragment de code, aussi bien en cas d'itération, que de **`continue`** ou **`break`**)

- Java 7 : Il existe une version de `try` « avec ressources » pour garantir la « fermeture » propre d'objets (voir aussi les interfaces `AutoCloseable` et `Closeable`)

La clause « finally » : exemple

```
class MonException extends Exception {}
public class TestFinally { // adapté de B. Eckel: "Thinking in Java"
  public static void main(String[] args) {
    int count = 0;
    while (true)
      try {  if (count == 0) { count++; throw new MonException(); }
             else if (count == 2)
                 throw new NullPointerException();
             else { count++; System.out.println("Cas normal"); }
      } catch (MonException e) {
            System.out.println("Dans le traite-exception");
      } finally { System.out.println("Dans le bloc finally");}
 }
}
```

Question : quels sont les affichages réalisés ?

Depuis Java 7

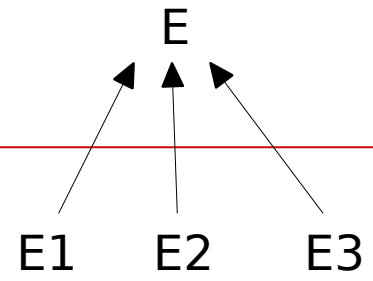
1. Factorisation de traite-exceptions

```
try { … }
catch (E1 | E2 e) { traitement commun aux deux cas }
```

- Dans l'exemple ci-dessus, `e` est implicitement `final`
- Dans le traite-exception on peut faire `throw e;`
  ou toute autre opération compatible avec le type (?) de `e` !

Évite la duplication de code dans des `catch` séparés, sans être obligé de généraliser E1 et E2 en leur superclasse (mais en se restreignant sur `e` aux opérations de cette superclasse).

Depuis Java 7



2. Typage plus précis des exceptions levables.

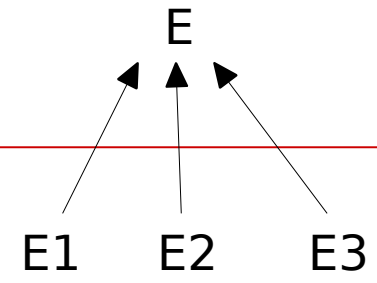
```
public void f(boolean b) throws E1, E2 {
try { if (b) throw new E1(); else throw new E2() ; }
catch (E e) {
    – traitement local –
    throw e;    // conforme au throws
}
```

OK **si** le compilateur peut vérifier que le bloc `try` ne peut lever que `E1` et `E2`, **et si** `e` n'est pas cible d'une affectation dans le `catch`.

*Nouvelles règles pour une **exception re-signalée** par un bloc `catch` :*

- *Le bloc `try` doit pouvor effectivement la lever*
- *Il n'y a pas un `catch` plus général devant*
- *Son type est sous-type ou super-type d'une exception du `catch`.*
- *La clause `throws` est respectée.*

Exemple Java 7




```
public class Test {

  public static void f1() throws E1 { throw new E1(); }
  public static void f2() throws E2 { throw new E2(); }
  public static void test() throws E1, E2 {
    try { f1(); f2(); }
    catch (E e) {
     if (...) e = new E3()   // KO au niveau du throw plus bas
     else e = new E2();      // KO au niveau du throw plus bas
     throw (E1) e;           // OK, compile
     throw (E3) e;           // KO : E3 pas dans le throws
     throw (Exception) e;    // KO : Exception pas levable dans try
     throw e;                // OK
    }
  }
}
```

Se généralise avec l'alternative entre plusieurs exceptions...