Les Exceptions

Rémi Forax

Les exceptions

Les exceptions en Java sont en fait assez compliquées à utiliser et donc très mal utilisé dans beaucoup de code

- A quoi servent les exceptions ?
- Comment doit on les utiliser ?
- Comment propager correctement des exceptions ?

Exception

Mécanisme qui permet de reporter des erreurs vers la méthode appelante (et la méthode appelante de la méthode appelante, etc.)

Problème en C

- Prévoir une plage de valeurs parmi les valeurs de retour possible pour signaler les erreurs
- Les erreurs doivent être gérer et propager manuellement

Mantra des Exceptions en Java

Throw early, Catch Late

-- Josh Blosch, Effective Java

- Une exception doit être levée le plus tot possible (le plus proche du problème)
- Une exception doit être attraper le plus tard possible
  - Peu de catch (en bas de stack)
  - Là où on peut **reprendre** sur l’erreur

On propage les exceptions

Par défaut, on propage les exceptions

```
void bar(Path path) throws IOException {
  try(var input = Files.newBufferedReader(path)) {
    ...
  }  // pas de catch ici, on utilise throws au niveau de la méthode
}

void foo(Path path) throws IOException {
   ...
   bar(path);   // on ne fait rien ici aussi
   ...
}

void main() {
  try {
    foo(Path.of("hello.txt"));
  } catch(IOException e) {
     // ici on gère
  }
}
```

Throws vs catch

Pour chaque méthode que l’on contrôle,
on a le choix entre

- Faire un catch pour **reprendre** sur l’erreur
- Déclarer l’exception en throws pour que l’appelant gère l’exception

- A cause de “Throw early, catch late”
  - On va préférer le “throws” au “catch”

Les exceptions et doc

Toutes les exceptions levée par une méthode doivent être documentée

- ```
  /**
    @throws IOException if an i/o exception occurs
  */
  void foo(Path path) { ...
  ```

- ```
  /// @throws IOException if an i/o exception occurs
  void bar(Path path) { ...
  ```

Checked Exception ?

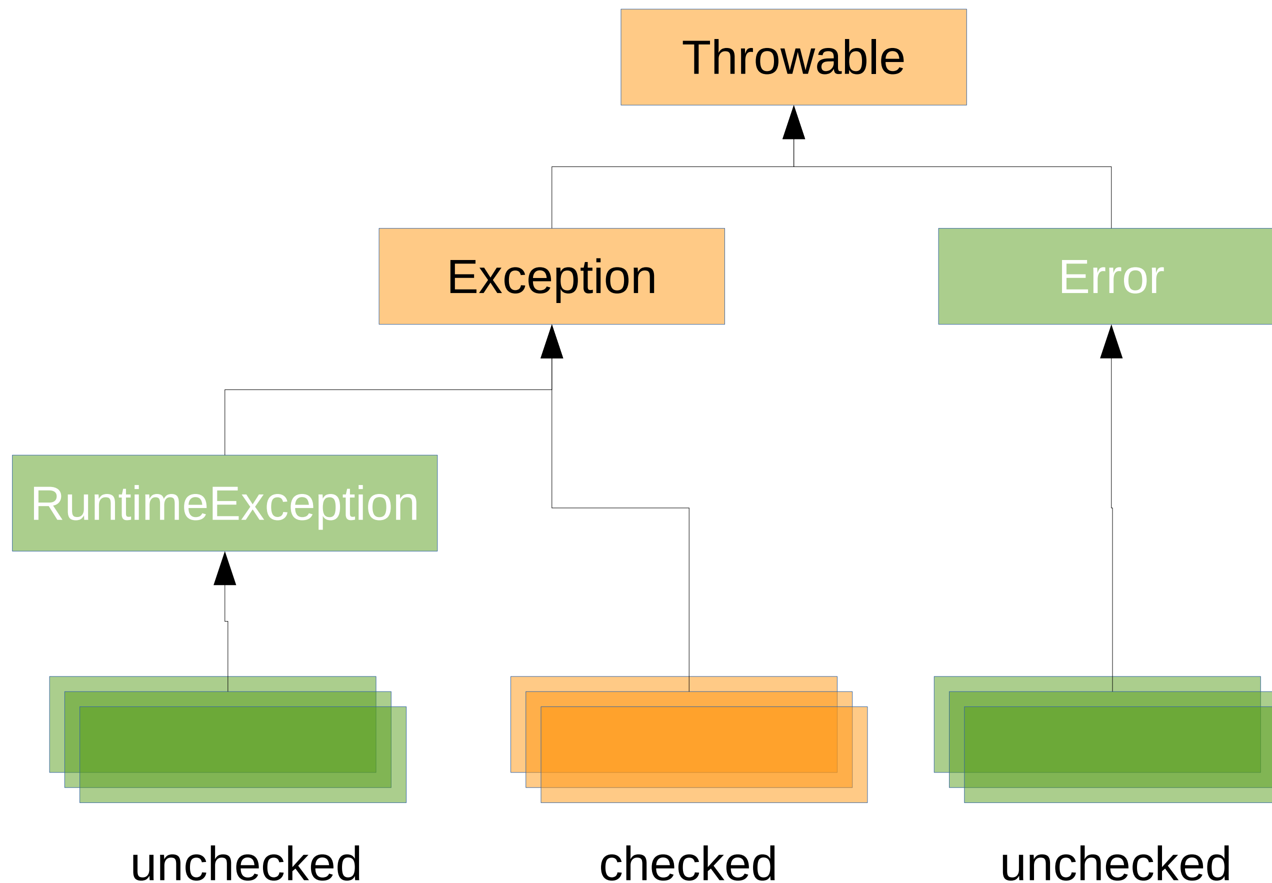
En fait, il y a plusieurs sortes d’exception

- Les exceptions *checked*, le compilateur force à choisir entre throws et catch
- Les exceptions unchecked, le compilateur s’en fout

=> on **ne doit pas** faire un catch/throws d’une exception unchecked

Hierarchie des Throwable

Il faut hériter de Throwable pour pouvoir être lancer par un throw



Usages des Throwable

- Exception (checked)
  - Erreurs du à des conditions externes,
    il faut reprendre sur ces erreurs pour avoir un programme stable
    - IOException, InterruptedException
- Error (unchecked)
  - Erreurs qu’il ne faut jamais attraper
    - OutOfMemoryError, StackOverflowError, AssertionError
- RuntimeException (unchecked)
  - Erreur de prog + erreur métier,qu’il ne faut pas attraper
    - NullPointerException, IndexOutOfBoundsException

Comment attraper des exceptions ?

Attraper des exceptions

La syntaxe try/catch, try()/catch permet d'attraper des exceptions

```
try {
  ...
} catch(InterruptedIOException e) {
  ...
} catch(IOException e) {
  ...
}
```

Le block du try doit être le plus petit possible

La syntaxe est volontairement verbeuse

Ordre des catch

Il faut ordonner les catch du plus spécifique au moins spécifique

- Pour la VM, les catch() sont des instanceof

```
try {
  ...
} catch(IOException e) {
  ...
} catch(InterruptedIOException e) {  // compile pas
  …                                  // block de code est pas
}                                    // appelable
```

Multi-catch

Si on a plusieurs exceptions qui ont la même reprise sur erreur, on peut les grouper (avec ‘|’)

```
try {
  ...
} catch(IOException | SAXException e) {
  … // est appelé si il y a une IOException
    // ou une SAXException
}
```

Precise rethrow

Pour les exceptions *checked*, le compilateur est capable même avec un catch plus générale de repropager les bonnes exceptions

```
void foo() throws NoSuchMethodException,
                  NoSuchFieldException {
  try {
    Foo.class.getMethod("bar", int.class);
    Foo.class.getField("baz");
  } catch(ReflectiveOperationException e) {
    throw e;
  }
}
```

catch(Exception)

Ahhhhhhhhhhhhhh,
dans ce cas on attrape, les erreurs de prog., les erreurs métiers et les erreurs d’I/O.

- Aucune chance d’écrire un code correcte dans le catch !

On écrit pas un code qui fait un catch(RuntimeException), catch(Error), catch(Exception) ou catch(Throwable)

- Si l’on veut gérer plusieurs exceptions différentes, on utilise soit le muti-catch, soit on crée une sous-classe !

Comment créer une exception ?

Créer son exception

On créé son exception lorsque l’on veut reprendre sur l’erreur d’une façon spécifique

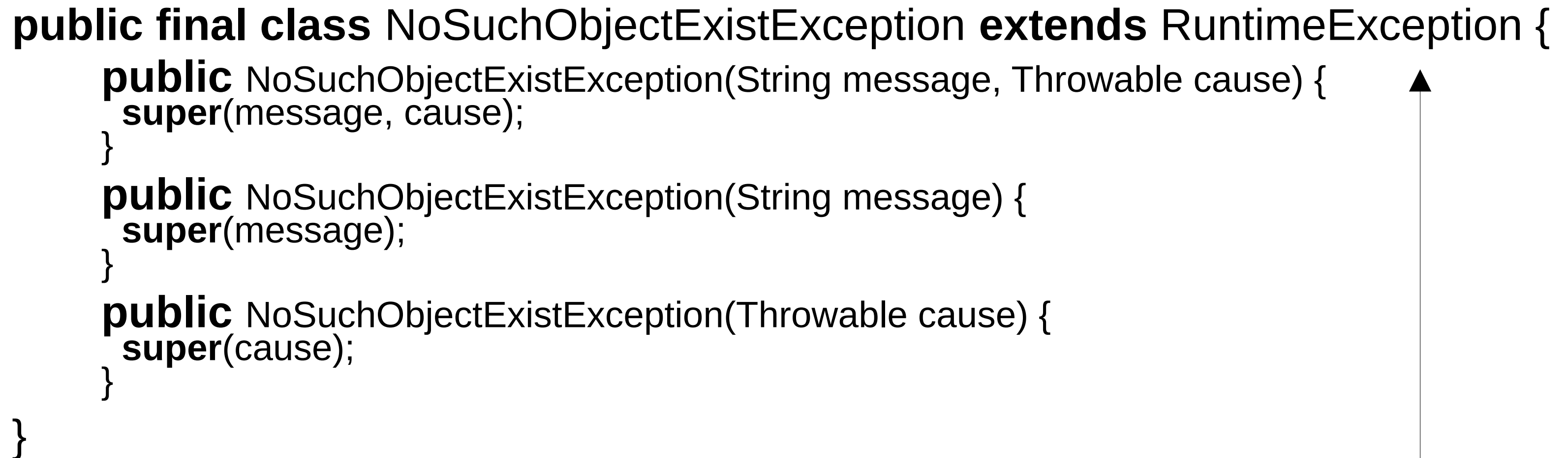
- Par ex: j’accède à ma base de donnée (à la main) et je veux lever une exception si findById marche pas
  - Le driver de la BDD lève déjà une SQLException
  - Mais je veux faire un traitement spécifique (plus tard)

Déclarer son exception

Une exception est créée avec 2 informations

- Un message
- Une cause (optionnel), exception ayant provoquée le problème

On doit déclarer 3 constructeurs



```
public final class NoSuchObjectExistException extends RuntimeException {
    public NoSuchObjectExistException(String message, Throwable cause) {
      super(message, cause);
    }
    public NoSuchObjectExistException(String message) {
      super(message);
    }
    public NoSuchObjectExistException(Throwable cause) {
      super(cause);
    }
}
```

On utilise RuntimeException ici, comme cela,
le compilateur nous embète pas

initCause()

Avant Java 1.4, il n’y avait pas de cause, donc les anciennes exceptions n’ont pas de constructeur qui prend une cause :(

On peut utiliser initCause() pour initialiser la cause même si il n’y a pas de constructeur pour

– **var** error = new NoSuchMethodError();
  error.initCause(cause);
  throw error;

  Ou en une expression (initCause renvoie “this”)

– **throw** (NoSuchMethodError)
    new NoSuchMethodError().initCause(cause);

Le problème des exceptions *checked*

Exception checked vs override

Il n’est pas toujours possible de faire un throws

Si la méthode redéfinie une méthode qui ne déclare pas levée d’exception (ou pas la bonne), on est coincé

```
class MyRunnable implements Runnable {
  @Override
  public void run() {  // la signature est figée :(
    ...
    Files.newInputStream(path); // lève IOException
  }
}
```

Astuce, on va propager une autre exception

On utilise un catch, mais c’est pour repropager

```
class MyRunnable implements Runnable {
  @Override
  public void run() {  // la signature est figée :(
    ...
    try {
      ...
      Files.newInputStream(path);
      ...
    } catch(IOException e) {
      throw new IOError(e);
    }
    ...
  }
}
```

IOError est pas checked,
donc pas de problème ?

Bah si, on n’empèche de pouvoir faire
un catch(IOException) plus tard

Tunneling

On propage une exception runtime que l'on va attraper de l'autre coté

```
class MyRunnable implements Runnable {
  @Override
  public void run() {  // la signature est figée :(
    try {
       ...
    } catch(IOException e) {  // on attrape l'IOException
       throw new UncheckedIOException(e);
    }
  }
}
```

```
var runnable = new MyRunnable();
 try {
  runnable.run();
} catch(UncheckedIOException e) {
  throw e.getCause();   // on re-propage l'IOException
}
```

Cela demande de contrôler tous les sites d'appels

UncheckedIOException vs IOError

Certaines méthodes du JDK comme Files.lines(), Files.list() ou Files.walk() lève une IOError au lieu de UncheckedIOException

IOError est une erreur et pas une Exception, elle n’est donc pas attrapé par un catch(Exception)

Conclusion

Les exceptions ne sont pas des pokemon !

Si on ne peut pas reprendre sur l’erreur à cette endroit, on écrit un throws

Si on peut reprendre sur l’erreur

- On essaye de reprendre sur l’erreur le plus bas possible dans la pile d’exécution pour éviter de dupliquer le code des catch

Dans le run() d’un Runnable ou le main, on doit écrire un catch