

# Introduction à Java

## Cours 5: Gestion des exceptions

Stéphane Airiau

Université Paris-Dauphine

# Gestion des Exceptions

Gérer l'inattendu !

## Exemple

---

```
1 public static Random generator = new Random();
2
3 public static int randInt (int low, int high) {
4     return low + (int) (generator.nextDouble() * (high-low +1));
5 }
```

Que se passe-t-il si l'utilisateur écrit

```
| randInt (10, 5);
```

On pourrait modifier le code de `randInt` pour prendre en compte ce type d'erreur (mais peut être que l'utilisateur a fait une erreur d'interprétation plus profonde...)

## Exemple (suite)

---

Java nous permet de lever une **exception** adaptée, par exemple :

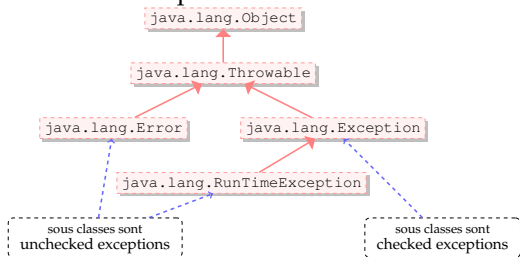
```
if (low > high)
    throws new IllegalArgumentException(
        "low should be <= high but low is " + low + " and high is " + high);
```

On lève une exception d'une classe qui porte un nom parlant `IllegalArgumentException` avec un message qui aidera au débogage.

Quand une exception est levée, l'exécution "normale" est interrompue. Le contrôle de l'exécution passe à un gestionnaire d'erreur.

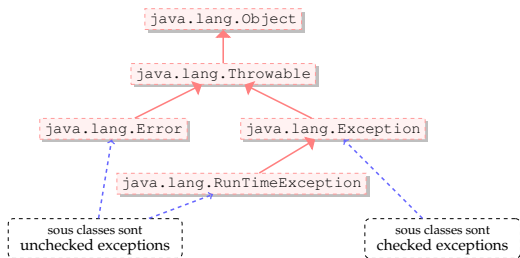
## Gestion des erreurs

Java possède un mécanisme de gestion des erreurs, ce qui permet de renforcer la sécurité du code. On peut avoir différents niveaux de problèmes :



- `Error` représente une erreur grave intervenue dans la machine virtuelle (par exemple `OutOfMemory`)
- La classe `Exception` représente des qui sont reportées au développeur.
- le développeur a la possibilité de **gérer** de telles erreurs et **éviter** que l'application ne se termine
- on veut gérer les erreurs que l'on peut anticiper

# Gestion des erreurs



- checked Exception représente des cas où on peut anticiper le problème (par exemple entrée-sortie).
- RuntimeException ne peuvent pas être vérifiées lors de la compilation  
plutôt des erreurs causées par le programmeur (NullPointerException) que des erreurs causées par une source incontrôlable (ex entrée-sortie).

- Utilisez des exceptions pour des conditions exceptionnelles
- checked exception : conditions pour lesquelles on peut raisonnablement espérer pouvoir récupérer et poursuivre l'exécution
  - ⇒ force le développeur à gérer une condition exceptionnelle (éviter qu'il ne la néglige trop)
  - ⇒ "*exception d'utilisation*" : ces erreurs doivent être déclarées dans la signature de la fonction si elles ne sont pas capturées par la fonction.
- running time exception : indique une erreur de programmation  
`NullPointerException` est "*unchecked*" (sinon, il faudra toujours traiter cette erreur car elle peut intervenir partout!)

## Code reuse : utilisez au plus les exceptions de la bibliothèque standard

---

- `IllegalArgumentException` : argument dont la valeur n'est pas appropriée
- `IllegalStateException` l'objet n'est pas "prêt" pour faire cet appel
- ... on pourrait utiliser seulement ces deux exceptions car la plupart des erreurs correspondent à un argument ou un état illégal
  - `NullPointerException`
  - `IndexOutOfBoundsException`
  - `ArithmeticException`
  - `NumberFormatException`
- pas une science exacte :  
ex : on possède une méthode pour tirer au sort  $k$  dominos dans la pioche,  $k$  étant un argument de ma méthode. Supposons qu'un utilisateur demande de tirer  $k$  dominos alors qu'il y a  $k-1$  dominos dans la pioche. On peut utiliser
  - `IllegalArgumentException` :  $k$  est trop grand
  - `IllegalStateException` pas assez de dominos dans la pioche



## Déclarer des checked exceptions

---

On peut utiliser une clause **throws** pour déclarer une méthode dont on peut anticiper l'échec.

```
public void write(Object obj, String filename)
    throws IOException, ReflectiveOperationException
```

Dans la déclaration, on peut grouper les erreurs dans une super classe, ou bien les lister. Dans l'exemple, on lève des exceptions de type `IO`, mais on pourrait nommer plus précisément toutes les types possibles (sous classes de `IOException`).

## Levée d'exception

---

Lors de la détection d'une erreur

- un objet qui hérite de la classe `Exception` est créé
- ➡ ce qui s'appelle **lever une exception**
- l'exception est propagé à travers la pile d'exécution jusqu'à ce qu'elle soit traitée.

```
1 | int[] tab = new int[5];  
2 | tab[5]=0;
```

### Exception in thread "main"

```
java.lang.ArrayIndexOutOfBoundsException: 5  
at Personnage.main(Personnage.java:2)
```

```
1 | int d=10,t1=5,t2=5;  
2 | System.out.println("vitesse:" + d / (t2-t1));
```

### Exception in thread "main"

```
java.lang.ArithmeticException: / by zero  
at Personnage.main(Personnage.java:21)
```

## Le bloc `try ... catch`

---

- bloc `try` : le code qui est susceptible de produire des erreurs
- on récupère l'exception créée avec le `catch`.
- on peut avoir plusieurs blocs `catch` pour capturer des erreurs de types différentes.
- en option, on peut ajouter un bloc `finally` qui sera toujours exécuté (qu'une exception ait été levée ou non)

Lorsqu'une erreur survient dans le bloc `try`,

- la suite des instructions du bloc est abandonnée
- les clauses `catch` sont testés **séquentiellement**
- le premier bloc `catch` correspondant à l'erreur est exécuté.
  - ⇒ l'ordre des blocs `catch` est donc de l'erreur la plus spécifique à la plus générale!
  - ⇒ on a un mécanisme de **filtre**

## Choisir où traiter une exception

---

Lorsqu'on utilise un morceau de code qui peut lever une exception, on peut

- traiter l'exception immédiatement dans un bloc `try ... catch`
- propager l'erreur :

```
public Domino[] tirer(int nb) throws IllegalStateException {  
    ...  
}  
...  
public void distribuer() {  
    ...  
    Domino[] jeu = tirer(7);  
    ...  
}
```

Ici, on a une erreur, car l'exception qui peut être levée dans `tirer` n'est pas traitée dans la méthode `distribue`.

## Choisir où traiter une exception

---

Lorsqu'on utilise un morceau de code qui peut lever une exception, on peut

- traiter l'exception immédiatement dans un bloc `try ... catch`
- propager l'erreur :

```
public Domino[] tirer(int nb) throws IllegalStateException {  
    ...  
}  
...  
public void distribuer() throws IllegalStateException {  
    ...  
    Domino[] jeu = tirer(7);  
    ...  
}
```

Dans l'exemple, on ne traite pas l'exception dans la méthode `distribuer`, on la propage à la méthode qui appelle la méthode `distribue`.

➡ choisir judicieusement l'endroit où l'erreur est traitée.

## Déclarer et bien Documenter les exceptions

---

- une méthode peut lever plusieurs exceptions
  - ➡ on peut être tenté de les regrouper dans une classe exception mère (mais on perd un peu d'information)
  - ➡ on peut déclarer chaque exception individuellement (N.B. le langage ne requiert pas la déclaration que chaque exception que le code peut lever)
- ➡ permet au développeur d'être plus attentif sur certains aspect
- **@throws** balise javadoc permet de documenter chaque exception
- Depuis Java7, on peut gérer plusieurs exceptions dans une même clause catch

```
catch (ExceptionType1 | ExceptionType2 | ExceptionType 3 e) {  
    ...  
}
```

## N'ignorez pas les exceptions

---

- si une méthode peut lever une exception
  - ↳ le développeur de la méthode n'a pas fait ce choix sans raison
  - ↳ ne l'ignorez pas !
- il est facile d'ignorer une exception sans rien faire.
- au minimum, écrivez un message indiquant pourquoi vous vous êtes permis d'ignorer l'exception.

## Pour les unchecked exceptions

---

- on peut utiliser la sortie erreur (différente de la sortie standard) pour afficher des message
- La méthode `printStackTrace()` de la classe `Throwable` donne le parcours complet de l'exception du moment où elle a été levée jusqu'à celui où elle a été capturée.



## Exemple

---

```
1 | int d=10,t1=5,t2=5;
2 | try{
3 |     System.out.println("vitesse:" + d / (t2-t1));
4 | }
5 | catch(ArithmeticException e) {
6 |     System.out.println(" vitesse non valide ");
7 | }
8 | catch(Exception e) {
9 |     e.printStackTrace();
10| }
```

## Créer sa propre exception

---

- La classe `MyException` hérite de la classe `Exception`.
- Une méthode qui risque lever une exception de type `MyException` l'indique à l'aide de **throws**

```
1 public class PotionMagiqueException extends Exception {
2     public PotionMagiqueException() {
3         super();
4     }
5     public PotionMagiqueException(String s) {
6         super(s);
7     }
8 }
9
10 public class GourdePotionMagique {
11     private int quantite, gorgee=2, contenance=20;
12     public GourdePotionMagique() { quantite=0; }
13
14     public boolean bois() throws PotionMagiqueException {
15         if (quantite-gorgee < 0)
16             throw new PotionMagiqueException
17                 (" pas assez de potion magique!");
18     }
19 }
```

## Exceptions et entrée/sortie

```
1  try {
2      FileInputStream fis = new FileInputStream(new File("test.txt"));
3      byte[] buf = new byte[8];
4      int nbRead = fis.read(buf);
5      System.out.println("nb bytes read: " + nbRead);
6      for (int i=0;i<8;i++)
7          System.out.println(Byte.toString(buf[i]));
8      fis.close();
9
10     BufferedReader reader =
11         new BufferedReader(new FileReader(new File("test.txt")));
12     String line = reader.readLine();
13     while (line!= null) {
14         System.out.println(line);
15         line = reader.readLine();
16     }
17     reader.close();
18 } catch (FileNotFoundException e) {
19     e.printStackTrace();
20 }
21 catch (IOException e) {
22     e.printStackTrace();
23 }
```