

Support de cours

Cours:

## Introduction à la programmation orientée objet (en Java)

Vidéo:

### W16-02-exceptionsyntaxe-JAVA-pt4

Concepts (extraits des sous-titres générés automatiquement) :

**Bloc try. Exception de type inputmismatchexception. Lancement d'une exception de ce type. Cas particulier. Bloc catch. Lecture d'un entier. Nombre maximum d'essais. Première fois. Fin du bloc catch. Exception correspondante. Troisième exemple. Mot clé finally. Lancement d'exception. Blocs catch. Niveau du catch.**



[vers la recherche de séquences vidéo](#)

(dans Introduction à la programmation orientée objet (en Java).)



[vers la vidéo](#)

Center for Digital Education. Plus de matériel de soutien pédagogique ici :

<https://www.epfl.ch/education/educational-initiatives/cede/educational-technologies-gallery/boocs-en/>

# Gestion des exceptions : syntaxe

## (Partie 4)

Introduction à la programmation orientée objet (en Java)

Jamila Sam, Jean-Cédric Chappelier et Vincent Lepetit

...

notes

résumé

0m 0s



```
int lireEntier(int maxEssais) throws Exception
{
    int nbEssais = 1;
    do {
        System.out.println("Donnez un entier : ");
        try {
            int i = clavier.nextInt();
            return i;
        }
        catch (InputMismatchException e) { java.util
            System.out.println("Il faut un nombre entier. Recommencez !");
            clavier.nextLine();
            ++nbEssais;
        }
    } while(nbEssais <= maxEssais);

    throw new Exception ("Saisie échouée");
}
```

Passons maintenant à un troisième exemple où nous allons mettre throw, try et catch dans une même méthode. C'est un cas particulier mais qui peut s'avérer parfois utile. Nous avons donc ici une méthode lireEntier dont le but va être justement de lire un entier qui retourne cet entier lu et qui prend comme paramètre un nombre maximum d'essais, et dont l'en-tête doit être prolongé par cette syntaxe, throws Exception qui vous sera expliquée plus tard. Nous avons donc ici tout le corps de cette méthode dans laquelle on voit que nous avons un bloc try, try sur la lecture d'un entier au clavier. On va vouloir lire un entier. Ce que l'on va faire, c'est qu'on va intercepter les throw possiblement lancés par cette lecture d'un entier au cas où la lecture échoue. C'est typiquement le cas si l'utilisateur, au lieu d'entrer un entier, entre par exemple une chaîne de caractères. Dans ce cas, cette lecture lance une exception de type InputMismatchException, laquelle est définie dans la bibliothèque java.util.

notes

résumé

0m 1s



```

int lireEntier(int maxEssais) throws Exception
{
    int nbEssais = 1;
    do {
        System.out.println("Donnez un entier : ");
        try {
            int i = clavier.nextInt();
            return i;
        }
        catch (InputMismatchException e) { java.util
            System.out.println("Il faut un nombre entier. Recommencez !");
            clavier.nextLine();
            ++nbEssais;
        }
    } while(nbEssais <= maxEssais);

    throw new Exception ("Saisie échouée");
}

```

Si la lecture échoue, on va avoir lancement d'une exception de ce type là, donc branchement au niveau du catch, puisqu'on avait mis ça dans un bloc try, qui va nous permettre par exemple d'afficher un message. "Il faut rentrer un nombre entier." Puis avec cette instruction, on supprime dans le flot d'entrée les caractères laissés par la mauvaise saisie précédente. Enfin, nous augmentons le nombre d'essais. Le programma va donc se dérouler de la façon suivante. Au départ, on aura un nbEssais qui vaut 1. On aura une boucle tant que le nombre d'essai est inférieur ou égal à maxEssais" que l'on a reçu comme paramètre. On demande un entier. On essaye de lire un entier et chaque fois que cette lecture échoue on se branchera sur le bloc catch qui attrape l'exception correspondante. On arrive donc ici à la fin du bloc catch si on a eu un échec de lecture. On continue l'exécution. On continue tant que nbEssais n'a pas atteint le nombre maximum d'essais. Dans le cas où la lecture réussit, on n'a pas de lancement d'exception. On a donc continuation de l'exécution normale. On retourne la valeur lue et on quitte le programme à ce niveau-là.

## notes

## résumé

1m 13s



**Notes :**

- ▶ Java 7 a introduit le multi-catch : `catch(Exception1 | Exception2 | ..)`
- ▶ s'il y a plusieurs blocs catches toujours les spécifier du plus spécifique au plus général  
(sinon, erreur signalée par le compilateur)

Enfin si l'on fait tellement d'erreurs que l'on dépasse le nombre maximum d'essais, il va se passer le déroulement suivant : on va avoir encore une erreur qui va lancer l'exception qui va être interceptée par ce catch ici. On va incrémenter le nombre d'essais. A la fin du catch, on continue l'exécution normale. Mais cette fois-ci le `nbEssais` va être strictement plus grand que le nombre `maxEssais`. Donc on sortira de la boucle `while` et on continuera l'exécution en lançant cette fois-ci une nouvelle exception, laquelle exception sera passée plus haut à d'autres méthodes qui auront appelé `lireEntier` et qui pourront possiblement attraper cette exception et la gérer. Terminons enfin par quelques remarques.

**notes****résumé**

2m 25s



On cherche à remplir 4 tâches élémentaires :

1. signaler une erreur
2. marquer les endroits réceptifs aux erreurs
3. leur associer (à chaque endroit réceptif) un moyen de gérer les erreurs qui se présentent
4. éventuellement, « faire le ménage » après un bloc réceptif aux erreurs

On a donc 4 mots du langage Java dédiés à la gestion des exceptions :

`throw` : indique l'erreur (i.e. « lance » l'exception)

`try` : indique un bloc réceptif aux erreurs

`catch` : gère les erreurs associées (i.e. les « attrape » pour les traiter)

`finally` : (optionel) indique ce qu'il faut faire après un bloc réceptif

La première c'est que depuis Java 7 on a ce qu'on appelle le multi-catch. On a le droit d'écrire `catch Exception1 ou Exception2 etc.` Puis enfin quelque chose que je vous ai déjà dit, quand on a plusieurs blocs `catch` il faut absolument toujours les spécifier du plus spécifique, de l'exception qui est la plus basse dans la hiérarchie des exceptions, à la plus générale, c'est à dire les exceptions les plus hautes dans la relation d'héritage des exceptions. Terminons enfin cette vidéo sur la syntaxe Java de la gestion des exceptions par ce point numéro 4,

notes

résumé

3m 5s



Le bloc `finally` est optionnel, il suit les blocs `catch`

Il contient du code destiné à être exécuté qu'une exception ait été lancée ou pas par le bloc `try`

☞ But : **faire le ménage** (fermer des fichiers, des connexions etc..)

où l'on ferait éventuellement le ménage après un bloc `catch` avec le mot clé `finally` qui indique de façon optionnelle ce qu'il faudrait faire après un bloc réceptif aux exceptions.

notes

résumé

3m 37s



```
class Inverse {  
    public static void main (String[] args) {  
        try {  
            int b = Integer.parseInt(args[0]);  
            int c = 100/b;  
            System.out.println("Inverse * 100 = " + c);  
        }  
        catch (NumberFormatException e1) {  
            System.out.println("Il faut un nombre entier !");  
        }  
        catch (ArithmeticException e2) {  
            System.out.println ("Parti vers l'infini !");  
        }  
        finally {  
            System.out.println("Passage obligé !");  
        }  
    }  
}
```

Ce mot clé finally contrôle donc un bloc qui est optionnel et qui suit les blocs catch associés à un try et qui contient du code qui est destiné à être exécuté que l'on ait lancé l'exception ou pas. Le but de ce bloc est de faire le ménage, par exemple fermer des fichiers, fermer des connexions etc. en cas de lancement d'exception, garantir que ce ménage est bien fait.

notes

résumé

3m 49s





Exemples d'exécution :

```
>java Inverse 4.1
Il faut un nombre entier!
Passage obligé !
```

```
>java Inverse 0
Parti vers l'infini!
Passage obligé !
```

```
>java Inverse 4
Inverse * 100 = 25
Passage obligé !
```

```
>java Inverse
Passage obligé !
```

```
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: 0
```

Prenons un exemple, nous avons ici un petit programme que nous allons pouvoir appeler de différentes façons. C'est la première fois que nous utilisons ici les arguments passés à main.

notes

résumé

4m 13s



```
class Inverse {
    public static void main (String[] args) {
        try {
            int b = Integer.parseInt(args[0]);
            int c = 100/b;
            System.out.println("Inverse * 100 = " + c);
        }
        catch (NumberFormatException e1) {
            System.out.println("Il faut un nombre entier !");
        }
        catch (ArithmeticException e2) {
            System.out.println ("Parti vers l'infini !");
        }
        finally {
            System.out.println("Passage obligé !");
        }
    }
}
```

Ça nous permet de pouvoir par exemple, depuis l'exécution de ce programme Inverse, de passer différents arguments ou éventuellement de ne pas passer des arguments. La façon dont c'est fait, c'est simplement de passer un tableau de string qui sont donc les arguments lors de l'appel de ce programme que l'on va appeler Inverse. On a ici un bloc try qui va essayer de lire les différents arguments Ici, args c'est donc un tableau de chaîne. args[0] c'est donc le premier élément de ce tableau de chaînes. Ce sera donc le premier argument passé au programme lorsqu'on l'exécutera, comme je vous l'ai indiqué précédemment. On essaye de lire un entier dans ce premier argument. Ensuite, on fait une division par b. Puis on affiche que 100 fois l'inverse c'est justement ce nombre c que l'on vient de calculer. Si le premier argument que l'on reçoit n'est pas un entier, cette expression va lancer une exception de type NumberFormatException et donc fera que l'on va se brancher sur le bloc catch sensible à ce type d'exception. On indiquera par exemple "Il faut un nombre entier !". Si par contre on réussit à lire un nombre entier dans le premier argument alors on passe ici, on fait ce calcul, où là, on a un risque de division par 0. Si b vaut 0, cette division va effectivement lancer une ArithmeticException, on va donc se brancher sur ce bloc sensible aux exceptions de type ArithmeticException Si on a une division par 0, on affichera que l'on part vers l'infini. Enfin, quoi qu'il advienne, que l'on ait une exception ici, que l'on ait une exception ici, ou que l'on ait rien du tout dans ce bloc sensible à la gestion des exceptions, on exécutera ce passage obligé. On exécutera ce bloc finally lié à ce bloc try. Voyons

## notes

## résumé

4m 25s



```
class Inverse {  
    public static void main (String[] args) {  
        try {  
            int b = Integer.parseInt(args[0]);  
            int c = 100/b;  
            System.out.println("Inverse * 100 = " + c);  
        }  
        catch (NumberFormatException e1) {  
            System.out.println("Il faut un nombre entier !");  
        }  
        catch (ArithmeticException e2) {  
            System.out.println("Parti vers l'infini !");  
        }  
        finally {  
            System.out.println("Passage obligé !");  
        }  
    }  
}
```

tout ceci en détails sur différents types d'appel de ce programme. Si, par exemple, on appelle notre programme avec un argument 4.1 qui est de type double, à ce moment là, ici, on n'a pas réussi à lire un entier. On se branche ici. On exécute "Il faut un nombre entier !". Comme on a terminé le bloc catch, on passe le bloc finally. Ici on écrit "passage obligé". Puis on termine le programme. Si par contre on a passé un entier mais c'est 0, on a réussi à lire un nombre entier donc on continue. Ici on fait la division par 0, ce qui fait que l'on a une ArithmeticException qui est lancée et interceptée par ce bloc catch, donc on affiche que l'on part vers l'infini. Puis on continue, comme toujours, par le bloc finally qui indique que l'on a un passage obligé. Troisième cas : si l'on appelle notre programme en passant un argument qui est un entier non nul. A ce moment là, on a bien réussi à lire un entier.

### notes

### résumé

Exemples d'exécution :

```
>java Inverse 4.1
Il faut un nombre entier!
Passage obligé !
```

```
>java Inverse 0
Parti vers l'infini!
Passage obligé !
```

```
>java Inverse 4
Inverse * 100 = 25
Passage obligé !
```

```
>java Inverse
Passage obligé !
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: 0
```

On continue. On a fait la division sans aucun souci.

notes

résumé

7m 13s



```
class Inverse {
    public static void main (String[] args) {
        try {
            int b = Integer.parseInt(args[0]);
            int c = 100/b;
            System.out.println("Inverse * 100 = " + c);
        }
        catch (NumberFormatException e1) {
            System.out.println("Il faut un nombre entier !");
        }
        catch (ArithmeticException e2) {
            System.out.println("Parti vers l'infini !");
        }
        finally {
            System.out.println("Passage obligé !");
        }
    }
}
```

Donc on peut afficher que 100 fois son inverse c'est 25. Dans ce cas là, on termine aussi en passant quand même par le finally

notes

résumé

7m 20s



Exemples d'exécution :

```
>java Inverse 4.1
Il faut un nombre entier!
Passage obligé !
```

```
>java Inverse 0
Parti vers l'infini!
Passage obligé !
```

```
>java Inverse 4
Inverse * 100 = 25
Passage obligé !
```

```
>java Inverse
Passage obligé !
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: 0
```

qui est exécuté de toute façon puisqu'on est dans un bloc try. Ici, à la fin de l'exécution de ce bloc try, on va se lancer dans l'exécution du bloc finally et afficher que l'on a un passage obligé. Pour terminer, un dernier exemple. On ne passe aucun argument donc se passe la chose suivante : Nous allons vouloir lire un entier dans le premier élément du tableau A. Mais il se trouve qu'il n'y a pas eu d'arguments, donc ce tableau est un tableau vide. Donc il n'y a même pas un élément à la position 0. Nous avons un index hors des limites et on va donc avoir lancement d'une exception

notes

résumé

7m 27s



```
class Inverse {
    public static void main (String[] args) {
        try {
            int b = Integer.parseInt(args[0]);
            int c = 100/b;
            System.out.println("Inverse * 100 = " + c);
        }
        catch (NumberFormatException e1) {
            System.out.println("Il faut un nombre entier !");
        }
        catch (ArithmeticException e2) {
            System.out.println("Parti vers l'infini !");
        }
        finally {
            System.out.println("Passage obligé !");
        }
    }
}
```

qui est l'exception `ArrayIndexOutOfBoundsException` Donc on aura lancement de cette exception, mais ce que vous pouvez remarquer c'est qu'effectivement on sera avant cela passé par le `finally`. En effet, ici nous sommes de nouveau toujours dans le bloc `try` et comme on va sortir, ici, en lançant une exception,

notes

résumé

8m 6s



## Bloc finally : exemple (2)

Exemples d'exécution :

```
>java Inverse 4.1
Il faut un nombre entier!
Passage obligé !
```

```
>java Inverse 0
Parti vers l'infini!
Passage obligé !
```

```
>java Inverse 4
Inverse * 100 = 25
Passage obligé !
```

```
>java Inverse
Passage obligé !
```

```
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: 0
```

cette exception n'est pas attrapée par aucun des catch. Néanmoins, on passe par le bloc finally. On passe toujours par le bloc finally associé à un bloc try. Donc on affiche ceci, avant de continuer le lancement de cette exception

notes

résumé

8m 25s





```
class Inverse {
    public static void main (String[] args) {
        try {
            int b = Integer.parseInt(args[0]);
            int c = 100/b;
            System.out.println("Inverse * 100 = " + c);
        }
        catch (NumberFormatException e1) {
            System.out.println("Il faut un nombre entier !");
        }
        catch (ArithmeticException e2) {
            System.out.println("Parti vers l'infini !");
        }
        finally {
            System.out.println("Passage obligé !");
        }
    }
}
```

qui n'est pas attrapée et donc qui termine l'exécution du programme avec un message de type "l'exception n'a pas été attrapée", avec ensuite la pile des appels à l'endroit où cette exception a été lancée. Vous voyez donc que quelque soit le schéma d'exécution du bloc try correspondant nous passons par le finally.

notes

résumé

8m 40s





En l'occurrence, ici, effectivement un tel finally n'est pas très utile. Mais si jamais nous avons eu des allocations de ressources, ouvertures de fichier dans ce bloc try nous garantissons en fermant ces ressources, en libérant ces ressources dans le bloc finally, nous garantissons que quelque soit le chemin d'exécution de ce bloc try le bloc finally libérera donc bien les ressources. C'est à ça que sert typiquement un bloc finally. Ceci conclut cette séquence sur la syntaxe des exceptions en Java. sur la syntaxe des exceptions en Java.

notes

résumé

9m 3s

