

# **Exception Handling in Java**

(Java SE 11 Developer Certification 1Z0-819)

## Table des matières

Exception Handling in Java.....	1
I.  Introducing Exception Handling : .....	2
A.  Introduction : .....	2
B.  Try/Catch Syntax : .....	2
C.  Chaining Catch Block : .....	4
D.  Multi-catch block : .....	5
E.  Finally Syntax : .....	6
F.  Cas pratique d'examen : .....	6
II.  Practicing TryWith ressources : .....	7
A.  Try-with-ressources syntax : .....	7
B.  Try-with Demo : .....	8
C.  Tricky bits : .....	9
D.  AutoCloseable Interface : .....	9
III.  Understanding Exception Types : .....	10
A.  Exception hierarchy : .....	10
B.  Exception Types : .....	10
C.  Runtime Exception : .....	11
D.  Checked Exception : .....	12
E.  Understanding Common Errors : .....	13
IV.  Throwing Exceptions : .....	14
A.  How to Throw : .....	14
B.  Rethrowing Exception : .....	15
C.  Overriding (redéfinir) – Overloading (surcharger) methods : .....	15
D.  Throwing Custom Exception : .....	16

## *I. Introducing Exception Handling :*

### *A. Introduction :*

*Plus que d'aider à gérer les exceptions, le but de ce cours est d'apprendre tout ce qui sera utile concernant les exceptions pour passer la certification.*

Un programme peut se planter pour diverses raisons :

- Il peut essayer d'accéder à une base de données en ligne mais le réseau ne fonctionne pas.
- Il peut essayer de lire un fichier sur le disque dur mais celui-ci n'existe pas.
- Il peut essayer d'accéder à des données qui sont protégées par un mot de passe.
- Etc...

De manière générale il y'a deux types d'erreurs :

1. Celles qu'on a provoqué en faisant des erreurs de codage.
2. Celles qui sont indépendantes de notre volonté et sur lesquelles on n'a pas de contrôle.

Les questions d'examen se concentreront sur les exceptions causées par des erreurs de programmation, donc celles qui sont sous notre contrôle.

**Rôle :**

Les exceptions fournissent un moyen de signaler les erreurs mais avec des noms significatifs (car avant c'était signalé avec un code qui n'était pas des plus explicite).

### *B. Try/Catch Syntax :*

Composition du bloc :

1. Un mot clef **TRY** suivi d'accolades **OBLIGATOIRES** (contrairement à un bloc *if/else*).
2. Un mot clef **CATCH** suivi de parenthèse et d'accolades (obligatoires aussi).
3. Un type d'exception et un identifiant (un paramètre) entre les parenthèses du *catch*.

```
try{  
    // partie du code qu'on veut tester car risqué  
}catch (ExceptionType Ex) {  
    //partie qui va gérer l'erreur (HANDLER)  
}
```

Petite précision : cet ensemble peut être désigné en anglais comme *Block* ou *Clause*.


Un exemple classique d'utilisation d'un bloc *try/catch* est avec une tentative de lecture de fichier. On place toute la méthode dans le bloc *try* et si une exception survient le bloc *catch* sera exécuté :

```
BufferedReader br = null;
try {
    br = new BufferedReader(new FileReader("basic_try_demo.txt"));

    // read line by line
    String line;
    while ((line = br.readLine()) != null) {
        System.out.println(line);
    }
}
catch (IOException e) {
    System.err.format("IOException: %s%n", e);
}
```

### Pour l'examen :

Il peut y avoir ce cas qui se présente à l'examen :



```
try{
    throw new IOException();
    openFile("file.txt"); //sera ignoré
} catch (IOException ex) {
    //handle
}
```

Ce code ne compilera pas, car quand une exception est levée dans la partie *try*, tout le restant de cette partie de bloc sera ignoré (c'est-à-dire : *openFile*(« *file.txt* ») et le compilateur sait que cela sera une instruction inaccessible ce qu'il a tendance à ne pas aimer.

Un autre cas possible est celui-ci :

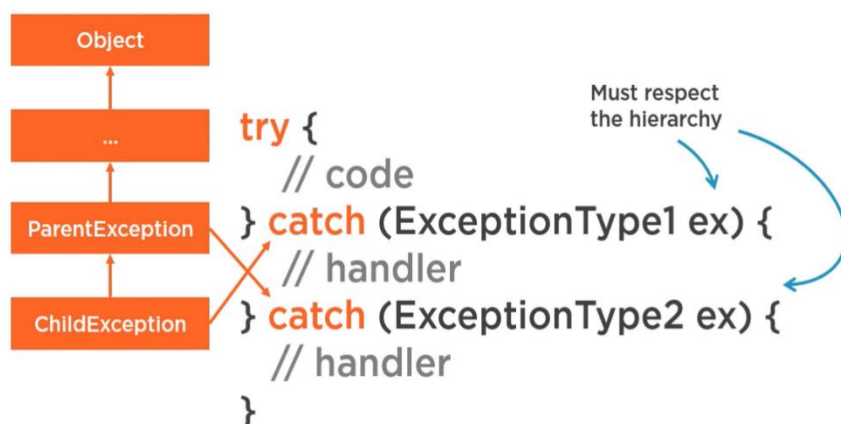
```
try {
    faireQuechose();
}
faitAutreChose(); //absence de catch ou finally = erreur de compilation
}
```

Un bloc *try/catch* se compose de deux parties **OBLIGATOIRES**. Donc si l'une est absente, il y'aura forcément une erreur de compilation.

### C. Chaining Catch Block :

Pour pouvoir gérer plusieurs types d'erreur différentes pour une seule opération, on peut tout à fait multiplier les blocs *catch* en les enchainant les uns après les autres sans limite.

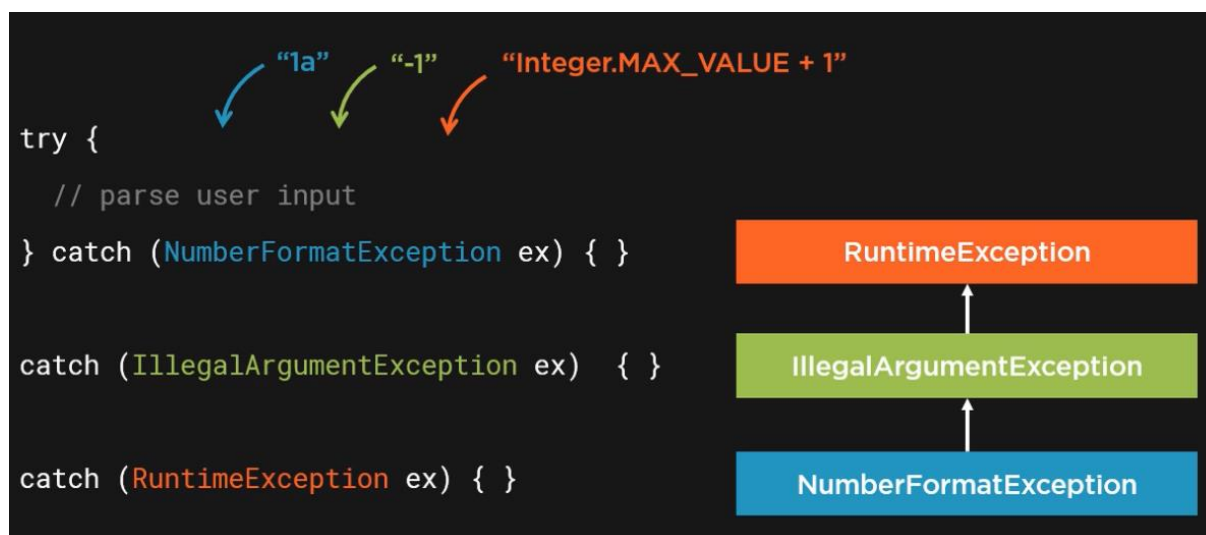
La seule **CONDITION** d'une telle syntaxe est le respect de la hiérarchie des classes d'exception. Comme toutes les classes d'exception sont des héritages et en premier, un héritage de la classe objet, il y'a forcément une hiérarchie parentale.



Il faut donc mettre en **PREMIER** les classes qui sont les plus basses dans le classement et mettre les autres au fur et à mesure de leur position dans la hiérarchie.

En gros, d'abord les enfants, ensuite les parents plus on descend dans les *catch*.

Dans l'exemple suivant, il faut s'imaginer qu'on doit gérer des *inputs* d'un utilisateur et qu'on ne souhaite que des entiers positifs :



On ajoute bel et bien l'exception la plus spécifique à la plus générale quand on descend dans les blocs *catch*. Si on avait inversé l'ordre on aurait eu une erreur de compilation.

Enfin, il faut savoir que les variables qui sont définies dans les blocs *catch* sont inaccessibles aux autres blocs car elles sont locales.

### D. Multi-catch block :

Dans cet exemple on observe que pour traiter un input qui serait mis dans un tableau, pour pouvoir gérer toutes les situations, il nous faut beaucoup de code et de lignes alors qu'on pourrait tout à fait regrouper certains blocs.

```
int num = 0;
try {
    num = Integer.parseInt(args[0]);
}
catch (ArrayIndexOutOfBoundsException ex) { /* identical handler */ }
catch (NumberFormatException ex)          { /* identical handler */ }
catch (IllegalArgumentException ex)        { }
catch (RuntimeException ex)              { }
System.out.println(num);
```

*{no input}* (pointing to args[0])

C'est à ça que vont nous servir les *multi-catch block*. Il nous permet d'attraper plusieurs exceptions différentes avec une seule ligne :

```
try {
    // risky code
} catch (Exception1 | Exception 2 e) {
    // handler
}
```

*Catch either of these* (pointing to Exception1 and Exception 2)

*Required between exception types* (pointing to the vertical bar |)

*Single identifier for all exceptions* (pointing to e, with a warning icon)

Il suffit juste de séparer les exceptions avec une barre verticale et de déclarer un seul identifiant à la fin.

Quand on procède à ce type de regroupement il faut encore respecter la règle parentale de la hiérarchie des classes. On ne peut regrouper des exceptions que si elles sont au même niveau d'héritage (dans l'exemple, les deux exceptions qui sont en bleus) sinon on a une erreur de compilation.

### E. Finally Syntax :

La principale raison de l'existence de ce block est de garantir que l'on puisse fermer certaines ressources auxquelles on a accès.

```

try {
    // open file / DB
} catch (ExceptionType ex) {
    // handler
} finally {
    // close file / DB
}

```

Optional if "finally" is present

Always (!) executes

Dans le cas par exemple où l'on veut accéder à une base de données ou un fichier, si jamais il y'a une exception qui se produit, le block *finally* **GARANTI** l'exécution du code qu'il contient. Il s'exécute quoi qu'il se passe.

On peut dès lors se servir de cette propriété pour fermer l'accès qu'on a ouvert et ce même si une exception est lancée.

L'autre particularité est qu'il rend du coup l'implémentation d'un bloc *catch*, **FACULTATIVE**.

#### Pour l'examen :

Une petite exception à l'exécution obligatoire du block *finally* existe, c'est lorsqu'on utilise la commande *System.exit(0)* dans le bloc *try*, et qui dit au programme de s'arrêter :

```

try {
    System.exit(0); // veut dire d'arrêter l'exécution du programme
}
finally{
    System.out.println("Ligne non exécutée");
}

```

### F. Cas pratique d'examen :

```

StringBuilder sb = new StringBuilder();
String str = null;
try {
    sb.append("a");
    str.toUpperCase();
    sb.append("b");
}
catch (IllegalArgumentException e) {
    sb.append("c");
}
catch (Exception e) {
    sb.append("d");
}
finally {
    sb.append("e");
}
System.out.println(sb);

```

Quel sera le mot affiché par la dernière ligne ?

- Dans le premier block on ajoute la lettre A.
- La deuxième ligne lance une exception donc la troisième ligne n'est pas exécutée.
- Le deuxième bloc, n'est pas exécutée car ce n'est pas une *illegalArgument* exception mais un *NullPointerException* exception. Donc on rentre dans le 3<sup>ème</sup> bloc car *Exception* regroupe tous les types d'exceptions. On ajoute alors un D.
- Enfin, le bloc *finally* est exécuté ce qui nous ajoute un E.

La réponse est : ade.

## II. Practicing TryWith ressources :

### A. Try-with-ressources syntax :

Prévoir toutes les exceptions dans le cas, par exemple de la lecture d'un fichier, en Java, peut s'avérer relativement complexe dans la mesure ou même dans le bloc *finally* il faut de nouveau déclarer un *try/catch* pour anticiper un potentiel problème lors de la fermeture du fichier.

```
BufferedReader br = null;
try {
    br = new BufferedReader(new FileReader("basic_try_demo.txt"));

    // read line by line
    String line;
    while ((line = br.readLine()) != null) {
        System.out.println(line);
    }
}
catch (FileNotFoundException e) {
    System.err.format("FileNotFoundException: %s%n", e);
}
catch (IOException e) {
    System.err.format("IOException: %s%n", e);
}
finally { // je déclare un nouveau try/catch à l'intérieur pour gérer la
    fermeture
    try {
        if (br != null) {
            br.close();
        }

    } catch (IOException e) {
        e.printStackTrace();
    }
}
```

C'est pour la partie de la fermeture que le TRY-with-Ressources peut nous être utile.

**SYNTAXE :**

On a des parenthèses juste après le *try* dans lesquelles on place les ressources **FERMABLES** (*fileInputStream*, *bufferReader* etc...), séparées entre elles par un point-virgule s'il y'en a plusieurs.



On peut encore déclarer un bloc *catch* et un bloc *finally* mais ce n'est pas obligatoire. Un bloc *finally* est implicitement déclaré dès que l'on fait un Try avec ressources.

***B. Try-with Demo :***

Dans cet exemple on ouvre un fichier *in* et on ouvre un fichier *out*. La troisième ligne copie le *in* dans le *out*.

```
try (FileInputStream in = new FileInputStream("in.txt");
    FileOutputStream out = new FileOutputStream("out.txt")) {
    out.write(in.readAllBytes());
}
```

Si ce n'est pas visible au premier abord, les deux premières lignes font partie de la même paire de parenthèses. Java gèrera donc la fermeture et ses exceptions pour nous. Malheureusement dans la plupart des cas, un bloc *catch* sera quand même obligatoire pour gérer les autres types d'exceptions liées dans ce notre cas à la lecture/écriture de fichier.

**Pour résumer :**

Le *try/with* ressource nous dispense de devoir écrire un *try/catch* dans le bloc *finally* quand on doit fermer une ressource.



### C. Tricky bits :

Quelques petites astuces concernant le *try/with* :

#### A. Déclaration :

On doit déclarer le type de la ressource **DANS** les parenthèses :

**Erreur** (le type est déclaré en dehors des parenthèses à la première ligne) :

```
FileInputStream in;
try(in = new FileInputStream("in.txt")) {
}
```

**Correct :**

```
try(FileInputStream in = new FileInputStream("in.txt")) {
}
```

On peut tout aussi bien utiliser le type `var` car c'est une variable locale.

#### B. Portée :

```
try (FileInputStream in = new FileInputStream("file.txt")) {
    in.read();
}
catch (Exception e) {
    in.read(); // does not compile!
}
finally {
    in.close(); // does not compile!
}
```

Assez simplement la portée de la variable ressource sera limitée au bloc dans lequel elle est déclarée. Ici le bloc `try` et si on essaye de l'utiliser ailleurs il y'a une erreur de compilation.

#### C. Ordre de fermeture :

On aurait tendance à penser que Java ferme les ressource dans l'ordre dans lesquelles elles sont déclarées mais c'est l'inverse qui se produit.

C'est la ressource 2 qui sera fermée en premier.

↓

```
try (Resource1 ; Resource2;) {
    // code
}
```

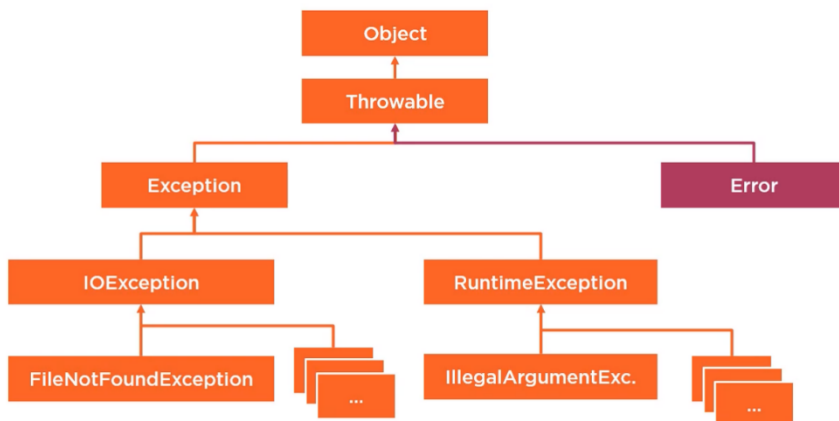
### D. AutoCloseable Interface :

Petite information intéressante, il est **OBLIGATOIRE** d'implémenter cette interface si l'on veut créer une classe qui serait compatible avec le *tryWithResource* (et par conséquent il faudra redéfinir la méthode *close()* ).

### III. Understanding Exception Types :

#### A. Exception hierarchy :

Comme les exceptions sont des classes, leur instantiation nécessite une utilisation obligatoire du mot clef `NEW`.



Si comme toutes les classes elles héritent des objets, les exceptions sont aussi les enfants de la classe *Throwable* qui en plus des exceptions est héritée par la sous classe *Error*.

Un des buts de la classe *Throwable* est d'amener les exceptions et les erreurs qui sont pourtant de nature et de buts très différents, « sous un même toit ».

Plus on descend dans la hiérarchie plus la gestion de l'exception sera spécifique.

#### B. Exception Types :

Il existe deux types de catégorie d'exception :

##### CHECKED :

Ce sont celles qui sont vérifiées par le compilateur. Si une méthode lance une exception de ce type et qu'on ne la gère pas, le code ne compilera tout simplement pas.

Dans cet exemple on a une méthode *doThat()* qui lance une *IOException* qui est fait partie de la catégorie *checked*. Elle doit donc obligatoirement être traitée. Dans notre exemple, ce n'est pas le cas donc le compilateur signale une erreur. Il faudra alors simplement mettre la méthode dans un bloc *Try/catch*.

```

public static void main(String[] args) {
    doThat(); //erreur signalée par le compilateur
}

private static void doThat() throws IOException {
    throw new IOException();
}
  
```

## UNCHECKED :

Elles sont aussi appelées les « *Runtime* » exception et comme on le devine, il n'est pas obligatoire de les gérer.

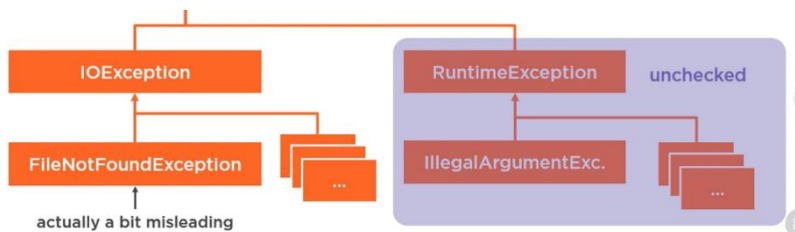
Exemple, je déclare une méthode *dothis()* qui renvoie une exception de type non vérifiée et contrairement à l'exemple précédent je ne suis pas obligé de mettre la méthode dans un bloc *Try/catch* :

```
public static void main(String[] args) {
    doThis(); //pas d'erreur signalée par le compilateur
}

private static void doThis() throws IllegalArgumentException {
    throw new IllegalArgumentException();
}
```

La règle générale veut que, quelle que soit leur type (vérifiée ou non) les exceptions feront planter le programme si elles ne sont pas gérées.

Donc en vérité la différence entre les deux types s'observe surtout au moment de la compilation.



Pour savoir si une exception est vérifiée ou non il suffit de voir si elle hérite de la classe *RuntimeException*. Dans ce cas elles seront alors non-vérifiée (*Unchecked*).

## C. Runtime Exception :

Ici on se familiarise un peu avec certains types d'exception qui sont assez courants.

Ici on aura une exception du type *ArithmeticException* :

```
public static void main(String[] args) {
    int result = 5/0;
}
```

Ici du type *ArrayIndexOutOfBoundsException* :

```
public static void main(String[] args) {
    int [] arr = new int[1];
    System.out.println(arr[1]);
}
```

Ici une *IllegalArgumentException*. Elle n'est en générale pas gérée ou attrapée car on essaye plutôt de régler le problème en amont en corrigeant les données qui seraient fournies.

```
public static void main(String[] args) {
    int age = 15;
    if (age < 21 ) {
```

```
        throw new IllegalArgumentException("message");
    }
}
```

Ici un *NullPointerException* :

```
public static void main(String[] args) {
    String s1 = null;
    s1.toLowerCase();
}
```

### ***D. Checked Exception :***

Cette fois-ci on se concentre surtout sur les deux qui sont les plus demandées à l'examen :

#### **IOException :**

C'est une exception qui sera levée à chaque fois que quelque chose ne va pas avec une opération d'entrée ou de sortie d'un fichier :

```
try (var in = new FileInputStream("file.txt")){
} catch(IOException e){
}
}
```

#### **FileNotFoundException :**

Assez clair, c'est si on essaye d'accéder à un fichier mais celui-ci est introuvable :

```
try {
    var br = new BufferedReader(new FileReader("file.txt"));
} catch (FileNotFoundException e) {
    e.printStackTrace();
}
```

NDR : Il y'a beaucoup d'exemple d'écriture qu'on est susceptible d'avoir à l'examen de la certification, qui permettent de tester nos connaissances sur ce sujet. J'invite donc le lecteur à aller voir la vidéo de Pluralsight sur ce sujet.

## *E. Understanding Common Errors :*

**Principe :** On ne doit pas les attraper ou les manipuler. Ce sont des exceptions **NON-Vérifiées** qui héritent de la classe **ERROR**.

Dans cette partie et pour la certification, seuls deux seront traitées :

### 1) **ExceptionInitializerError :**

```
public class InitErrorExample {
    // cause 1
    static int n = 2 / 0;

    static {
        // cause 2
        int n = 1;
        if (n < 2)
            throw new
IllegalArgumentExcepTion();
    }
    public static void main(String[]
args) {
    }
}
```

Pour comprendre cette erreur, il faut se rappeler ce qu'est un *Class Initializer*. C'est un bloc qui permet d'initialiser une variable qui sera commune à toutes les classes et qui se lancera avant le chargement d'une classe (voir partie POO).

Donc si cet *initializer* lève une exception il la lancera ainsi que la cause (qui dans notre exemple sera la cause 1).

On aura donc comme message :

Exception in thread "main" java.lang.ExceptionInInitializerError  
 Caused by: java.lang.ArithmeticException: / by zero

### 2) **StackOverflowErrorExample :**

C'est une erreur qui est causée par une boucle de référence.

On peut voir ici que la méthode *calculate()* fait référence à elle-même ce qui renvoie à son exécution....

```
public class StackOverflowErrorExample {

    public static void main(String[] args) {

        calculate(1, 2);
    }

    private static void calculate(int a, int b) {
        ↺ calculate(a, b); ↻
    }
}
```

### 3) **NoClassDefFoundError :**

Cette erreur signifie que si la classe était disponible au moment de la compilation, pour une raison ou pour une autre, elle a disparu au moment de l'exécution (éventuellement le fichier jar qui aurait été supprimé du drive).

## IV. Throwing Exceptions :

### A. How to Throw :

On peut commencer par un exemple : je crée une méthode `setAge()` qui sera dans la classe `personne` et j'ajoute une exception dans la signature de méthode :

Version avec une *checked* exception :

```
void setAge2(int age) throws IOException {
    this.age = age;
    throw new IOException();
}
```

Version avec une *unchecked* exception :

```
void setAge1(int age) throws IllegalArgumentException {
    if (age <= 0) {
        throw new IllegalArgumentException();
    }
    this.age = age;
}
```

Donc la question est, si j'utilise ces méthodes telles quelles dans le « *main* », le code compilera-t-il ?

```
Person p = new Person();
p.setAge1(10);
```

Dans ce cas-ci oui car l'exception est *unchecked*.

Dans le cas de la méthode `setAge2()`, le code ne compilera pas car la méthode est *checked*.

```
Person p = new Person();
p.setAge2(5);
```

D'ailleurs IntelliJ dira que la méthode est *unhandled* et qu'il faut donc la gérer. Il faut donc la mettre dans un bloc TRY/CATCH.

Il y'a un point essentiel auquel il faut être attentif dans le traitement de l'exception, c'est l'utilisation du mot clef car il y'en a deux différents :

**THROWS** : Utilisé pour une déclaration dans la signature de méthode.

**THROW** : Utilisé pour lancer une nouvelle exception. Associé avec le mot clef **NEW**.

Donc quand on déclare une exception dans la méthode `setAge()`, on doit aussi la déclarer dans la signature de méthode. C'est **OBLIGATOIRE** pour les exceptions **CHECKED** et non obligatoire pour les autres. D'ailleurs la documentation Java ne le recommande pas pour les exceptions qui hérite de *Runtime (unchecked)*. La raison est simple, il peut y en avoir des très nombreuses, ce qui tendrait à rendre le code plus lourd à lire si on ajoute des exceptions à toutes les signatures de méthode.

## B. Rethrowing Exception :

Comme on le sait une exception **CHECKED** doit être traité dans un bloc *try/catch*.



```

void calculate() throws IOException {
    Data d = fetchData();
    // handle data
}

Data fetchData() throws IOException {
    Connection conn = openAConnection();
    return conn.queryDb("...");
}

```

La question qui se pose c'est, où le placer ? Et là, il n'y a pas vraiment de consensus. Soit on peut la traiter directement dans la méthode où on a déclaré l'exception soit on la traite en **AVAL**.

Je peux donc passer la « balle » de l'exception en utilisant la déclaration dans la signature de méthode. Je passerai ainsi l'exception de méthode en méthode jusqu'à ce que je décide d'enfin la gérer.

## C. Overriding (redéfinir) – Overloading (surcharger) methods :

Si on a une classe qui hérite d'une autre (ici enfant hérite de parent) et qu'elle redéfinit une méthode qui a une exception dans sa signature il y'a une règle à respecter :

Quand on classe redéfinit une méthode d'une classe supérieure ou d'une interface, il n'est **PAS AUTORISEE** d'ajouter une nouvelle **CHECKED** exception d'un niveau **SUPERIEUR** à la signature.

Donc cet exemple est incorrect car *Exception* est plus haut dans la hiérarchie que *IOException*.

D'ailleurs si la classe parent ne jette aucune *checked exception*, on ne peut pas en définir une dans la signature de la méthode enfant.

En gros il faut respecter la hiérarchie des classes et la présence ou l'absence d'exception dans la signature de la méthode parente.

```

class Parent {
    void doThing() throws IOException {
    }
}

class Child extends Parent {
    @Override
    void doThing() throws Exception {
    }
}

```

```
signature
void doThing() throws IOException { }

void doThing() throws RuntimeException { }
not part of the signature
```

Dans le cas d'une surcharge de méthode, il faut savoir que le `throws` ne fait pas partie de la signature de méthode donc impossible de changer la déclaration d'exception.

Donc ces deux méthodes-ci ne peuvent exister dans la même classe.

### ***D. Throwing Custom Exception :***

Si l'on veut un type d'exception plus adapté à nos besoins, Java permet de créer son propre type.

En général on va toujours donner le nom de la classe suivi du mot `Exception`. Pour qu'elle soit en plus considérée comme effective, cette exception doit en plus hériter d'une classe. Soit *Exception* (si on en veut une de type *Checked*) soit *RuntimeException* (si on en veut une de type *Unchecked*).

Evidemment on peut descendre plus bas dans la hiérarchie des classes si on a besoin d'hériter d'un type spécifique en particulier.

```
public class CustomException extends RuntimeException {
    public CustomException(String msg) {
        super(msg);
    }
}
```