Les exceptions

Tous droits réservés à STUDI - Reproduction interdite

Table des matières

I. Contexte 3

II. Erreur/Exception 3

III. Exercice : Appliquer la notion 7

IV. La gestion des exceptions 8

V. Exercice : Appliquer la notion 12

VI. Le try-catch-finally (capture) 12

VII. Exercice : Appliquer la notion 16

VIII. Le throws/throw (spécification) 16

IX. Exercice : Appliquer la notion 18

X. Le try-with-resources 19

XI. Exercice : Appliquer la notion 21

XII. Les exceptions personnalisées 22

XIII. Exercice : Appliquer la notion 24

XIV. Essentiel 24

XV. Auto-évaluation 24

A. Exercice final 24

B. Exercice : Défi 28

Solutions des exercices 32

Tous droits réservés à STUDI - Reproduction interdite

I. Contexte

Durée : 1 h 45

Environnement de travail : E. D. I. Intellij Idea

Prérequis : les commons objets, les classes...

Contexte

Lorsque nous lançons un programme, on peut rencontrer des problèmes sous la forme de différents types d'erreurs et faire face à un comportement imprévisible ou un fonctionnement exceptionnel de notre programme.

Lorsqu'une erreur se produit, Java s'arrête normalement et génère un message d'erreur. Le terme technique pour cela est : lever une **exception** (*throw an exception*).

Lorsqu'une exception se produit, le déroulement normal du programme est interrompu et le programme/l'application se termine anormalement, ce qui n'est pas recommandé, par conséquent, ces exceptions doivent être traitées.

II. Erreur/Exception

Objectifs

- Comprendre ce qu'est une erreur/exception en Java
- Différencier les erreurs de logique métier, de compilation et d'exécution
- Les approches de résolution des erreurs de logique métier et de compilation

Mise en situation

Lorsque nous lançons un programme, on peut rencontrer des problèmes, sous la forme de différents types d'erreurs et faire face à un comportement imprévisible ou un fonctionnement exceptionnel de notre programme.

Il existe deux types de familles d'erreurs qui se produisent souvent :

- Les erreurs de code logiciel, ce qui inclut les **erreurs de compilation** et les **erreurs d'exécution**
- Les erreurs de **logique métier** .

Erreur de code logiciel

Ce sont des erreurs liées au code et au fonctionnement de la JVM. Lorsqu'elles se produisent, la JVM arrête le programme et produit un message d'erreur. Java va donc **lever une exception** (relever une erreur) car nous sommes confrontés à un comportement exceptionnel de l'application (le plantage ou crash de l'application).

On distingue deux types d'erreurs:

1. **Les erreurs de compilation** : elles surviennent lors de la phase de compilation du code, c'est-à-dire la phase où la JVM va traduire (compiler) le fichier texte du code source .java en fichier binaire .class (bytecode/code machine ou fichier exécutable par la machine) grâce au compilateur **javac**.
2. **Les erreurs d'exécution** : elles surviennent lors de la phase d'exécution du fichier binaire .class par la JVM (Java Virtual Machine).

1. Les erreurs de compilation/Throwable, Error

Ces erreurs portent généralement sur les violations des règles **syntaxiques** et **sémantiques** de la grammaire du langage Java.

Exemple **Règles Syntaxiques**

Une violation des règles grammaticales du langage peut être par exemple : le non-respect des règles de nomination de variables...

```
1 public class Main {
2   public static void main(String[] args) {  System.out.println "hello World  }
3 }
```

```
1 console@user:~$
2 java: '(' expected
3 java: ')' expected
4 java: unclosed string literal
5 java: ';' expected
```

Ce code viole quelques règles syntaxiques qui sont exposées dans la console telles que :

- les parenthèses ouvrantes et fermantes manquantes à l'appel de la méthode **println( )** pour l'affichage des données à la console
- la non-fermeture de la chaîne de caractère par un "
- et enfin le point-virgule manquant ; à la fin de l'instruction.

Exemple **Règles sémantiques**

Une violation des règles sémantiques peut être par exemple : l'utilisation de mots interdits, de code non interprétable, de types incorrects...

```
1 public class Main {
2   public static void main(String[] args) {
3     ...
4     String 123 = "123";
5
6     boolean zero = 0;
7
8     int entier = 2.5;
9
10    float nombre-a-virgule = 100;
11    System.out.println(~);
12  }
13 }
```

```
1 java: not a statement
2 java: incompatible types: int cannot be converted to boolean
3 java: possible lossy conversion from double to int
4 java: illegal start of expression
```

Ce code viole plusieurs règles sémantiques du langage Java telles que :

- les noms de variables Java ne peuvent pas commencer par des nombres ou n'être simplement que des nombres
- des conversions de types incorrects ou l'insertion de types incompatibles entre eux
- des éléments qui ne sont pas gérés par le langage Java.

Tous droits réservés à STUDI - Reproduction interdite

Fondamental **Correction des erreurs de compilation**

Choisissez de bons éditeurs de code comme Atom, VS Code, Sublime Text... ou encore mieux, de bons IDE comme Eclipse, NetBeans ou Intellij Idea Ultimate qui peuvent détecter les erreurs de grammaire (syntaxe/sémantique) en mettant en évidence (coloration, surlignage...) des parties de code contenant des anomalies directement lors de l'écriture de votre code.

Cependant, ils ne sont pas toujours suffisants ou ne répondent pas toujours à toutes nos envies. Par conséquent, Java fournit un mécanisme de gestion des erreurs : la **gestion des exceptions** qu'on verra au chapitre suivant.

2. Les erreurs d'exécution/RuntimeError

Lorsque l'étape de compilation est terminée et que le programme produit un bytecode, on passe à la phase d'exécution de ce bytecode.

Pendant la phase d'exécution, le programme peut se terminer brusquement et donc détecter des erreurs d'exécution impossibles à détecter lors de la phase de compilation.

Exemple **Type d'erreur d'exécution**

Ce genre d'erreur se produit lorsque le programme effectue une opération interdite ou illégale.

```
1 public class DiviserParZero {
2     public static void main(String[] args) {
3         int distance = 10;
4         int temps = 0;
5         System.out.println("La vitesse vaut ==> " + distance / temps);
6     }
7 }
```

```
1 Exception in thread "main" java.lang.ArithmeticException: / by zero
2   at DiviserParZero.main(DiviserParZero.java)
```

La JVM vient de relever une **ArithmeticException** qui expose l'exécution d'une opération illégale ou interdite : la division par zéro.

Nous savons bien que la vitesse est le rapport de la distance par le temps. La compilation de ce programme se passe sans problème.

Cependant, l'opération de division par zéro est illégale, donc cette erreur sera détectée à l'exécution de ce programme.

Fondamental **Correction des erreurs d'exécution**

En testant notre code, et en utilisant des techniques de vérifications formelles (ex **PathFinder**), on arrive à bien s'assurer de ne pas rencontrer d'erreurs d'exécution.

On peut utiliser le système de débogage ou la gestion des exceptions pour traiter ce genre d'erreurs.

Erreur de logique métier

Ce sont des erreurs difficiles à détecter car elles ne provoquent pas d'interruption de programme mais produisent plutôt des résultats logiquement incohérents.

Tous droits réservés à STUDI - Reproduction interdite

Exemple **Type d'erreur de logique métier classique**

```java
public class Main {
  public static void main(String[] args) {
    String[] nba = { "Lebron James", "Stephen Curry", "Kevin Durant"};
    String[] uefa = { "Cristiano Ronaldo", "Lionel Messi", "Kylian Mbappé"};

    /* Souhait : Afficher la liste des joueurs de la NBA */
    for (String nba_players : uefa) {
      System.out.println(nba_players);
    }
  }
}
```

```
console@user:~$
Cristiano Ronaldo
Lionel Messi
Kylian Mbappé
```

En effet, l'erreur illustrée ci-dessus sera difficile à détecter car l'application affiche bien ce qu'on lui demande et ne plante pas.

Cependant, Lionel Messi sur la liste des basketteurs NBA est logiquement incorrect (voire impossible).

Fondamental **Correction des erreurs de logique métier**

Pour détecter et corriger les erreurs de logique métier, on a besoin des programmes appelés **debugger** qui vont suivre, instruction par instruction, l'exécution du programme et relever le contenu de toutes les variables à chaque étape.

Syntaxe **À retenir**

Dans ce chapitre, les points importants à retenir sont :

- Les erreurs de logique métier sont difficiles à détecter car elles passent souvent inaperçues puisque le programme compile et s'exécute sans signaler un quelconque problème ; on a vu comment les détecter et les corriger.
- Les erreurs de code Java qui se produisent :
  - lors de la phase de compilation du programme : on a abordé différentes approches, techniques et mécanismes qu'offrent certains outils pour les détecter et les corriger tels que les bons éditeurs de code et les bons **EDI**, un **débugger** ou la gestion des exceptions.
  - lors de la phase d'exécution du programme : on a abordé différentes approches qu'offrent certains outils pour les détecter et les corriger tels que **PathFinder** et aussi le mécanisme Java pour la gestion des exceptions.

Complément

Les règles de grammaire (spécifications) Java à respecter afin d'éviter les erreurs de compilation : https://docs.oracle.com/javase/specs/jls/se15/html/jls-2.html[1]

Lien vers la documentation de PathFinder : http://javapathfinder.sourceforge.net/

1 https://docs.oracle.com/javase/specs/jls/se7/html/

Tous droits réservés à STUDI - Reproduction interdite

Liens vers la documentation des techniques de débogage :

- Avec l'E.D.I. Intelli Idea Ultimate ==> https://www.jetbrains.com/help/idea/debugging-your-first-java-application.html[1]
- Avec l'E.D.I. Eclipse ==> https://www.vogella.com/tutorials/EclipseDebugging/article.html
- https://jmdoudoux.developpez.com/cours/developpons/eclipse/chap-eclipse-debug.php

III. Exercice : Appliquer la notion

Pour mettre en évidence la notion d'erreur, nous allons faire une illustration très simple.

Question

[solution n°1 p.33]

Identifiez le type d'erreur que l'on a et quelle approche adopter pour le résoudre.

```
1 import java.io.File;
2 import java.io.FileReader;
3
4 public class FileNotFoundStudi {
5
6    public static void main(String args[]) {
7       File file = new File("E://file.txt");
8       FileReader fr = new FileReader(file);
9    }
10 }
```

Indice :

Erreur de logique métier : tout compile bien, cependant, les résultats sont logiquement incohérents.

Erreur de compilation : porte sur le respect des règles (syntaxiques et sémantiques) grammaticales du langage Java.

Erreur d'exécution : survient après compilation et nécessite d'être gérée par des mécanismes propres au langage (gestion des exceptions en Java par exemple).

Indice :

Type d'erreur : les erreurs de code sont soit des erreurs de **compilation**, soit des erreurs d'**exécution** et surtout pas des erreurs de **logique métier**.

Et donc trois choix :

- **Erreur de logique métier**
- **Erreur de compilation**
- **Erreur d'exécution**

Indice :

Approche de résolution : selon les types d'erreurs

Les erreurs de logique métier ==> Débogage : Utiliser un débogueur pour suivre instruction par instruction l'exécution du programme et le contenu de toutes les variables à chaque étape.

Les erreurs de compilation ==> EDI checker/ Débogage/Gestion des exceptions : Un bon éditeur révèle déjà les anomalies, cependant on peut toujours utiliser un débogueur ou encore gérer les exceptions.

Les erreurs d'exécution ==> PathFinder/Gestion des exceptions ou : Utiliser une vérification formelle (PathFinder) ou encore la gestion des exceptions.

1 https://www.jetbrains.com/help/idea/debugging-your-first-java-application.html

Tous droits réservés à STUDI - Reproduction interdite

IV. La gestion des exceptions

Objectifs

- Comprendre le mécanisme des exceptions et son utilisation
- Comprendre les différents types d'erreur en Java
- La gestion des exceptions et la hiérarchie des exceptions en Java

Mise en situation

Lorsque nous sommes confrontés à un comportement exceptionnel d'une application (le plantage ou le crash de l'application), le langage Java nous offre un mécanisme nous permettant de gérer la situation avec élégance afin de donner à un utilisateur une expérience toujours positive de l'application.

Gestion d'exception

Les exceptions représentent le mécanisme de gestion des erreurs intégré du langage Java.

Il est composé d'objets représentant les erreurs et d'un ensemble de trois mots-clés qui permettent de détecter et de traiter ces erreurs **(try, catch, finally)** mais aussi de les lever ou de les propager **(throw, throws)**.

Parmi les raisons fréquentes qui produisent ces types d'erreurs, on retrouve souvent :

- Les données non valides entrées par un utilisateur
- Un fichier introuvable que le programme doit utiliser
- Une connexion réseau perdue ou en panne en plein milieu d'une communication, un système de fichiers corrompu ou encore la JVM en manque de mémoire.

Exigence et types d'exceptions

Les méthodes d'un programme Java robuste et valide qui peuvent lever certaines exceptions doivent respecter les exigences **capture** des exceptions ou de la **spécification** des exceptions.

- Être entourées par un bloc **try/catch** permettant de capturer des exceptions afin de les traiter (**capture**)
- Ou ajouter le mot-clé **throws** dans leur déclaration qui spécifie l'exception à gérer (**spécification**).

Toutes les exceptions ne sont pas soumises à l'exigence de capture ou de spécification. On distingue trois grandes catégories d'exception dont une seule est soumise à l'exigence.

1. Les **exceptions vérifiées**
2. Les **exceptions non vérifiées/exceptions d'exécution**
3. Les **erreurs**

1. Les exceptions vérifiées/les exceptions de compilation

Ce sont des exceptions qu'une application bien rédigée doit anticiper et surmonter. On doit soit les propager, soit les gérer nous-mêmes.

Exemple **Un fichier texte introuvable par un programme**

Par exemple, supposons qu'une application demande à un utilisateur un nom de fichier d'entrée pour pouvoir l'utiliser après.

Normalement, l'utilisateur fournit le nom d'un fichier existant et lisible. Mais parfois, l'utilisateur fournit le nom d'un fichier inexistant (fichierInexistant.txt).

Tous droits réservés à STUDI - Reproduction interdite

```
1 import java.io.FileReader;
2
3 public class FileNotFoundStudi {
4
5    public static void main(String args[]) {
6       File file = new File("C://fichierInexistant.txt");
7       FileReader fr = new FileReader(file);
8    }
9 }
```

```
1 C:\>javac FileNotFoundStudi.java
2 java: error: unreported exception FileNotFoundException; must be caught or declared to be
  thrown
3       FileReader fr = new FileReader(fichierInexistant.txt);
4                       ^
5 1 error
```

Ici, sur la console, Java suggère comment surmonter le problème du plantage brutal du programme en raison de l'omission de la ressource de fichier que le programme tente d'ouvrir en vain.

Un programme bien écrit interceptera cette exception et informera l'utilisateur de l'erreur, demandant éventuellement un nom de fichier corrigé.

Lorsque ces exceptions surviennent, le compilateur Java nous demande de gérer. Nous devons soit lancer l'exception de manière déclarative dans la pile d'appels, soit la gérer nous-mêmes.

La plupart des classes et méthodes intégrées au langage Java sont codées de façon à gérer les exceptions qu'elles peuvent générer.

Les exceptions vérifiées/de compilations sont soumises à l'exigence de capture ou de spécification.

Quelques exemples d'exceptions vérifiées sont **IOException** et **ServletException**, **FileNotFoundException**...

2. Les exceptions non vérifiées/RuntimeException

Ce sont des exceptions qui sont internes au programme, et que le programme ne peut généralement pas anticiper ou surmonter.

Il n'est pas nécessaire de les manipuler. On doit donc les corriger manuellement ou enlever les codes causant ces exceptions.

Ce sont des bogues de programmation tels que des erreurs de logique, la mauvaise utilisation d'une API...

Exemple **Transmission d'un objet null comme fichier**

Par exemple, considérons l'application décrite précédemment. Si nous transmettons un objet *null* au lieu d'un fichier, nous obtenons une erreur décrite comme suit :

```
1 import java.io.FileReader;
2
3 public class FileNotFoundStudi {
4
5    public static void main(String args[]) throws FileNotFoundException {
6       File file = null;
7       FileReader fr = new FileReader(file);
8    }
9 }
```

Tous droits réservés à STUDI - Reproduction interdite

```
C:\>javac FilenotFoundStudi.java
java: Exception in thread "main" java.lang.NullPointerException
      FileReader fr = new FileReader(file);
                                     ^
1 error
```

Ici on transmet un objet *null* au fichier `file` qui attend un objet de type *File*. Ceci cause une exception.

L'application peut intercepter cette exception (ici **NullPointerException**), mais il est probablement préférable d'éliminer le bogue à l'origine de l'exception.

Les exceptions d'exécution ne sont pas soumises à l'exigence Catch ou Specify. Les exceptions d'exécution sont celles indiquées par **RuntimeException** et ses sous-classes.

Quelques exemples d'exceptions non vérifiées sont **NullPointerException**, **IllegalArgumentException** et **SecurityException**.

3. Les erreurs

Ce sont des exceptions qui sont externes au programme, et que le programme lui-même ne peut généralement pas anticiper ou récupérer.

En d'autres mots, les erreurs représentent des conditions graves et généralement irrécupérables comme une incompatibilité de bibliothèque, une récursion infinie ou des fuites de mémoire.

Exemple **Un fichier texte corrompu et impossible à lire**

Par exemple, supposons qu'une application récupère avec succès un fichier en entrée mais que le programme n'arrive pas à lire le fichier à cause d'un dysfonctionnent matériel ou système.

Un programme peut choisir d'intercepter cette exception afin d'avertir l'utilisateur du problème ou juste afficher le message d'erreur sur la console et s'arrête après.

Dans la plupart des cas, il serait étrange de gérer, d'instancier ou d'étendre les erreurs. Habituellement, nous voulons que celles-ci se propagent tout le long.

Les erreurs ne sont pas soumises à l'exigence de capture ou de spécification. Elles sont les exceptions indiquées par **Error** et ses sous-classes.

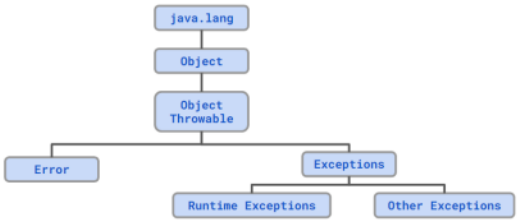
Quelques exemples d'erreurs sont **StackOverflowError** et **OutOfMemoryError**, **IOError**...

Toutes les exceptions sont vérifiées sauf celles indiquées par **Error**, **RuntimeException** et leurs sous-classes.

Hiérarchie des exceptions/La classe Throwable

Les exceptions ne sont que des objets Java, tous héritent de la classe **Throwable** :

Toutes les classes d'exception sont des sous-types de la classe **java.lang.Exception**. La classe d'exception est une sous-classe de la classe **Throwable**. Outre la classe d'exception, il existe une autre sous-classe appelée **Error** qui est dérivée de la classe **Throwable**. Le schéma ci-contre indique l'arborescence des différents types d'erreurs.



Tous droits réservés à STUDI - Reproduction interdite

Méthode **Les méthodes d'exception**

Voici la liste des méthodes importantes disponibles dans la classe **Throwable**. Ces méthodes sont très utiles pour la gestion des exceptions après leurs captures.

Méthodes	Description
public String getMessage( )	Renvoie un message détaillé sur l'exception qui s'est produite. Ce message est initialisé dans le constructeur **Throwable**.
public Throwable getCause( )	Renvoie la cause de l'exception représentée par un objet **Throwable**.
public String toString( )	Renvoie le nom de la classe concaténée avec le résultat de **getMessage()**.
public void printStackTrace( )	Imprime le résultat de **toString()** avec la trace de la pile dans **System.err**, le flux de sortie d'erreur.
public StackTraceElement [] getStackTrace( )	Renvoie un tableau contenant chaque élément de la trace de pile. L'élément à l'index 0 représente le haut de la pile d'appels et le dernier élément du tableau représente la méthode au bas de la pile d'appels.
public Throwable fillInStackTrace( )	Remplit la trace de pile de cet objet **Throwable** avec la trace de pile actuelle, en ajoutant toutes les informations précédentes dans la trace de pile.

Syntaxe **À retenir**

Dans ce chapitre, les points importants à retenir sont :

- Les méthodes d'un programme Java robuste et valide qui peuvent lever certaines exceptions doivent respecter les exigences **Capture (try, catch, finally)** des exceptions ou de la **spécification (throws/throw)** des exceptions.
- **Les exceptions vérifiées/de compilation** : ce sont des exceptions vérifiées par le compilateur au moment de la compilation et le programmeur est invité à gérer ces exceptions.
- **Les exceptions non vérifiées/RuntimeException** : les exceptions non vérifiées ne sont pas vérifiées au moment de la compilation, mais elles sont vérifiées à l'exécution.
- **Les erreurs/Error** : ce sont des exceptions qui sont externes au programme, et que le programme lui-même ne peut généralement pas anticiper ou récupérer.
- Toutes les exceptions sont vérifiées sauf celles indiquées par **Error (les erreurs), RuntimeException (erreurs d'exécutions)** et leurs sous-classes.
- La hiérarchie des exceptions où l'on voit que toutes les exceptions sont des sous-classes de la classe **Throwable.**

Tous droits réservés à STUDI - Reproduction interdite

V. Exercice : Appliquer la notion

Pour mettre en évidence la notion de la gestion des exceptions, nous allons faire une illustration très simple.

Question

[solution n°2 p.33]

Identifiez le type d'exception et dites si elle fait partie des exceptions vérifiées ou non vérifiées.

```
1 import java.io.File;
2 import java.io.FileReader;
3
4 public class FilenotFoundStudi {
5
6    public static void main(String args[]) {
7       File file = new File("E://file.txt");
8       FileReader fr = new FileReader(file);
9    }
10 }
```

```
1 C:\>javac FilenotFoundStudi.java
2 FilenotFound_Demo.java:8: error: unreported exception FileNotFoundException; must be caught or
  declared to be thrown
3       FileReader fr = new FileReader(file);
4                       ^
5 1 error
```

Indice :

- **Les exceptions vérifiés/de compilation** : ce sont des exceptions vérifiées par le compilateur au moment de la compilation et le programmeur est invité à gérer ces exceptions.
- **Les exceptions non vérifiées/RuntimeException** : les exceptions non vérifiées ne sont pas vérifiées au moment de la compilation, mais elles sont vérifiées à l'exécution.
- **Les erreurs / Error** : ce sont des exceptions qui sont externes au programme, et que le programme lui-même ne peut généralement pas anticiper ou récupérer.

VI. Le try-catch-finally (capture)

Objectifs

- Comprendre comment utiliser les trois composants du bloc **try-catch-finally** du gestionnaire d'exception pour la capture et la gestion des exceptions

Mise en situation

Nous voulons capter les exceptions afin de les gérer et permettre au programme de poursuivre son cours normalement, grâce à **try-catch-finally**.

Les mots-clés Try, Catch et Finally (capture)

L'instruction **try** est utilisée pour entourer le code susceptible de lever une exception. Elle doit être utilisée dans la méthode.

Le bloc **try** doit être suivi soit par **catch**, soit par **finally** ou les deux. Cela signifie que nous ne pouvons pas utiliser **try** tout seul.

L'instruction **catch** permet de définir un bloc de code à exécuter si une erreur se produit dans le bloc **try**.

Il doit être précédé du bloc **try**, ce qui signifie que nous ne pouvons pas utiliser le bloc catch seul. Il peut être suivi d'un bloc **finally** ou pas.

Tous droits réservés à STUDI - Reproduction interdite

L'instruction **finally** : elle permet de spécifier un bloc de code qui doit être exécuté dans tous les cas, avec ou sans exception. Cela peut être utile, généralement dans les scénarios où vous utilisez des ressources (base de données, entrées/sorties de fichiers...).

Elle est toujours précédé des blocs **try** et/ou **catch** et ne peut donc pas s'utiliser toute seule dans le cadre de la gestion des exceptions.

Syntaxe **Mots-clés try-catch : toujours par paires**

Une méthode intercepte une exception en utilisant une combinaison des mots-clés **try** et **catch**.

Le mot-clé **try** est utilisé pour entourer le code susceptible de lever une exception. Il doit être utilisé dans la méthode.

Le bloc **catch** est utilisé pour gérer l'exception en déclarant le type d'exception dans le paramètre. L'exception déclarée doit être l'exception de classe parente (c'est-à-dire, **Exception**) ou le type d'exception généré. Cependant, la bonne approche consiste à déclarer le type d'exception générée.

```
1 try {
2   //  Block de code pour exécuter du code qui pourrait lever une exception
3 }
4 catch(Exception e) {
5   //  Block de code pour gérer les exceptions captées
6 }
```

Le bloc **catch** doit être utilisé uniquement après le bloc **try**. Vous pouvez utiliser plusieurs blocs **catch** avec un seul bloc **try**.

Si une exception se produit au niveau de l'instruction particulière du bloc **try**, le reste du code de bloc ne s'exécutera pas. Il est donc recommandé de ne pas conserver le code dans le bloc **try** qui ne lèvera pas d'exception.

Exemple

On essaie d'afficher le dixième élément d'une liste qui n'en compte que 3.

```
1 public class Main {
2     public static void main(String[ ] args) {
3         int[] myNumbers = {1, 2, 3};
4         System.out.println(myNumbers[10]); // error!
5     }
6 }
```

```
1 C:\>javac Main.java
2 Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: Index 10 out of bounds
  for length 3
3   at exceptions.Main.main(Main.java)
```

Le programme vient de générer une exception de dépassement de la taille de tableau : **ArrayIndexOutOfBoundsException**.

Nous pouvons utiliser **try, catch** pour attraper l'erreur et exécuter du code pour gérer l'exception **ArrayIndexOutOfBoundsException** comme suit :

```
1 public class Main {
2     public static void main(String[ ] args) {
3         try {
4             int[] myNumbers = {1, 2, 3};
5             System.out.println(myNumbers[10]); // error!
6         } catch (ArrayIndexOutOfBoundsException exception) {
7             System.out.println("Nous venons ici de capturer l'exception
  ArrayIndexOutOfBoundsException et permettons ainsi au programme de ne pas s'arrêter
  brusquement");
```

Tous droits réservés à STUDI - Reproduction interdite

```
        }
    }
}
```

```
C:\>javac Main.java
Nous venons ici de capturer l'exception ArrayIndexOutOfBoundsException et permettons ainsi au programme de ne pas s'arrêter brusquement
```

Syntaxe **Try-catch-finally**

Parfois, nous avons du code qui doit s'exécuter indépendamment du fait qu'une exception se produise, et c'est là qu'intervient le mot-clé **finally** .

```
try {
  //  Block de code pour exécuter du code qui pourrait lever une exception
}
catch(Exception e) {
  //  Block de code pour gérer les exceptions captées
}
finally {
 //   Block de code à exécuter, après try...catch, quel que soit le résultat:
}
```

En général, les programmes se présenteront comme suit :

```
try {
    ExecuteMyCode();
}
catch (Exception exception){
    DoSomethingWithException(exception);
} finally {
    AlwaysExecuteMe();
}
```

Exemple

```
public class Main {
    public static void main(String[ ] args) {
        try {
            int[] myNumbers = {1, 2, 3};
            System.out.println(myNumbers[10]); // error!
        } catch (ArrayIndexOutOfBoundsException exception) {
            System.out.println("Nous venons ici de capturer l'exception ArrayIndexOutOfBoundsException et permettons ainsi au programme de ne pas s'arrêter brusquement");
        } finally {
             System.out.println("Je me montrerai quoi qu'il arrive avant");
        }
    }
}
```

```
C:\>javac Main.java
Nous venons ici de capturer l'exception ArrayIndexOutOfBoundsException et permettons ainsi au programme de ne pas s'arrêter brusquement
Je me montrerai quoi qu'il arrive avant
```


Tous droits réservés à STUDI - Reproduction interdite

Remarque **Capture de plusieurs exceptions**

Parfois, le code peut lever plus d'une exception, nous aurons donc autant de blocs **catch** séparés pour un bloc **try** :

```
1 try {
2     ExecuteMyCode();
3 } catch (InvalidOperationException ioe) {
4     DoSomethingWithException(ioe);
5 } catch (ArgumentException ae) {
6     DoSomethingWithException(ae);
7 } finally {
8     AlwaysExecuteMe();
9 }
```

On peut aussi attraper plusieurs exceptions dans un même bloc **catch** lorsque nous savons que la façon dont nous traiterons ces erreurs sera la même,

```
1 try {
2     ExecuteMyCode();
3 } catch (InvalidOperationException | ArgumentException exception) {
4    DoSomethingWithException(exception);
5 } finally {
6     AlwaysExecuteMe();
7 }
```

Remarque **Try imbriqué**

Parfois, une partie d'un bloc peut provoquer une erreur et le bloc entier lui-même peut provoquer une autre erreur. Dans de tels cas, les gestionnaires d'exceptions doivent être imbriqués.

```
1 try {
2     try {
3         ExecuteThis();
4     } catch (ThisException e) {
5         DoSomethongWithThisException();
6     }
7     try {
8         ExecuteThat();
9     } catch (ThatException e) {
10        DoSomethongWithThatException();
11    }
12    ExecuteMyCode();
13 } catch (InvalidOperationException | ArgumentException exception) {
14    DoSomethingWithException(exception);
15 } finally {
16    AlwaysExecuteMe();
17 }
```

Syntaxe **À retenir**

Cette section décrit comment utiliser les trois composants du gestionnaire d'exception : les blocs **try**, **catch** et **finally**.

- L'instruction **try** permet de définir un bloc de code à tester pour les erreurs lors de son exécution.
- L'instruction **catch** permet de définir un bloc de code à exécuter si une erreur se produit dans le bloc **try**.
- L'instruction **finally** permet de spécifier un bloc de code qui doit être exécuté dans tous les cas, avec ou sans exceptions.

Tous droits réservés à STUDI - Reproduction interdite

On a aussi vu qu'on pouvait faire des captures multiples et des blocs de **try** imbriqués.

VII. Exercice : Appliquer la notion

Cet exercice permet de mettre en évidence la notion de capture des exceptions (**try-catch-finally**).

Question

[solution n°3 p.33]

Indiquez les éléments manquants pour gérer l'erreur dans le code ci-dessous.

```
__ {
  int[] myNumbers = {1, 2, 3};
  System.out.println(myNumbers[10]);
} _____ (Exception e) {
  System.out.println("Something went wrong.");
}
________ {
  System.out.println("The 'try catch' is finished.");
}
```

Indice :

L'instruction **try** vous permet de définir un bloc de code à tester pour les erreurs lors de son exécution.

L'instruction **catch** vous permet de définir un bloc de code à exécuter, si une erreur se produit dans le bloc try.

L'instruction **finally** vous permet d'exécuter du code, après **s**, quel que soit le résultat.

VIII. Le throws/throw (spécification)

Objectifs

- Comprendre comment utiliser le mot-clé **throws** du gestionnaire d'exception pour la spécification des exceptions
- Comprendre l'utilisation du mot-clé **throw** pour la levée d'exceptions

Mise en situation

Nous savons déjà comment capturer et gérer les exceptions. Parfois, il est approprié que la méthode ne fasse que spécifier les exceptions qui peuvent s'y produire et déléguer la gestion aux autres méthodes.

Syntaxe **Throws**

Le mot-clé **throws** permet de spécifier les types d'exceptions qu'une méthode peut générer.

Cela ne lève pas d'exception. Il spécifie qu'il peut se produire une exception dans la méthode. Il est toujours utilisé avec la signature de méthode.

Ainsi, cela permet à une méthode de déléguer la gestion de l'exception via le mécanisme de **try...catch** à une méthode plus haut dans la pile d'appels.

Si une méthode ne gère pas d'exceptions, le type d'exceptions qui peuvent s'y produire doit être spécifié dans la clause de mot-clé **throws** dans la déclaration de la méthode afin que les méthodes situées plus haut dans la pile d'appels puissent les gérer ou les spécifier.

```
method (Arguments) throws Exception1,Exception2,Exception,... { ... }
```


Tous droits réservés à STUDI - Reproduction interdite

Exemple

Reprenons l'exemple du programme qui lit un fichier passé en paramètre qui est susceptible de générer une exception si le fichier est introuvable.

```
1 import java.io.File;
2 import java.io.FileInputStream;
3 import java.io.FileNotFoundException;
4
5 public class FileNotFoundStudi {
6
7     public static void findFile() throws FileNotFoundException {
8         // code that may produce IOException
9         File newFile = new File("fichierExistant.txt");
10        FileInputStream stream = new FileInputStream(newFile);
11    }
12
13    public static void main(String[] args) {
14        try {
15            findFile();
16        } catch (FileNotFoundException e) {
17            System.out.println(e);
18        }
19    }
20 }
```

```
1 C:\>javac FileNotFoundStudi.java
2 java.io.FileNotFoundException: fichierExistant.txt (Le fichier spécifié est introuvable)
```

Ici, la méthode **findFile( )** spécifie qu'une exception de type **FileNotFoundException** peut être levée. La méthode **main()** appelle cette méthode et gère l'exception si elle est levée.

Syntaxe **Throw**

Le mot-clé **throw** permet de lever explicitement une exception vérifiée ou non vérifiée.

```
1 throw new Exception("Exception message");
```

Exemple **Levée d'une exception vérifiée**

Reprenons encore une fois notre exemple de programme qui tente de lire son fichier.

```
1 import java.io.FileNotFoundException;
2
3 public class FileNotFoundStudi {
4
5     public static void findFile() throws FileNotFoundException {
6         throw new FileNotFoundException("File not found Studi new exception!");
7     }
8
9     public static void main(String[] args) {
10        try {
11            findFile();
12            System.out.println("Le reste du code dans le bloc try qui ne s'exécutera pas si
  l'exception ci-dessus est interceptée");
13        } catch (FileNotFoundException e) {
14            System.out.println(e);
15        }
16    }
17 }
```

Tous droits réservés à STUDI - Reproduction interdite

```
C:\>javac FileNotFoundStudi.java
java.io.FileNotFoundException: File not found Studi new exception!
```

La méthode **findFile( )** lève une exception de type **FileNotFoundException** avec le message que nous avons passé à son constructeur de la méthode **FileNotFoundException( )**.

Notez que puisqu'il s'agit d'une exception vérifiée, nous devons la spécifier dans la clause du mot-clé **throws**.

Exemple **Levée d'une exception non vérifiée**

On veut qu'un programme lève une exception si l'âge est inférieur à 18 ans (imprimez « Accès refusé aux mineurs ! »). Si l'âge est supérieur à 18 ans ou plus, imprimez « Accès autorisé car vous avez l'âge requis ! » :

```java
public class Main {
    static void checkAge(int age) {
        if (age < 18) {
            throw new ArithmeticException("Accès refusé aux mineures!");
        }
        else {
            System.out.println("Accès autorisé car vous avez l'âge requis!");
        }
    }

    public static void main(String[] args) {
        try {
            checkAge(15); // Set age to 15 (which is below 18...)
        } catch (ArithmeticException e) {
            System.out.println(e);
        }
    }
}
```

```
C:\>javac Main.java
java.lang.ArithmeticException: Accès refusé aux mineurs!
```

La méthode **checkAge(int age)** lève une exception de type **ArithmeticException** avec le message que nous avons passé à son constructeur de la méthode **ArithmeticException( )**.

Notez que puisqu'il s'agit d'une exception non vérifiée, nous ne sommes pas obligés de la spécifier dans la clause du mot-clé **throws**.

Syntaxe **À retenir**

- Le mot-clé **throws** est utilisé pour spécifier les exceptions pouvant survenir lors de l'exécution d'un programme
- le mot-clé **throw** est utilisé pour lever une nouvelle exception et la lancer dans le bloc **catch**.

IX. Exercice : Appliquer la notion

Cet exercice à pour but de mettre en évidence la notion de spécification des exceptions (**throws**/**throw**).

Tous droits réservés à STUDI - Reproduction interdite

Question

[solution n°4 p.33]

Indiquez les éléments manquants pour spécifier (**throws**) et lever (**throw**) les exceptions dans le code ci-dessous.

```
1 import java.io.IOException;
2
3 public class Main {
4
5     static class Device {
6         void method() ______ IOException {
7             ______ new IOException("Device error");
8         }
9     }
10
11    public static void main(String[] args) _____ IOException {
12        try {
13            Device device = new Device();
14            device.method();
15            System.out.println("Normal flow");
16        } catch (IOException e) {
17            System.out.println(e.getMessage());
18        }
19
20    }
21 }
```

Indice :

- Le mot-clé **throws** permet de spécifier les types d'exceptions qu'une méthode peut générer.
- Le mot-clé **throw** permet de lever explicitement une exception vérifiée ou non vérifiée.

X. Le try-with-resources

Objectifs

- Comprendre le mécanisme du **try-with-resources** du gestionnaire d'exceptions pour la capture et la gestion des exceptions
- Comprendre pourquoi et quand utiliser le bloc **try-with-resources** par rapport au mécanisme **try-catch-finally**

Mise en situation

Le **try-with-resources** : il permet de déclarer les ressources à utiliser dans un bloc **try** avec l'assurance que les ressources seront fermées après l'exécution de ce bloc.

Syntaxe **Utilisation**

En termes simples, pour être fermée automatiquement, une ressource doit être à la fois déclarée et initialisée à l'intérieur de la clause du mot-clé **try**.

```
1 try (resource declaration) {
2   // use of the resource
3 } catch (ExceptionType e1) {
4   // catch block
5 }
```

Tous droits réservés à STUDI - Reproduction interdite

Exemple **Try...catch...finally**

Généralement, lorsque nous utilisons des ressources telles que des flux, des connexions, etc., nous devons les fermer explicitement en utilisant le bloc **finally**.

```
1 import java.io.*;
2
3 public class Main {
4     public static void main(String[] args) {
5         BufferedReader br = null;
6         String line;
7         try {
8             System.out.println("Entering try block");
9             br = new BufferedReader(new FileReader("fichierInexistant.txt"));
10            while ((line = br.readLine()) != null) {
11                System.out.println("Line =>"+line);
12            }
13        } catch (IOException e) {
14            System.out.println("IOException in try block => " + e.getMessage());
15        } finally {
16            System.out.println("Entering finally block");
17            try {
18                if (br != null) {
19                    br.close();
20                }
21            } catch (IOException e) {
22                System.out.println("IOException in finally block => " + e.getMessage());
23            }
24        }
25    }
26 }
```

```
1 Entering try block
2 IOException in try block => fichierInexistant.txt (Le fichier spécifié est introuvable)
3 Entering finally block
```

Comme nous pouvons le voir dans l'exemple ci-dessus, l'utilisation du bloc **finally** pour nettoyer les ressources rend le code plus complexe.

Avez-vous également remarqué le bloc **try-catch** dans le bloc **finally** ?

En effet, une exception de type **IOException** peut également se produire lors de la fermeture de l'instance **BufferedReader** à l'intérieur de ce bloc.

L'instruction **try-with-resources** effectue une gestion automatique des ressources.

Nous n'avons pas besoin de fermer explicitement les ressources car JVM les ferme automatiquement. Cela rend le code plus lisible et plus facile à écrire.

Exemple **Utilisation du try-with-resources à la place du try...catch...finally**

Remplaçons le code du **try-catch-finally** de l'exemple ci-dessus par **try-with-resources** :

```
1 import java.io.BufferedReader;
2 import java.io.FileReader;
3 import java.io.IOException;
4
5 public class FileNotFoundStudi {
6     public static void main(String[] args) {
7         String line;
8         try(BufferedReader br = new BufferedReader(new FileReader("fichierInexistant.txt"))) {
9             while ((line = br.readLine()) != null) {
```


Tous droits réservés à STUDI - Reproduction interdite

```
                System.out.println("Line =>"+line);
            }
        } catch (IOException e) {
            System.out.println("IOException in try block =>" + e.getMessage());
        }
    }
}
```

```
IOException dans le block try => fichierInexistant.txt (Le fichier spécifié est introuvable)
```

Le programme est maintenant succinct.

Remarque

Pour utiliser une classe avec une instruction **try-with-resources**, elle doit implémenter l'interface **AutoCloseable** et la méthode **close()** de celle-ci est appelée automatiquement lors de l'exécution.

Vous pouvez déclarer plusieurs classes dans l'instruction **try-with-resources**.

```
import java.io.*;
import java.util.*;

public class Main {
    public static void main(String[] args) throws IOException{
        try (Scanner scanner = new Scanner(new File("testRead.txt"));
             PrintWriter writer = new PrintWriter(new File("testWrite.txt"))) {
            while (scanner.hasNext()) {
                writer.print(scanner.nextLine());
            }
        } catch (FileNotFoundException e) {
            System.out.println(e.getMessage());
        }
    }
}
```

```
testRead.txt (Le fichier spécifié est introuvable)
```

La ressource déclarée au bloc **try** est implicitement déclarée comme finale.

Syntaxe **À retenir**

- L' instruction **try-with-resources** ferme toutes les ressources qui implémentent l'interface **AutoCloseable**.
- Avec le **try-with-resources**, le bloc du **finally** n'est plus obligé de fermer les ressources et rend le code plus succinct par rapport au **try-catch-finally**.

XI. Exercice : Appliquer la notion

Cet exercice met en évidence la notion de **try-with-resources**.

Question

[solution n°5 p.34]

Indiquez les éléments manquants pour capter les exceptions en utilisant le **try-with-resources**.

```
import java.io.FileReader;
import java.io.IOException;

public class Main {

    public static void main(String[] args) {
```

Tous droits réservés à STUDI - Reproduction interdite

```
7         ___(FileReader fr = new FileReader("fichierInexistant.txt")) {
8             char [] a = new char[50];
9             fr.read(a);
10            for(char c : a)
11                System.out.print(c);
12        } ____(IOException e) {
13            System.out.println(e.getMessage());
14        }
15    }
16 }
```

Indice :

- Le **try-with-resources** permet de fermer les ressources utilisées automatiquement.

XII. Les exceptions personnalisées

Objectifs

- Créer les exceptions personnalisées
- Comprendre les exceptions personnalisées (vérifiées et non vérifiées)

Mise en situation

Les exceptions Java couvrent presque toutes les exceptions générales qui sont liées à la programmation.

Cependant, nous devons parfois compléter ces exceptions standards par les nôtres.

Syntaxe **Création d'une exception**

Nous pouvons définir notre propre classe d'exception comme ci-dessous :

```
1 class MyException extends Exception { ... }
```

Il suffit d'étendre la classe d'exception prédéfinie pour créer votre propre exception

Conseil

Toutes les exceptions doivent étendre la classe **Throwable**.

Exceptions vérifiées/non vérifiées personnalisées : pour créer une exception non vérifiée personnalisée, nous devons étendre la classe **java.lang.Exception**.

Exemple **Exceptions vérifiées personnalisées**

Les exceptions **Checked** sont des exceptions qui doivent être traitées explicitement.

```
1 public class IncorrectFileNameException extends Exception {
2     public IncorrectFileNameException(String errorMessage) {
3         super(errorMessage);
4     }
5 }
```

Notez que nous devons également fournir un constructeur qui prend une chaîne comme message d'erreur et appelle le constructeur de classe parent.

Tous droits réservés à STUDI - Reproduction interdite

Exemple **Les exceptions non vérifiées personnalisées**

Reprenons l'exemple du programme qui tente en vain d'ouvrir un fichier.

Supposons que nous ayons besoin d'une exception personnalisée si le nom de fichier ne contient aucune extension.

```
1 public class IncorrectFileExtensionException
2   extends RuntimeException {
3     public IncorrectFileExtensionException(String errorMessage, Throwable err) {
4         super(errorMessage, err);
5     }
6 }
```

Complément **Utilisation des exceptions personnalisées**

Voyons comment nous pouvons utiliser les exceptions personnalisées que nous venons d'écrire dans notre exemple de programme lecteur de fichier.

```
1 import java.io.File;
2 import java.io.FileNotFoundException;
3 import java.util.Scanner;
4
5 public class FileNotFoundStudi {
6
7     public static void main(String[] args) throws IncorrectFileNameException {
8         String fileName = "fichierInexistant.txt";
9         try (Scanner file = new Scanner(new File(fileName))) {
10            if (file.hasNextLine()) {
11                file.nextLine();
12            } else {
13                throw new IllegalArgumentException("Non readable file");
14            }
15        } catch (FileNotFoundException err) {
16                throw new IncorrectFileNameException("Incorrect filename : ", err);
17        } catch (IllegalArgumentException err) {
18                throw new IncorrectFileExtensionException("Filename does not contain extension
  : ", err);
19            }
20        }
21
22    static class IncorrectFileNameException extends Exception {
23        public IncorrectFileNameException(String errorMessage, FileNotFoundException err) {
24            super(errorMessage);
25        }
26    }
27
28    static class IncorrectFileExtensionException
29            extends RuntimeException {
30        public IncorrectFileExtensionException(String errorMessage, Throwable err) {
31            super(errorMessage, err);
32        }
33    }
34 }
```

Tous droits réservés à STUDI - Reproduction interdite

Syntaxe **À retenir**

Vous pouvez créer vos propres exceptions en Java. Gardez les points suivants à l'esprit lorsque vous écrivez vos propres classes d'exceptions :

- Toutes les exceptions doivent hériter de la classe **Throwable**.
- Si vous souhaitez écrire une exception vérifiée, vous devez étendre la classe **Exception**.
- Si vous souhaitez écrire une exception d'exécution, vous devez étendre la classe **RuntimeException**.

XIII. Exercice : Appliquer la notion

Avec cet exercice, mettez en application la notion d'exceptions personnalisées.

Question

[solution n°6 p.34]

Indiquez les éléments manquants pour définir une nouvelle exception de type **InsufficientFundsException**.

```
1 import java.io.IOException;
2
3 import java.io.*;
4
5 public class _________________________ extends _________ {
6    private double amount;
7
8    public InsufficientFundsException(double amount) {
9       this.amount = amount;
10   }
11
12   public double getAmount() {
13      return amount;
14   }
15 }
```

Indice :

- Si vous souhaitez écrire une exception vérifiée, vous devez étendre la classe **Exception**.
- Si vous souhaitez écrire une exception d'exécution, vous devez étendre la classe **RuntimeException**.

XIV. Essentiel

XV. Auto-évaluation

A. Exercice final

Exercice 1

[solution n°7 p.35]

Exercice

Tous droits réservés à STUDI - Reproduction interdite

Quelle assertion est erronée à propos du bloc **try** ?

- Les codes du bloc **try** peuvent lever plusieurs exceptions.
- Le bloc **try** doit être avant les blocs **catch** et **finally**.
- Les codes du bloc **try** ne doivent jamais inclure de boucles.
- Il peut y avoir des codes à l'intérieur du bloc **try** qui ne peuvent pas lever d'exceptions.
- Aucune réponse n'est bonne.

Exercice

Quelle assertion est erronée à propos du bloc **catch** ?

- Le bloc **catch** qui capte une classe d'exception enfant doit faire succéder le bloc **catch** d'une exception de classe parente.
- Le bloc **catch** qui intercepte une classe d'exception enfant doit précéder au bloc **catch** d'une exception de classe parent.
- S'il n'y a pas de bloc **catch** après un bloc **try**, il doit y avoir un bloc **finally**.
- Il n'y a aucune bonne réponse.

Exercice

Le code suivant est-il correct ?

```
1 try {
2
3 } finally {
4
5 }
```

- **Oui**, un bloc **try** n'a pas obligatoirement besoin d'un bloc **catch** s'il y a un bloc **finally**.
- **Non**, car on ne peut avoir un bloc **finally** sans un bloc **catch** qui le précède.
- **Oui**, car seul le bloc **try** est requis pour spécifier les exceptions, tandis que les blocs **catch** et **finally** sont facultatifs ; on peut les enlever tous les deux sans problème.
- **Non**, car les blocs **try** et **finally** ne peuvent être présents ensemble que si un bloc **catch** est là.
- **Aucune bonne réponse**

Exercice

Un programme doit-il nécessairement lever toutes les exceptions qui peuvent survenir ?

- **Oui**, si un programme ne lève pas toutes les exceptions possibles, il ne compilera pas.
- **Non**, mais si un programme lève un type d'exception, il doit également lever tous les autres types.
- **Non**, un programme peut uniquement lever les exceptions que vous souhaitez.
- **Oui**, sinon cela ne compilera pas car la JVM n'aime pas ce genre de pratique.
- **Aucune bonne réponse**

Exercice

Tous droits réservés à STUDI - Reproduction interdite

Quelle est la méthode de la classe **Throwable** qui renvoie le contenu du message d'erreur ?

- ○ `getError( )`
- ○ `getException( )`
- ○ `getMessage( )`
- ○ `traceMessage( )`
- ○ Aucune bonne réponse

Exercice

Quel(s) sont les types d'exceptions qui ne sont pas vérifiés ?

- ☐ La classe **Throwable** et ses sous-classes
- ☐ La classe **RuntimeException** et ses sous-classes
- ☐ La classe **IOException** et ses sous-classes
- ☐ La classe **Error** et ses sous-classes
- ☐ La classe **ArithmeticException** uniquement

Exercice

Quelles sont les deux options qu'une méthode doit présenter pour gérer une exception ?

- ○ **Option 1**. Ignorer ces exceptions est généralement la bonne chose à faire.
  Option 2. Vérifier ces exceptions, peut-être avons-nous fait une erreur.
- ○ **Option 1**. Gérer l'exception dans un bloc **catch**.
  Option 2. Renvoyer la gestion de l'exception à la méthode expéditrice.
- ○ **Option 1**. Gérer l'exception dans un bloc **catch**.
  Option 2. Spécifier l'exception à l'aide du mot-clé **throws**.
- ○ **Option 1**. Gérer l'exception dans un bloc **catch**.
  Option 2. Gérer l'exception dans un bloc **try**.
- ○ Aucune bonne réponse

Exercice

Y a-t-il un problème avec ce gestionnaire d'exceptions tel qu'il est écrit ? Dites si ce code sera compilé.

```
1 try {
2
3 } catch (Exception e) {
4
5 } catch (ArithmeticException a) {
6
7 }
```

- ○ **Oui**, car un bloc **try** peut lever plusieurs exceptions et ces exceptions doivent être gérées par plusieurs blocs **catch**.
- ○ **Non**, car l'exception de type **Exception** englobe l'exception de type **ArithmeticException** dont elle est la classe parente.

Tous droits réservés à STUDI - Reproduction interdite

- **Non**, car l'exception de type **ArithmeticException** qui est un sous-type d'**Exception** doit être interceptée et l'exception de type **Exception** ne doit pas être interceptée en plus.
- **Oui**, car on capture distinctement les exceptions. Donc ce code va compiler sans aucun problème.
- **Aucune bonne réponse**

Exercice

Quel(s) type(s) d'exception(s) peut engendrer le code du programme de calcul de la masse corporelle ci-dessous ?

```
1 public class Main {
2     public static void main(String[] args) {
3         int taille = 12;
4         int poids = 0;
5         System.out.println("Votre IMC ==> " + taille / poids);
6     }
7 }
```

- [ ] Erreur de compilation : les divisions doivent être dans un bloc **try**
- [ ] Erreur d'exécution : **RuntimeException**
- [ ] Erreur de compilation : **DivideByZeroException**
- [ ] Erreur d'exécution :**ArithmeticException**
- [ ] Aucune bonne réponse

Exercice

Quel sera le résultat de ce programme ?

```
1 public class Main {
2
3     public static void myMethod() throws Exception {
4         System.out.println("ThrowIt");
5         throw new Exception();
6     }
7     public static void main(String[] args) {
8         try {
9             System.out.println("Hello World!");
10            myMethod();
11        } catch (Exception e) {
12            System.out.println("Caught");
13        } finally {
14            System.out.println("finally");
15        }
16        System.out.println("Good bye!");
17    }
18 }
```

- [ ] La compilation échoue, trop de levée d'exceptions.
- [ ] ThrowIt
  Hello World!
  Caught
  finally
  Good bye!

Tous droits réservés à STUDI - Reproduction interdite

- ☐ ThrowIt
  Hello World!
  Caught
  finally
- ☐ Hello World!
  ThrowIt
  Caught
  finally
  Good bye!
- ☐ Aucune bonne réponse

Exercice

Supposons qu'une méthode `A( )` appelle une méthode `B( )` ; et que la méthode `B( )` appelle à son tour une méthode `C( )` qui peut lever une exception de type **Exception**.

Peut-on reporter l'exception de sorte que la méthode `A( )` puisse la gérer ?

```
1 public class Main {
2
3     public static void A() {
4         B();
5     }
6
7     public static void B() {
8         C();
9     }
10
11    public static void C() {
12        throw new Exception("Some exceptions in method C");
13    }
14
15    public static void main(String[] args) throws Exception {
16        A();
17    }
18 }
```

- ☐ **Non**, la méthode `C( )` doit gérer seule son exception car les exceptions doivent toujours obligatoirement être gérées par les méthodes ou codes susceptibles de les générer.
- ☐ **Non**, on doit obligatoirement gérer les exceptions dans le `main( )`, c'est la règle.
- ☐ **Oui**, si les en-têtes des méthodes `B( )` et `C( )` spécifient l'exception dans leur déclaration à l'aide du mot-clé **throws**.
- ☐ **Oui**, si et seulement la méthode `C( )` spécifie l'exception dans sa déclaration à l'aide du mot-clé **throws**.
- ☐ **Aucune bonne réponse**

B. Exercice : Défi

Dans cet exercice, nous allons concevoir un programme Java robuste qui couvre tous les aspects de la conception du code de gestion des exceptions.

Imaginons un programme mettant en œuvre les fonctionnalités de base des distributeurs automatiques de billets dans la région de Montpellier.

Une des règles fondamentales que le programme doit implémenter est que l'on ne puisse pas retirer de l'argent dont le montant dépasse le solde bancaire du compte courant.

Tous droits réservés à STUDI - Reproduction interdite

On a proposé un code qui fait ce qui lui est demandé et ne fait rien lorsque lorsque l'on veut retirer un montant supérieur à celui du compte courant.

Le Team Manager a donc décidé de vous confier la responsabilité de corriger les erreurs du programme car il sait que vous avez suivi notre cours d'exceptions.

Votre défi est donc de corriger le programme suivant et également d'utiliser les bonnes pratiques du mécanisme de gestion des exceptions proposé par Java afin que le programme puisse fonctionner à nouveau et implémenter correctement la règle sur les retraits.

Question

[solution n°8 p.40]

Vous trouverez ci-dessous le programme Java contenant une classe représentant un compte bancaire lambda dans lequel les utilisateurs peuvent afficher, déposer ou retirer de l'argent.

Le code ci-dessous est utilisé pour enregistrer les transactions et ne fait rien lorsqu'un utilisateur essaie de retirer un montant supérieur à celui qu'il a dans son compte.

Nous avons donc trois problèmes :

1. Comment gérer les exceptions
2. Comment utiliser le mécanisme d'exceptions pour implémenter le signalement à l'utilisateur qu'il ne doit retirer qu'un montant inférieur à celui qu'il a sur son compte.
3. Gérer les ressources utilisées correctement et capturer les exceptions qu'on va lever afin que le programme puisse fonctionner correctement.

```
1 import java.util.Scanner;
2
3 public class Compte {
4     private int solde = 100;
5
6     public static void main(String[] args) throws InterruptedException {
7         Compte compte = new Compte();
8         System.out.println("""
9                 Bonjour et Bienvenue dans notre banque
10                \tTapez 1 ==> Vérifiez le solde de votre compte
11                \tTapez 2 ==> Faire un dépôt
12                \tTapez 3 ==> Faire un retrait
13                \tTapez 4 pour enregistrer les opérations effectuées et quitter l'application
""");
14        Scanner sc = new Scanner(System.in);
15        boolean isWrongAnswer = true;
16        while (isWrongAnswer) {
17
System.out.println("*******************************************************************************");
18            System.out.println("Veuillez saisir le numéro de la transaction que vous souhaitez
effectuer:");
19            switch (sc.nextInt()) {
20                case 1 -> compte.soldeActuel();
21                case 2 -> {
22                    System.out.println("Entrez le montant du dépôt");
23                    compte.depot(sc.nextInt());
24                }
25                case 3 -> {
26                    System.out.println("Entrez le montant du retrait");
27                    compte.retrait(sc.nextInt());
28                }
29                case 4 -> {
30                    System.out.println("Opération(s) enregistrée(s)! Au revoir");
31                    Thread.sleep(2000);
32                    isWrongAnswer = false;
33                }
```

Tous droits réservés à STUDI - Reproduction interdite

```
34                 default -> {
35                     System.out.println("Tapez uniquement les nombres entre 1 et 4");
36                 }
37             }
38         }
39     }
40
41     /* Le solde doit toujours être positif */
42     public int getSolde() {
43         return solde;
44     }
45
46     /* Le montant à retirer doit toujours être > le solde actuel */
47     public void retrait(int montant) {
48         this.solde -= montant;
49         soldeActuel();
50     }
51
52     public void depot(int montant) {
53         this.solde += montant;
54         soldeActuel();
55     }
56
57     public void soldeActuel() {
58         System.out.println("Le solde du compte courant est de " + this.getSolde() + " euros");
59     }
60
61 }
```

```
1 C:\>javac Compte.java
2 java: unreported exception java.lang.InterruptedException; must be caught or declared to be
  thrown
```

Nous devons donc traiter tous les problèmes liés à ce programme en utilisant le mécanisme de gestion des exceptions Java que nous avons appris dans le cours.

Indice :

1. Le compilateur affiche une exception de type **InterruptedException**.

La plupart des classes et méthodes intégrées au langage Java sont codées de façon à gérer les exceptions qu'elles peuvent générer. Et leurs exceptions sont dites vérifiées.

Nous allons donc ici soit gérer l'exception **InterruptedException** générée par la méthode `sleep( )` de la classe **Thread** (classe intégrée du langage Java), soit la propager (la spécifier).

```
1 public static void main(String[] args) throws InterruptedException { ... }
```

Nous avons donc ajouté la spécification de l'exception **InterruptedException** dans la déclaration de la méthode `main( )` à l'aide du mot-clé **throws** afin de la propager et de déléguer son traitement à une méthode située plus haut dans la liste des appels de méthode.

```
1 C:\>javac Compte.java
2 Bonjour et Bienvenue dans notre banque
3   Tapez 1 ==> Vérifiez le solde de votre compte
4   Tapez 2 ==> Faire un dépôt
5   Tapez 3 ==> Faire un retrait
6   Tapez 4 pour enregistrer les opérations effectuées et quitter l'application
7 ******************************************************************************
8   Veuillez saisir le numéro de la transaction que vous souhaitez effectuer:
9 1
10 Le solde du compte courant est de 100 euros
11 ******************************************************************************
```


Tous droits réservés à STUDI - Reproduction interdite

```
  Veuillez saisir le numéro de la transaction que vous souhaitez effectuer:
3
Entrez le montant du retrait
250
Le solde du compte courant est de -150 euros
*******************************************************************************
  Veuillez saisir le numéro de la transaction que vous souhaitez effectuer:
4
Opération(s) enregistrée(s)! Au revoir

Process finished with exit code 0
```

Le code compile, cependant, on voit bien que l'on a effectué un retrait de 250 euros sur un compte au solde de 100 euros. Après cette opération, le solde affiche -150 euros

Pour corriger cela, il ne reste plus qu'à mettre en place la gestion de la règle sur les retraits d'argent.

Indice :

2. L'implémentation de cette règle peut se faire par un mécanisme de gestion d'exception personnalisée.

Nous allons créer une exception personnalisée appelée **MontantSoldeDoitEtreSuperieurAuMontantRetraitException**.

```java
class MontantSoldeDoitEtreSuperieurAuMontantRetraitException extends Exception {
    private String message;

    public MontantSoldeDoitEtreSuperieurAuMontantRetraitException(String message) {
        this.message = message;
    }

    public String getMessage() {
        return this.message;
    }
}
```

Nous allons donc l'appliquer dans notre fonction de retrait de la classe compte comme suit :

```java
    /* Le montant à retirer doit toujours être > au solde actuel */
    public void retrait(int montant) {
        if (montant > solde) {
            throw new MontantSoldeDoitEtreSuperieurAuMontantRetraitException (
                    "Le solde actuel de " + compte.getSolde( ) + " euros est inférieur par rapport au montant de " + montant + " euros que vous souhaitez retirer");
        }
        this.solde -= montant;
        soldeActuel();
    }
```

En effet, nous levons notre exception à l'aide du mot-clé **throw**. Ainsi à l'aide de ce code, nous générerons une exception permettant au programme de gérer les retraits supérieurs au solde courant.

Ensuite on spécifie l'exception ainsi levée partout où la méthode (la méthode `retrait( )`) qui la lève est appelée.

```java
public void retrait(int montant) throws MontantSoldeDoitEtreSuperieurAuMontantRetraitException { ... }
```

```java
public static void main(String[] args) throws InterruptedException, MontantSoldeDoitEtreSuperieurAuMontantRetraitException { ... }
```

```
C:\>javac Compte.java
Bonjour et Bienvenue dans notre banque
  Tapez 1 ==> Vérifiez le solde de votre compte
  Tapez 2 ==> Faire un dépôt
  Tapez 3 ==> Faire un retrait
  Tapez 4 pour enregistrer les opérations effectuées et quitter l'application
```

Tous droits réservés à STUDI - Reproduction interdite

```
Veuillez saisir le numéro de la transaction que vous souhaitez effectuer:
3
Entrez le montant du retrait
400
Exception in thread "main" exceptions.MontantSoldeDoitEtreSuperieurAuMontantRetraitException
  at exceptions.Compte.retrait(Compte.java:50)
  at exceptions.Compte.main(Compte.java:28)

Process finished with exit code 1
```

On voit maintenant que lorsqu'un utilisateur tente de retirer 400 euros alors que son solde est de 100 euros, le compilateur Java affiche notre exception qui défend cette opération.

Nous allons maintenant gérer notre exception à l'aide du mécanisme de capture et de gestion des exception des exceptions en Java, le **try-catch-finally.**

Indice :

3. Capture et gestion des exceptions

- Ce code utilise la ressource **scanner** sans pour autant la libérer à la fin de nos opérations.
- Ensuite on peut directement capter et afficher le message de l'exception personnalisée écrite ci-dessus dans la `main( )` de l'application.

Procédons d'abord par la capture de notre exception personnalisée de type **MontantSoldeDoitEtreSuperieurAuMontantRetraitException** et libérer la ressource scanner qu'on utilise.

```
        ...
        Scanner sc = new Scanner(System.in);
        try {
            ...
        } catch (MontantSoldeDoitEtreSuperieurAuMontantRetraitException exception) {
            System.out.println(exception.getMessage());
        } finally {
            sc.close();
        }
    }
```

On remarque ici qu'on a enlevé la spécification de l'exception **MontantSoldeDoitEtreSuperieurAuMontantRetraitException** dans la déclaration de la méthode `main( )` puisque maintenant on a capté et géré l'exception en question à l'aide du mécanisme du **try-catch-finally.**

On a aussi libéré la ressource du scanner dans le bloc de code **finally.**

Indice :

Pour finir, nous allons remplacer le mécanisme du **try...catch...finally** par celui du **try-with-resources** pour rendre le code plus succinct.

```
try (Scanner sc = new Scanner(System.in)) { ... }
```

Solutions des exercices

Tous droits réservés à STUDI - Reproduction interdite

p. 7 Solution n°1

Erreur de compilation

```
C:\>javac FileNotFoundStudi.java
java: error: unreported exception FileNotFoundException; must be caught or declared to be thrown
      FileReader fr = new FileReader(file);
                      ^
1 error
```

Le résultat de la console ici indique : Java lève une exception de type **FileNotFoundException** et le compilateur javac demande à ce que cette exception soit gérée.

Type d'erreur :**erreur de compilation**

Approche de résolution :**gestion d'exception**

p. 12 Solution n°2

Exceptions vérifiés / de compilation

```
C:\>javac FilenotFoundStudi.java
FilenotFound_Demo.java:8: error: unreported exception FileNotFoundException; must be caught or declared to be thrown
      FileReader fr = new FileReader(file);
                      ^
1 error
```

Type d'exception : **Exceptions vérifiées/de compilation**

Le résultat de la console ici indique : Java lève une exception de type **FileNotFoundException** et le compilateur de Java **demande/exige** que cette exception soit gérée (**must be caught or declared to be thrown**).

p. 16 Solution n°3

```
try {
  int[] myNumbers = {1, 2, 3};
  System.out.println(myNumbers[10]);
} catch (Exception e) {
  System.out.println("Something went wrong.");
}
finally {
  System.out.println("The 'try catch' is finished.");
}
```

p. 19 Solution n°4

```
import java.io.IOException;

public class Main {

    static class Device {
        void method() throws IOException {
            throw new IOException("Device error");
        }
```

Tous droits réservés à STUDI - Reproduction interdite

```
9     }
10
11    public static void main(String[] args) throws IOException {
12        try {
13            Device device = new Device();
14            device.method();
15            System.out.println("Normal flow");
16        } catch (IOException e) {
17            System.out.println(e.getMessage());
18        }
19
20    }
21 }
```

```
1 C:\>javac Main.java
2 Device error
```

p. 21 Solution n°5

```
1 import java.io.FileReader;
2 import java.io.IOException;
3
4 public class Main {
5
6     public static void main(String[] args) {
7         try(FileReader fr = new FileReader("fichierInexistant.txt")) {
8             char [] a = new char[50];
9             fr.read(a);
10            for(char c : a)
11                System.out.print(c);
12        } catch (IOException e) {
13            System.out.println(e.getMessage());
14        }
15    }
16 }
```

```
1 C:\>javac Main.java
2 fichierInexistant.txt (Le fichier spécifié est introuvable)
```

p. 24 Solution n°6

```
1 import java.io.*;
2
3 public class InsufficientFundsException extends Exception {
4    private double amount;
5
6    public InsufficientFundsException(double amount) {
7       this.amount = amount;
8    }
9
10   public double getAmount() {
11      return amount;
12   }
13 }
```


Tous droits réservés à STUDI - Reproduction interdite

Exercice p. 24 Solution n°7

Exercice

Quelle assertion est erronée à propos du bloc **try** ?

- ○ Les codes du bloc **try** peuvent lever plusieurs exceptions.
- ○ Le bloc **try** doit être avant les blocs **catch** et **finally**.
- ◉ Les codes du bloc **try** ne doivent jamais inclure de boucles.
- ○ Il peut y avoir des codes à l'intérieur du bloc **try** qui ne peuvent pas lever d'exceptions.
- ○ Aucune réponse n'est bonne.

Q L'assertion 3 est fausse car on peut implémenter bel et bien n'importe quel code à l'intérieur d'un bloc **try** y compris les boucles. Il est même possible de mettre des **try-catch** dans un bloc **try** !

Exercice

Quelle assertion est erronée à propos du bloc **catch** ?

- ◉ Le bloc **catch** qui capte une classe d'exception enfant doit faire succéder le bloc **catch** d'une exception de classe parente.
- ○ Le bloc **catch** qui intercepte une classe d'exception enfant doit précéder au bloc **catch** d'une exception de classe parent.
- ○ S'il n'y a pas de bloc **catch** après un bloc **try**, il doit y avoir un bloc **finally**.
- ○ Il n'y a aucune bonne réponse.

Q En effet, lors de la capture des exceptions, les exceptions les plus spécifiques doivent être capturées avant les plus générales. Autrement dit, les sous-classes doivent être capturées avant leurs superclasses et pas l'inverse comme à l'assertion 2.

Exercice

Le code suivant est-il correct ?

```
1 try {
2
3 } finally {
4
5 }
```

- ◉ **Oui**, un bloc **try** n'a pas obligatoirement besoin d'un bloc **catch** s'il y a un bloc **finally**.
- ○ **Non**, car on ne peut avoir un bloc **finally** sans un bloc **catch** qui le précède.
- ○ **Oui**, car seul le bloc **try** est requis pour spécifier les exceptions, tandis que les blocs **catch** et **finally** sont facultatifs ; on peut les enlever tous les deux sans problème.
- ○ **Non**, car les blocs **try** et **finally** ne peuvent être présents ensemble que si un bloc **catch** est là.
- ○ **Aucune bonne réponse**

Q Oui, c'est correct et très utile car si le code du bloc **try** lève plusieurs exceptions et qu'il n'y a pas de bloc **catch** associé pour intercepter ces exceptions levées, le bloc **finally** s'exécutera toujours peu importe le comportement du bloc **try**.

Tous droits réservés à STUDI - Reproduction interdite

Exercice

Un programme doit-il nécessairement lever toutes les exceptions qui peuvent survenir ?

- ○ **Oui**, si un programme ne lève pas toutes les exceptions possibles, il ne compilera pas.
- ○ **Non**, mais si un programme lève un type d'exception, il doit également lever tous les autres types.
- ◉ **Non**, un programme peut uniquement lever les exceptions que vous souhaitez.
- ○ **Oui**, sinon cela ne compilera pas car la JVM n'aime pas ce genre de pratique.
- ○ **Aucune bonne réponse**

Rien ne vous oblige à gérer toutes les exceptions qu'un code peut lever.

En capturant l'exception de type **Exception** par exemple, vous capturez toutes les exceptions de type **Exception** et ses sous-types.

Exercice

Quelle est la méthode de la classe **Throwable** qui renvoie le contenu du message d'erreur ?

- ○ `getError( )`
- ○ `getException( )`
- ◉ `getMessage( )`
- ○ `traceMessage( )`
- ○ Aucune bonne réponse

C'est la méthode `getMessage( )` de la classe **Throwable** qui renvoie un message détaillé sur l'exception qui s'est produite.

Exercice

Quel(s) sont les types d'exceptions qui ne sont pas vérifiés ?

- ☐ La classe **Throwable** et ses sous-classes
- ☑ La classe **RuntimeException** et ses sous-classes
- ☐ La classe **IOException** et ses sous-classes
- ☑ La classe **Error** et ses sous-classes
- ☐ La classe **ArithmeticException** uniquement

Toutes les exceptions sont vérifiées sauf celles indiquées par **Error (Les erreurs), RuntimeException (erreurs d'exécutions)** et leurs sous-classes.

Exercice

Quelles sont les deux options qu'une méthode doit présenter pour gérer une exception ?

- ○ **Option 1**. Ignorer ces exceptions est généralement la bonne chose à faire.
  Option 2. Vérifier ces exceptions, peut-être avons-nous fait une erreur.
- ○ **Option 1**. Gérer l'exception dans un bloc **catch**.
  Option 2. Renvoyer la gestion de l'exception à la méthode expéditrice.

Tous droits réservés à STUDI - Reproduction interdite

- ◉ **Option 1**. Gérer l'exception dans un bloc **catch**.
  Option 2. Spécifier l'exception à l'aide du mot-clé **throws**.
- ○ **Option 1**. Gérer l'exception dans un bloc **catch**.
  Option 2. Gérer l'exception dans un bloc **try**.
- ○ Aucune bonne réponse

Gérer les exceptions dans un bloc **catch** permet d'exécuter le code que vous voulez lorsque le programme rencontre l'exception captée. La spécifier permet de déléguer sa gestion à une autre méthode située plus haut dans la liste d'appel des méthodes. L'assertion 3 est donc vraie.

Le bloc **try** sert à exécuter un code susceptible de lever une exception, et si une exception est levée, il faudra un bloc **catch**, **finally** ou une levée par une clause **throw** pour la gérer, donc l'assertion 4 est fausse.

Les autres assertions sont fausses également à cause de l'une des deux options toujours proposées.

Exercice

Y a-t-il un problème avec ce gestionnaire d'exceptions tel qu'il est écrit ? Dites si ce code sera compilé.

```
1 try {
2
3 } catch (Exception e) {
4
5 } catch (ArithmeticException a) {
6
7 }
```

- ○ **Oui**, car un bloc **try** peut lever plusieurs exceptions et ces exceptions doivent être gérées par plusieurs blocs **catch**.
- ◉ **Non**, car l'exception de type **Exception** englobe l'exception de type **ArithmeticException** dont elle est la classe parente.
- ○ **Non**, car l'exception de type **ArithmeticException** qui est un sous-type d'**Exception** doit être interceptée et l'exception de type **Exception** ne doit pas être interceptée en plus.
- ○ **Oui**, car on capture distinctement les exceptions. Donc ce code va compiler sans aucun problème.
- ○ **Aucune bonne réponse**

Selon la hiérarchie des exceptions, on a : **Object** > **Throwable** > **Exception** > **RuntimeException** > **ArithmeticException**.

La compilation échoue car **ArithmeticException** a déjà été interceptée. **ArithmeticException** est une sous classe de la classe **Exception**.

Dans ce code on capture l'exception **ArithmeticException** quand elle a déjà été capturée lors de la capture de l'exception de type **Exception** juste avant.

Pour que le code compile, il aurait fallu que l'exception **ArithmeticException** soit capturée avant l'exception-mère de type **Exception**.

Exercice

Quel(s) type(s) d'exception(s) peut engendrer le code du programme de calcul de la masse corporelle ci-dessous ?

Tous droits réservés à STUDI - Reproduction interdite

```java
public class Main {
    public static void main(String[] args) {
        int taille = 12;
        int poids = 0;
        System.out.println("Votre IMC ==> " + taille / poids);
    }
}
```

- [ ] Erreur de compilation : les divisions doivent être dans un bloc **try**
- [x] Erreur d'exécution : **RuntimeException**
- [ ] Erreur de compilation : **DivideByZeroException**
- [x] Erreur d'exécution :**ArithmeticException**
- [ ] Aucune bonne réponse

```
Exception in thread "main" java.lang.ArithmeticException: / by zero
  at exceptions.Main.main(Main.java)
```

On voit bien qu'en exécutant le code on obtient une **ArithmeticException**.

Rappel : selon la hiérarchie des exceptions, on a : **Object** > **Throwable** > **Exception** > **RuntimeException** > **ArithmeticException**.

Donc ce code peut lever une **ArithmeticException** ou une **RuntimeException** ou encore une **Exception**.

Exercice

Quel sera le résultat de ce programme ?

```java
public class Main {

    public static void myMethod() throws Exception {
        System.out.println("ThrowIt");
        throw new Exception();
    }
    public static void main(String[] args) {
        try {
            System.out.println("Hello World!");
            myMethod();
        } catch (Exception e) {
            System.out.println("Caught");
        } finally {
            System.out.println("finally");
        }
        System.out.println("Good bye!");
    }
}
```

- [ ] La compilation échoue, trop de levée d'exceptions.
- [ ] ThrowIt
  Hello World!
  Caught
  finally
  Good bye!

Tous droits réservés à STUDI - Reproduction interdite

- [ ] ThrowIt
  Hello World!
  Caught
  finally
- [x] Hello World!
  ThrowIt
  Caught
  finally
  Good bye!
- [ ] Aucune bonne réponse

La méthode `main( )` capture et gère bien l'exception dans le bloc **catch**, puis poursuit son exécution normalement.

Exercice

Supposons qu'une méthode `A( )` appelle une méthode `B( )` ; et que la méthode `B( )` appelle à son tour une méthode `C( )` qui peut lever une exception de type **Exception**.

Peut-on reporter l'exception de sorte que la méthode `A( )` puisse la gérer ?

```java
public class Main {

    public static void A() {
        B();
    }

    public static void B() {
        C();
    }

    public static void C() {
        throw new Exception("Some exceptions in method C");
    }

    public static void main(String[] args) throws Exception {
        A();
    }
}
```

- [ ] **Non**, la méthode `C( )` doit gérer seule son exception car les exceptions doivent toujours obligatoirement être gérées par les méthodes ou codes susceptibles de les générer.
- [ ] **Non**, on doit obligatoirement gérer les exceptions dans le `main( )`, c'est la règle.
- [x] **Oui**, si les en-têtes des méthodes `B( )` et `C( )` spécifient l'exception dans leur déclaration à l'aide du mot-clé **throws**.
- [ ] **Oui**, si et seulement la méthode `C( )` spécifie l'exception dans sa déclaration à l'aide du mot-clé **throws**.
- [ ] **Aucune bonne réponse**

On peut soit gérer une exception dans la méthode susceptible de la générer, soit la déléguer à une méthode située plus au-dessus dans la pile d'appel de méthodes.

Tous droits réservés à STUDI - Reproduction interdite

```java
public class Main {

    public static void A() {
        try {
            B();
        } catch (Exception e) {
            System.out.println(e.getMessage());
        }
    }

    public static void B() throws Exception {
        C();
    }

    public static void C() throws Exception {
        throw new Exception("Some number exceptions in method C");
    }

    public static void main(String[] args) {
        A();
    }
}
```

```
Some number exceptions in method C
```

p. 29 **Solution n°8**

Le résultat final

Voici donc notre programme ainsi rendu robuste et comprenant les bonnes règles de gestion des exceptions.

```java
import java.util.Scanner;

public class Compte {
    private int solde = 100;

    public static void main(String[] args) throws InterruptedException {
        Compte compte = new Compte();
        System.out.println("""
                Bonjour et Bienvenue dans notre banque
                \tTapez 1 ==> Vérifiez le solde de votre compte
                \tTapez 2 ==> Faire un dépôt
                \tTapez 3 ==> Faire un retrait
                \tTapez 4 pour enregistrer les opérations effectuées et quitter l'application
""");
        try (Scanner sc = new Scanner(System.in)) {

            boolean isWrongAnswer = true;
            while (isWrongAnswer) {

System.out.println("*******************************************************************************");
                System.out.println("\tVeuillez saisir le numéro de la transaction que vous
souhaitez effectuer:");
                switch (sc.nextInt()) {
                    case 1 -> compte.soldeActuel();
                    case 2 -> {
                        System.out.println("Entrez le montant du dépôt");
                        compte.depot(sc.nextInt());
                    }
                    case 3 -> {
```


Tous droits réservés à STUDI - Reproduction interdite

```
27                          System.out.println("Entrez le montant du retrait");
28                          compte.retrait(sc.nextInt());
29                      }
30                      case 4 -> {
31                          System.out.println("Opération(s) enregistrée(s)! Au revoir");
32                          Thread.sleep(2000);
33                          isWrongAnswer = false;
34                      }
35                      default -> System.out.println("Tapez uniquement les nombres entre 1 et
4");
36                  }
37              }
38          } catch (MontantSoldeDoitEtreSuperieurAuMontantRetraitException exception) {
39              System.out.println(exception.getMessage());
40          }
41      }
42
43      /* Le solde doit toujours être positif */
44      public int getSolde() {
45          return solde;
46      }
47
48      /* Le montant à retirer doit toujours être > le solde actuel */
49      public void retrait(int montant) throws
MontantSoldeDoitEtreSuperieurAuMontantRetraitException {
50          if (montant > solde) {
51              throw new MontantSoldeDoitEtreSuperieurAuMontantRetraitException (
52                      "Le solde actuel de " + this.getSolde() + " euros est inférieur par
rapport au montant de " +
53                              montant + " euros que vous souhaitez retirer");
54          }
55          this.solde -= montant;
56          soldeActuel();
57      }
58
59      public void depot(int montant) {
60          this.solde += montant;
61          soldeActuel();
62      }
63
64      public void soldeActuel() {
65          System.out.println("Le solde du compte courant est de " + this.getSolde() + "
euros");
66      }
67
68  }
69
70  class MontantSoldeDoitEtreSuperieurAuMontantRetraitException extends Exception {
71      private final String message;
72
73      public MontantSoldeDoitEtreSuperieurAuMontantRetraitException(String message) {
74          this.message = message;
75      }
76
77      public String getMessage() {
78          return this.message;
79      }
80  }
```

Tous droits réservés à STUDI - Reproduction interdite

```
C:\>javac Compte.java
Bonjour et Bienvenue dans notre banque
  Tapez 1 ==> Vérifiez le solde de votre compte
  Tapez 2 ==> Faire un dépôt
  Tapez 3 ==> Faire un retrait
  Tapez 4 pour enregistrer les opérations effectuées et quitter l'application
*******************************************************************************
  Veuillez saisir le numéro de la transaction que vous souhaitez effectuer:
1
Le solde du compte courant est de 100 euros
*******************************************************************************
  Veuillez saisir le numéro de la transaction que vous souhaitez effectuer:
2
Entrez le montant du dépôt
150
Le solde du compte courant est de 250 euros
*******************************************************************************
  Veuillez saisir le numéro de la transaction que vous souhaitez effectuer:
3
Entrez le montant du retrait
200
Le solde du compte courant est de 50 euros
*******************************************************************************
  Veuillez saisir le numéro de la transaction que vous souhaitez effectuer:
3
Entrez le montant du retrait
100
Le solde actuel de 50 euros est moindre par rapport au montant de 100 euros que vous voulez retirer

Process finished with exit code 0
```


Tous droits réservés à STUDI - Reproduction interdite