Gestion des exceptions en Java

Alla LO

Enseignant Chercheur en IA & BI
Université Alioune Diop de Bambey
13 octobre 2025

Objectifs spécifiques

À la fin de ce chapitre, l'étudiant sera capable de :

- Comprendre la définition et la gestion des exceptions en Java.
- Utiliser les blocs `try-catch-finally` pour gérer les exceptions.
- Créer et lancer ses propres exceptions.
- Différencier les exceptions de type `Error` et `RuntimeException`.
- Comprendre les avantages de l'utilisation des exceptions.
- Appliquer les meilleures pratiques pour la gestion des exceptions.

Plan du cours

1. Qu'est-ce qu'une situation exceptionnelle ?
2. La classe Error
3. La classe Exception
   - Traitement des exceptions
   - Créer son propre type d'exception
   - La clause finally
4. La classe RuntimeException
5. Quelques bonnes pratiques
6. Exemple pratique détaillé
7. Conclusion

Introduction

De quoi parle-t'on ?

Bien souvent, un programme doit traiter des situations exceptionnelles qui n'ont pas un rapport direct avec sa tâche principale. Ceci oblige le programmeur à réaliser de nombreux tests avant d'écrire les instructions utiles du programme. Cette situation a deux inconvénients majeurs :

- le programmeur peut omettre de tester une condition ;
- le code devient vite illisible, car la partie utile est masquée par les tests.

Java remédie à cela en introduisant un Mécanisme de gestion des exceptions qui est l'objet de ce cours. Grâce à ce mécanisme, on peut améliorer grandement la lisibilité du code en découplant le code utile de celui qui traite des situations exceptionnelles, et on peut aussi déléguer au langage la tâche d'énumération des tests à effectuer.

Qu'est-ce qu'une exception en Java ?

Definition

Une situation exceptionnelle peut être assimilée à une erreur (dans le cadre de ce cours), c'est-à-dire une situation qui est externe à la tâche principale d'un programme. En Java, on distingue trois types d'erreurs, qui sont de degrés de gravité différents, à savoir :

- les erreurs graves qui causent généralement l'arrêt du programme et qui sont représentées par la classe java.lang.Error ;
- les erreurs qui doivent généralement être traitées et qui sont représentées par la classe java.lang.Exception ;
- les erreurs qui peuvent ne pas être traitées et qui sont des objets de la classe java.lang.RuntimeException qui hérite de java.lang.Exception.

Toutes ces classes héritent directement ou indirectement de la classe java.lang.Throwable. Voici un petit diagramme récapitulatif de tout cela :

Qu'est-ce qu'une situation exceptionnelle ?

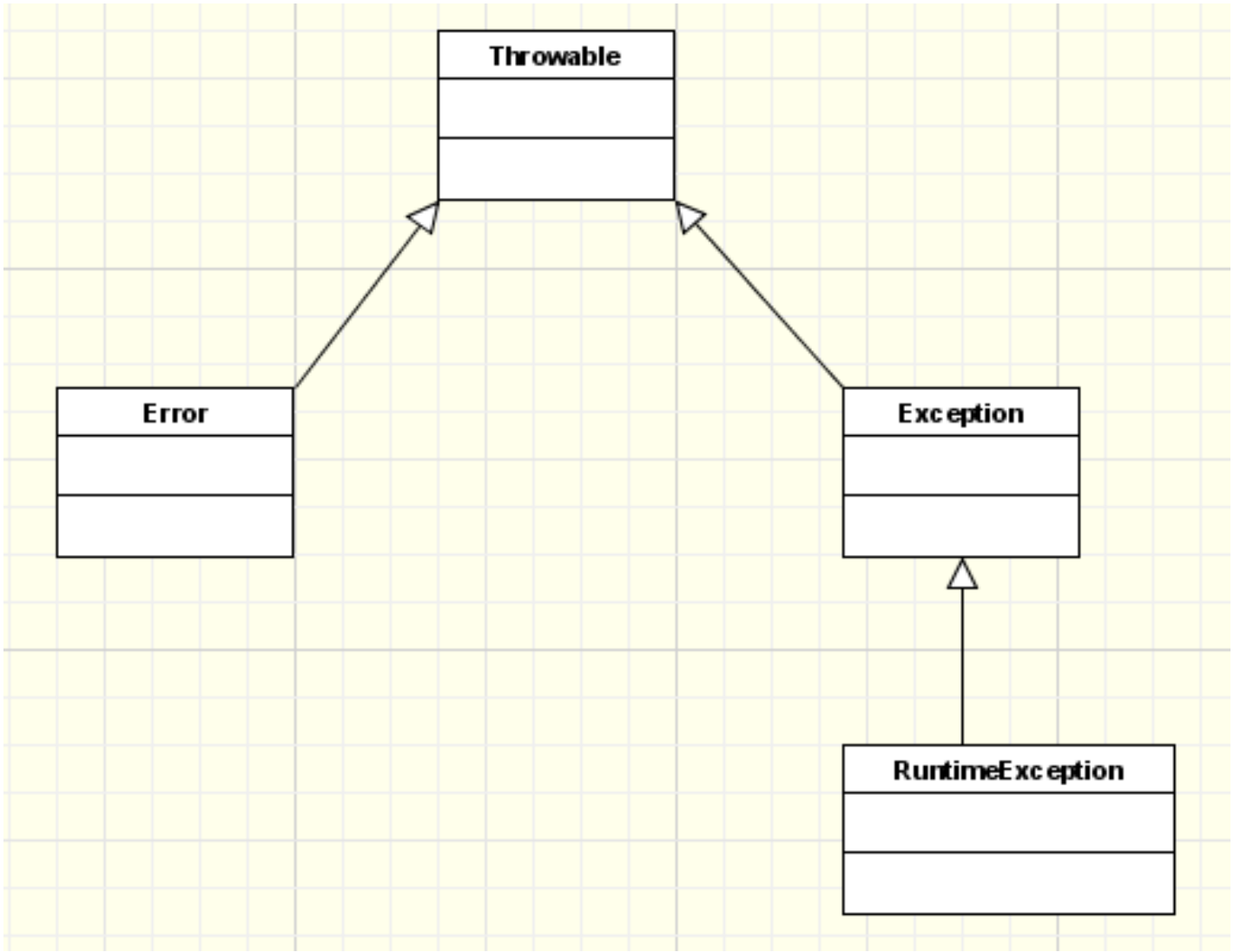

Figure – 1. Diagramme de classe

La classe Error

Cette classe est instanciée lorsqu'une erreur grave survient, c'est-à-dire une erreur empêchant la JVM de faire correctement son travail. Les objets de type Error ne sont pas destinés à être traités et il est même déconseillé de le faire. Un exemple récurrent est java.lang.OutOfMemoryError qui signifie que la machine virtuelle Java ne dispose plus d'assez de mémoire pour pouvoir allouer des objets. À titre d'exemple, voici un code qui alloue de la mémoire pour un tableau de 1 000 000 000 String et par ce fait déclenche une OutOfMemoryError :

Exemple

```
1 public class ErreurMemoire {
2     public static void main(String[] args) {
3         String[] tableau=new String[1000000000];
4     }
5 }
```

La classe Error

Les erreurs, lorsqu'elles surviennent, ont la particularité d'arrêter le thread en cours, sauf si elles sont traitées par un catch. Ainsi, n'importe quel type de Throwable peut être « catché ». Le code précédent deviendrait alors :

Exemple

```
1 try {
2 String[] tableau=new String[1000000000]; // OutOfMemoryE
3 } catch (Error e) {
4   System.out.println("Oups! Une erreur est survenue: "+e
5 }
6 System.out.println("Fin du programme");
```

Note :

Mais comme mentionné plus haut, cette pratique est à éviter, car les Errors correspondent à des problèmes graves qu'il n'est généralement pas possible de traiter dans le code.

La classe Exception

Les objets de type Exception ou bien de l'une de ses sous-classes sont instanciés lorsqu'une erreur au niveau applicatif survient. On dit, dans ce cas-là, qu'une exception est levée. Lorsqu'une exception est levée, elle se propage dans le code en ignorant les instructions qui suivent, et si aucun traitement ne survient, elle débouche sur la sortie standard. Voici un bout de code illustrant cela :

Code

```
1 public class PropagationException {
2     public static void main(String[] args) {
3         String chemin="/Un/chemin/vers/une/
4                       classe/qui/n'existe/pas";
5         Class.forName(chemin);//levee d'une
6                             ClassNotFoundException
7         System.out.println("fin du programme");
8     }
9 }
```

La classe Exception

On voit bien sur cet exemple que l'instruction qui suit la levée de l'exception n'est pas exécutée : on n'obtient pas l'affichage fin du programme.

Traitement des exceptions

Try/Catch

Les exceptions sont traitées via des blocs try/catch qui veulent littéralement dire essayer/attraper. On exécute les instructions susceptibles de lever une exception dans le bloc try et en cas d'erreur ce sont les instructions du bloc catch qui seront exécutées, pourvu qu'on attrape bien l'exception levée. Reprenons notre exemple de tout à l'heure et traitons l'exception. Ce qui donne le code suivant :

Traitement des exceptions

Code

```
1  public class PropagationException {
2      public static void main(String[] args) {
3          try{
4              String chemin="/Un/chemin/vers/une/classe/
5                                  qui/n'existe/pas";
6              Class.forName(chemin);//lev e d'une
7                                  ClassNotFoundException
8              System.out.println("fin du programme");
9          }catch(ClassNotFoundException ex){
10             System.out.println("Une exception
11                                 est survenue");
12         }
13     }
14 }
```

Traitement des exceptions

Comme prévu, on obtient bien l'affichage :« Une exception est survenue » . Il faut tout de même faire attention au type d'exception qu'on met dans le catch : on aurait pu simplement déclarer une exception de type Exception. Cela aurait pour effet d'attraper toutes les exceptions levées dans le bloc try, car l'ensemble des exceptions déclarées dans la JDK hérite de cette classe (Exception)., il va sans dire que la réciproque n'est pas vraie. On peut également mettre plusieurs blocs catch qui se suivent afin de fournir un traitement spécifique pour chaque type d'exception. Cela doit être fait en respectant la hiérarchie des exceptions. Un code comme celui-ci ne compilera pas !

Code

```
1  public class PropagationException {
2
3      public static void main(String[] args) {
4          try{
5              String chemin="/Un/chemin/vers/une/classe/
6                      qui/n'existe/pas";
7              Class.forName(chemin);//lev e d'une
8                  ClassNotFoundException
9              System.out.println("fin du programme");
10         }catch(Exception e){
11             //traitement
12             //erreur de compilation, car les
13             //autres blocs catch ne seront
14             //jamais ex cut s
15         }catch(ClassNotFoundException ex){
16             System.out.println("Une exception est
17              survenue");
```

Il faudrait plutôt écrire ceci :

code

```
1  public class PropagationException {
2      public static void main(String[] args) {
3          try{
4              String chemin="/Un/chemin/vers/une/classe/
5                  qui/n'existe/pas";
6              Class.forName(chemin);//lev e d'une
7                      ClassNotFoundException
8              System.out.println("fin du programme");
9          }catch(ClassNotFoundException ex){
10             System.out.println("Une exception
11                     est survenue");
12         }catch(Exception e){
13             //traitement pas d'erreur de compilation
14         }
15     }
16 }
```

Créer son propre type d'exception

Pour créer son propre type d'exception, il faut écrire une classe héritant de la classe Exception. Allons-y donc, créons une exception qu'on appellera NombreNonValideException qu'on lèvera si l'utilisateur de notre programme entre un nombre non compris entre 0 et 9. Voici à quoi ressemble notre classe NombreNonValideException :

Code

```
1 public class NombreNonValideException extends Exception{
2
3     /** Cr e une nouvelle instance de
  NombreNonValide */
4     public NombreNonValide() {}
5
6 }
```

Créer son propre type d'exception (suite)

On pourrait se contenter du code précédent, cependant il est souvent préférable d'utiliser les mêmes constructeurs que la classe Exception, afin de simplifier leurs créations et l'encapsulation d'exception. Voici à quoi ça correspondrait :

Créer son propre type d'exception (suite)

Code

```
1  public class NombreNonValideException extends Exception{
2      /**
3      * Cr e une nouvelle instance de
4      * NombreNonValideException
5      */
6      public NombreNonValideException() {}
7      /**
8      * Cr e une nouvelle instance de
9      * NombreNonValideException
10     * @param message Le message d taillant exception
11     */
12     public NombreNonValideException(String message) {
13         super(message);
14     }
```

Créer son propre type d'exception (suite)

Code suite

```
1     """ Cr e une nouvelle instance de
NombreNonValideException  @param
2     cause L'exception    l'origine
de cette exception """
3     public NombreNonValideException(Throwable cause) {
4         super(cause);
5     }
6     """ Cr e une nouvelle instance de NombreNonValideEx
@param cause L'exception    l'origine de cette exception
7     """
8     public NombreNonValideException(String message,
9         Throwable cause) {
10        super(message, cause);
11    }
12 }
```

Créer son propre type d'exception (suite)

Et voici notre programme :

```
1  public class Nombre {
2      public void parseAndPrint(String number) throws
3      NombreNonValideException {
4          try {
5              int i = Integer.parseInt(number);
6              if (i < 0 || i > 9) {
7                  throw new NombreNonValideException("bad
8                                          value");
9              }
10             System.out.println(i);
11         } catch (NumberFormatException e) {
12             // encapsulation de l'exception
13             throw new NombreNonValideException("parse
14                                     error", e);
15         }
16     }
17 }
```

Créer son propre type d'exception (suite)

Note :

Les NumberFormatException sont encapsulés dans une NombreNonValideException.

Il y a ici deux choses à remarquer. Tout d'abord la présence de la clause throws dans la signature de la méthode, celle-ci est obligatoire pour toute méthode qui peut lever une exception. Ensuite, on voit que pour lever une exception il faut user du mot clé throw suivi du type de l'exception qu'on instancie.

La clause finally

Le mot clé finally, généralement associé à un try, permet l'exécution du code situé dans son bloc et ceci quelle que soit la manière dont s'est déroulé l'exécution du bloc try. Voici sans plus attendre un exemple :

Code

```
1  public class PropagationException {
2
3      public static void main(String[] args) {\\
4          try{
5              Object chaine="bonjour";
6              Integer i=(Integer)chaine;//lev e d'une Cla
7              System.out.println("fin du programme");
8          }finally{
9              System.out.println("on passe par le bloc fin
10
11         }
12     }
```

La classe RuntimeException

Les exceptions héritant de java.lang.RuntimeException représentent des erreurs qui peuvent survenir lors de l'exécution du programme. Le compilateur n'oblige pas le programmeur ni à les traiter ni à les déclarer dans une clause throws. Les classes java.lang.ArithmeticException (qui peuvent survenir lors d'une division par 0 par exemple), et la classe ***java.lang.ArrayIndexOutOfBoundException*** (qui survient lors d'un dépassement d'indice dans un tableau) sont des exemples de ***RuntimeException***. Autrement dit, ce genre de code passe sans problème la compilation :

La classe RuntimeException

Code

```
1 public class CompilationRuntimeException {
2     public static void main(String[] args) {
3         String[] tableau={"A","B","C"};
4         for(int i=0;i<=3;i++){
5             System.out.println(tableau[i]);
6         }
7     }
8 }
```

Mais à l'exécution, on obtient bien une ArrayIndexOfBoundsException sans l'avoir préalablement déclarée dans une clause throws.

Ne jamais ignorer une exception

Une erreur fréquente du débutant est de mettre un bloc catch vide sans aucune instruction afin de pouvoir compiler le programme. Ceci est très dangereux, car cela risque de devenir une mauvaise habitude. En effet, si une exception survient, elle sera passée sous silence et le programme continuera de fonctionner ce qui peut déboucher sur des bogues incompréhensibles. Ayons donc le réflexe de bien traiter les exceptions dans les blocs catch ou au moins de mettre un printStackTrace, ça ne mange pas de pain ! .

Ne jamais ignorer une exception

Code

```
1 public class NePasIgnorerUneException {
2     public static void main(String[] args) {
3         try{
4         //traitement susceptible de lever une exception
5         }catch(Exception ex){
6             ex.printStackTrace();
7         }
8     }
9 }
```

Utiliser la clause throws de manière exhaustive

Supposons que nous ayons une exception C qui hérite d'une exception B qui elle-même hérite d'une autre exception A, alors une méthode f() qui peut lancer ces trois exceptions doit le déclarer dans sa signature via la clause throws. Une mauvaise manière de déclarer f() est la suivante :

Code

```
1 public void f() throws A{
2 //corps de la m thode
3 }
```

Utiliser la clause throws de manière exhaustive

Bien que cette façon de faire passe sans problème l'étape de compilation, elle est déconseillée, car elle ne fournit pas toutes les exceptions qu'elle peut lever à l'utilisateur de cette méthode. La bonne manière de coder cela est la suivante :

Code

```
1 public void f() throws A, B, C {
2 //corps de la m thode
3 }
```

En effet, dans ce cas-là, l'utilisateur de f() est en mesure de connaître toutes les exceptions susceptibles d'être levées par f et est à même de fournir une gestion fine de celles-ci.

Les exceptions ne sont pas faites pour le contrôle de flux

C'est une très mauvaise idée que d'utiliser les exceptions pour contrôler le flux. Considérons à titre d'exemple le code suivant :

Code

```
1 while(true){//faire quelque chose
2 if(condition d'arr t)
3     throw new FinDeBoucleException();
4 }
```

Ici, on se sert de la levée d'une exception pour sortir de la boucle. Ce genre de code, bien qu'il fonctionne, a plusieurs inconvénients : il n'est pas efficace (création d'un objet supplémentaire, à savoir l'exception), il est difficilement compréhensible et modifiable. Le langage Java fournit suffisamment d'instructions de contrôle pour éviter totalement ce genre de code, par exemple on peut utiliser l'instruction break.

Les exceptions et les entrées/sorties

Concernant les entrées/sorties en Java, pensez à utiliser le pattern suivant :

Code

```
1  try{
2       //d claration de la ressource
3       try{
4            //utilisation de la ressource
5       }finally{
6            //fermeture de la ressource
7       }
8  }catch(ExceptionEntreeSortie ex){
9       //traitement de l'exception
10 }
```

Les exceptions et les entrées/sorties

Voici sans plus attendre une mise en œuvre de ce pattern avec un programme qui affiche toutes les lignes d'un fichier texte en majuscules :

Code

```
1  import java.io.BufferedReader;
2  import java.io.FileReader;
3  import java.io.IOException;
4  public class FluxMajuscule {
5      /** Cr e une nouvelle instance de
   FluxMajuscule */
6      public FluxMajuscule() {
7      }
8
9      public void readMaj(){
10         try{
```

Les exceptions et les entrées/sorties (suite)

Code

```
1               BufferedReader br=new BufferedReader(new
2                   FileReader("monFichier.txt"));
3               try{
4                   String ligne;
5                   String ligneMajuscule;
6                   while((ligne=br.readLine())!=null){
7                       ligneMajuscule=ligne.toUpperCase();
8                       System.out.println(ligneMajuscule);
9                   }
10              }finally{
11                  br.close();
12              }
13          }catch(IOException ex){
14              ex.printStackTrace();
15          }
16      }
17  }
```

Attention au return dans un bloc finally !

On sort d'un bloc try lorsque l'une des situations suivantes se présente :

- le bloc try se termine normalement ;
- une exception survient ;
- une instruction de rupture de séquence telle que break, continue ou return est utilisée.

Maintenant, examinons le code suivant :

```
1 public class ReturnFinally {
2
3     public int methode1(){
4         try{
5             return 1;
6         }catch(Exception e){
7             return 2;
8         }
9     }
```

Attention au return dans un bloc finally !

Code

```
1       public int methode2(){
2           try{
3               return 3;
4           }finally{
5               return 4;
6           }
7       }
8       public static void main(String[] args) {
9           ReturnFinally rf=new ReturnFinally();
10          System.out.println("methode1 renvoie : "+
11                  rf.methode1());
12          System.out.println("methode2 renvoie : "+
13                  rf.methode2());
14      }
15  }
```

Attention au return dans un bloc finally ! (suite)

On obtient l'affichage suivant :

```
1          methode1 renvoie : 1
2          methode2 renvoie : 4
```

Bien que le premier résultat soit prévisible, le deuxième l'est beaucoup moins. En effet, on aurait tendance à penser qu'employer une instruction de rupture de séquence telle que return permet de quitter la méthode, ceci est vrai sauf si une clause finally existe.
Conclusion : évitez d'employer des instructions de rutpture de séquence telle que break, continue ou return à l'intérieur d'un bloc try. Si c'est inévitable, assurez-vous qu'aucune clause finally ne modifie la valeur de retour de votre méthode.

Utiliser les exceptions standards

Bien qu'il est aisé de créer son propre type d'exception, l'API Java en fournit suffisamment en standard pour vous éviter cette tâche

Voici quelques exceptions courantes :

- `IOException` : Erreur d'entrée/sortie.
- `FileNotFoundException` : Fichier introuvable.
- `SQLException` : Erreur lors de l'exécution d'une requête SQL.
- `NullPointerException` : Tentative d'utilisation d'un objet nul.

Exemple avec FileNotFoundException

Exemple :

```
1 try {
2     FileInputStream file = new FileInputStream("fichi-
3     er.txt");
4 } catch (FileNotFoundException e) {
5     System.out.println("Fichier non trouv .");
6 } catch (IOException e) {
7     System.out.println("Erreur d'entr e/sortie.");
8 }
```

Une exception peut en cacher une autre !

L'exception qui apparaît sur la sortie standard n'est pas forcément celle qui est à l'origine de l'erreur. En effet, en exécutant ce code :

Code

```
1 public class TestException {
2  public static void main(String[] args)throws Exception{
3    try{
4       throw new Exception("1");
5     }catch(Exception ex){
6       throw new Exception("2");
7     }
8   }
9 }
```

On obtient la sortie suivante :
Exception in thread "main" java.lang.Exception : 2

Une exception peut en cacher une autre ! (suite)

On voit bien que c'est la deuxième exception qui est renvoyée (celle qui se trouve dans le bloc catch) alors que c'est la première exception (celle qui se trouve dans le bloc try) qui est à l'origine de l'erreur.
Ceci nous amène à parler de l'encapsulation des exceptions qui consiste à regrouper plusieurs exceptions en une seule sans pour autant perdre l'information utile en cas d'erreur. L'encapsulation est réalisée généralement grâce au constructeur de la classe Exception qui prend en paramètre un Throwable.
Mais comme un bout de code vaut mieux qu'un long discours, voici tout de suite un exemple :

Une exception peut en cacher une autre ! (suite)

Code

```
1 try {
2       maMethodeQuiRenvoitPlusieursTypesDException();
3  }catch (Exception e) {
4     // En englobe toutes les exceptions dans une exceptio
5     throw new Exception("Un probl me est survenu", e);
6  }
```

Ce qui donne sur la sortie standard :

Une exception peut en cacher une autre !(suite)

```
1 Exception in thread "main" java.lang.Exception: Un probl
2     at Main.main(Main.java:72)
3 Caused by: java.io.IOException: IO error
4     at Main.maM thodeQuiRenvoitPlusieursTypesDException
5     at Main.main(Main.java:69)
```

Conclusion

En englobant des exceptions dans une autre, on peut simplifier la gestion des exceptions sans pour autant perdre l'information qui leur est associée.

Bien déterminer la totalité du traitement qui sera interrompu lorsqu'une exception est levée

Un exemple d'erreur typique, avec ce code qui utilise la « reflection » pour instancier un objet :

Code

```
1 Class type = null;
2 Object object = null;
3 try {
4         type = Class.forName("monpackage.MaClasse"); //
5 } catch (ClassNotFoundException e) {
6     e.printStackTrace();
7     // + traitement particulier     ClassNotFoundExceptio
8 }
```

Bien déterminer la totalité du traitement qui sera interrompu lorsqu'une exception est levée

Code

```
1  try {
2      object = type.newInstance(); // throws Instantia-
3      //tionException, IllegalAccessException
4  } catch (InstantiationException e) {
5      e.printStackTrace();
6      // + traitement particulier     Instantia-
7      //tionException
8  } catch (IllegalAccessException e) {
9      e.printStackTrace();
10     // + traitement particulier
11     //IllegalAccessException
12 }
13 String string = object.toString();
```

Bien déterminer la totalité du traitement qui sera interrompu lorsqu'une exception est levée

Le code peut renvoyer trois types d'exceptions selon les méthodes, qui sont bien englobées dans des ***try/catch***, et le code marche correctement lorsqu'aucune exception n'est générée... Pourtant il n'est pas du tout sécurisé, car lorsqu'une exception survient, elle n'interrompt qu'une partie du traitement : par exemple, si la méthode ***Class.forName()*** remonte une exception, l'objet type restera toujours à null, mais on tentera quand même d'appeler la méthode newInstance() dessus, ce qui provoquera une ***NullPointerException***...
La solution consiste à ce que les blocs ***try/catch*** doivent englober la totalité du traitement à interrompre en cas de problème. Dans ce cas, il est donc préférable d'utiliser le code suivant :

Bien déterminer la totalité du traitement qui sera interrompu lorsqu'une exception est levée (suite)

Code

```
1  try {
2      Class type = Class.forName("monpackage.MaClasse"); /
3      Object object = type.newInstance(); // throws Instan
4      //tionException, IllegalAccessException
5      String string = object.toString();
6  } catch (ClassNotFoundException e) {
7      e.printStackTrace();
8      // + traitement
9  } catch (InstantiationException e) {
10     e.printStackTrace();
11     // + traitement
12 } catch (IllegalAccessException e) {
13     e.printStackTrace();
14     // + traitement
15 }
```

Bien déterminer la totalité du traitement qui sera interrompu lorsqu'une exception est levée (suite)

De plus, ce code a le mérite d'être bien plus lisible :

- Tout le code utile est regroupé à l'intérieur du try.
- Tous les catch sont au même niveau, ce qui pourrait permettre d'utiliser un traitement commun

Exemple pratique détaillé

Retrait invalide

InvalidWithdrawalException

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

Exemple pratique détaillé

BankAccount

```
1  class BankAccount {
2      private double balance;
3
4      public BankAccount(double balance) {
5          this.balance = balance;
6      }
7
8      public void withdraw(double amount) throws
9      InvalidWithdrawalException {
10         if (amount > balance) {
11             throw new InvalidWithdrawalException("Mon-
12             tant insuffisant !");
13         }
14         balance -= amount;
15     }
16 }
```

Exemple pratique détaillé

Main

```
1  public class Main {
2      public static void main(String[] args) {
3          BankAccount account = new BankAccount(1000);
4          try {
5              account.withdraw(1500);
6          } catch (InvalidWithdrawalException e) {
7              System.out.println("Erreur : " +
8              e.getMessage());
9          }
10     }
11 }
```

Conclusion

Dans ce chapitre, nous avons étudié la gestion des exceptions en Java, notamment leur définition, l'utilisation des blocs try-catch-finally, la création d'exceptions personnalisées, ainsi que les différences entre exceptions vérifiées et non vérifiées. Nous avons également vu comment propager et capturer des exceptions, utiliser le multi-catch, et appliquer de bonnes pratiques pour garantir une gestion efficace des erreurs.
En atteignant ces objectifs, vous êtes désormais capable de gérer des erreurs de manière proactive dans vos applications Java, en améliorant leur robustesse et leur lisibilité, tout en assurant la bonne gestion des ressources et des flux critiques, comme les fichiers et les connexions réseau.