Chapitre 11

Les exceptions

1. Intérêt et définition
2. Hiérarchie organisationnelle des exceptions
3. Le traitement des exceptions
4. Le propagation et la levée d’exception
5. Les exceptions personnalisées

1 Intérêt et définition

2 Hiérarchie organisationnelle des exceptions

3 Le traitement des exceptions

4 Le propagation et la levée d’exception

5 Les exceptions personnalisées

1 Intérêt et définition

- Une exception est un signal qui indique qu'un événement anormal est survenu dans un programme

  Exemples :

  - Ajout d'un élément dans une collection à une position invalide
  - Division par 0 dans un calcul
  - Utilisation d'une variable objet non instancié
  - Erreur de casting de type de données entre deux objets
  - Défaillance de l'environnement d'exécution
  - etc..

- La notion d'exception est offerte aux programmeurs Java pour leur permettre de résoudre de manière simple et efficace le problème de la gestion des erreurs qui surviennent lors de l'exécution d'un programme
  - La gestion des exceptions entraine la **robustesse** du programme et donc sa capacité à être **résilient**

- La robustesse d'une application est souvent comprise comme sa capacité à continuer à rendre un service acceptable dans un environnement dégradé, c'est-à-dire, quand toutes les conditions normalement attendues ne sont pas satisfaites

1 Intérêt et définition

2 Hiérarchie organisationnelle des exceptions

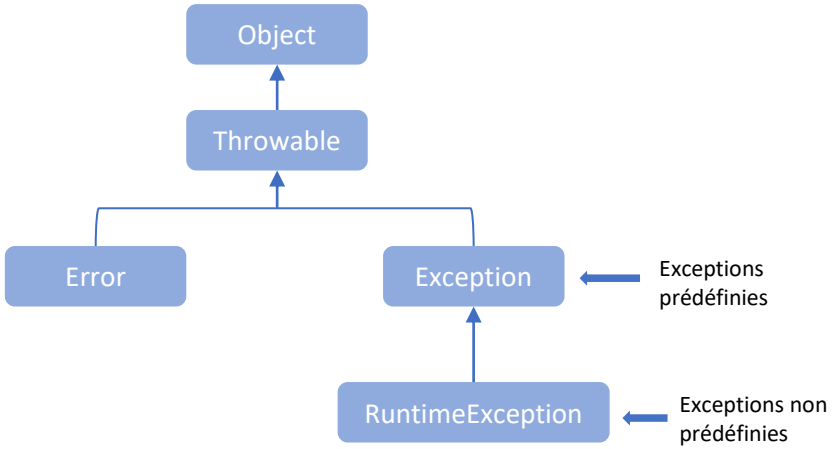
3 Le traitement des exceptions

4 Le propagation et la levée d’exception

5 Les exceptions personnalisées

2 Hiérarchie organisationnelle des exceptions (1/3)

- Java organise la hiérarchie (en terme d'héritage) des exceptions comme ceci :



La classe mère de toutes les erreurs qui peuvent survenir lors de l'exécution d'un programme Java est Throwable, elle-même héritant de la classe de base Object

La classe Error correspond aux erreurs fatales qui relèvent des problèmes systèmes. Ce genre d'erreur est irrécupérable et donc inutile de tenter de la traiter. Exemple, si l'OS tombe en panne pendant l'exécution du programme, ce dernier n'y peut rien

Les erreurs à gérer dans un programme concerne celles des classes Exception et RuntimeException. Elles sont résiduelles aux instructions écrites dans le programme

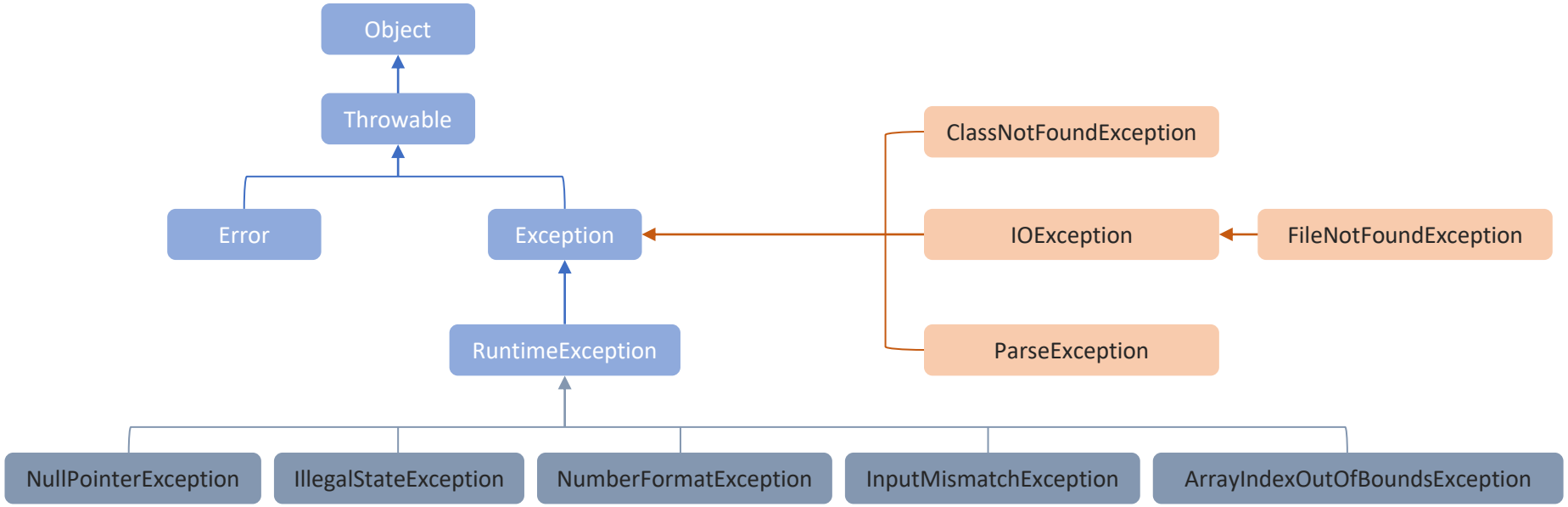
- Il existe deux catégories d'exceptions à gérer dans un programme :
  - Les exceptions qui sont **sous-classes d'Exception (mais différentes de RuntimeException)** appelées **exceptions prédéfinies**
  - Les exceptions qui sont **sous-classes de RuntimeException** appelées **exceptions non prédéfinies**

2 Hiérarchie organisationnelle des exceptions (2/3)

- Une **exception prédéfinie** (ou **checked exception** en anglais) dans un programme en est une
  - dont la survenue est prévisible d'avance lorsqu'on manipule une fonction donnée issue d'une bibliothèque
  - et dont lors de la phase de compilation, le compilateur *javac* vérifie que l'on a pris en compte son traitement. Exemples :
    - **ClassNotFoundException** => survient si l'on tente de charger un objet dont la JVM ne retrouve pas la classe
    - **IOException** => survient si d'une erreur dans la manipulation d'objets d'entrée/sortie
    - **FileNotFoundException** => survient si la tentative d'ouverture d'un fichier échoue
    - **ParseException** => survient si le parsing d'une donnée échoue (ex: parsing d'une donnée lue un objet Scanner)

- Une **exception non prédéfinie** (ou **unchecked exception** en anglais) dans un programme en est une
  - dont la survenue n'est pas prévisible à priori
  - et est donc due à une erreur de logique dans l'écriture des instructions d'un programme. Exemples :
    - **NullPointerException** => Survient si l'on manipule un objet qui vaut null et donc non instancié
    - **IllegalStateException** => survient pour indiquer qu'une méthode a été appelée à un endroit inapproprié
    - **ArrayIndexOutOfBoundsException** => survient lors de l'ajout/suppression d'un élément à une position invalide d'une collection
    - **NumberFormatException** => survient pour indiquer une tentative de mauvaise conversion d'une String en un numérique
    - **InputMismatchException** => survient pour indiquer qu'une méthode a été appelée avec des mauvais paramètres

2 Hiérarchie organisationnelle des exceptions (3/3)



Il existe de nombreuses autres classes d'exception natives de Java et ce schéma n'est qu'une esquisse sur quelques unes

1 Intérêt et définition

2 Hiérarchie organisationnelle des exceptions

3 Le traitement des exceptions

4 Le propagation et la levée d’exception

5 Les exceptions personnalisées

3 Le traitement des exceptions (1/3)

- Le traitement d'une ou de plusieurs exceptions obéit à la gestion de code dans les blocs suivants :

```
try {

    //bloc de code concerné par les erreurs qui peuvent survenir

} catch(ExceptionATraiter_1 | ExceptionATraiter_2 | … | ExceptionATraiter_n exc) {

    //bloc de code qui traite les différentes exceptions

} finally {

    //bloc de code qui doit obligatoirement être exécuté après que celui dans le try
    //soit terminé (en erreur ou non)

}
```

Les blocs try et catch vont de paire. Mais seul le bloc try est obligatoire. Le bloc catch est facultatif, mais il est rare de pas le définir

La variable exc est de nomenclature libre et l'on est aussi libre de déclarer 1 ou plusieurs exceptions à traiter

Lorsqu'on ne veut pas entrer dans le détail précis des exceptions à traiter, on utilise la mère des exceptions, la classe Exception

Le bloc finally est facultatif

- Si plusieurs classes d'exceptions sont passées dans le bloc *catch*, il faut s'assurer qu'une sous classe ne soit déclarée après sa classe mère. Sinon, il y aura une erreur de compilation
  Exemple : si ExceptionATraiter_2 est sous-classe de ExceptionATraiter_1, alors la définition ci-dessus n'est pas bonne

3 Le traitement des exceptions (2/3)

- Exemple 1 : ce programme lit les données du fichier file1.txt et les recopie dans le fichier file2.txt et traite les exceptions si jamais ça survenait

```
import java.io.BufferedReader;
import java.io.BufferedWriter;
import java.io.FileReader;
import java.io.FileWriter;
import java.io.IOException;

public class ExempleLectureEcritureFichier {

   public static void main(String[] args) throws IOException {
     FileReader fr = null; FileWriter fw = null;
     BufferedReader bf = null; BufferedWriter bw = null;
     try{
        fr = new FileReader("C:/tmp/bidule/file1.txt");
        fw = new FileWriter("C:/tmp/bidule/file2.txt");

        bf = new BufferedReader(fr);
        bw = new BufferedWriter(fw);

        String ligne = bf.readLine();
        while(ligne != null) {
            bw.write(ligne);
            bw.newLine();
            ligne = bf.readLine();
        }
        bw.flush();
     }catch(IOException exc) {

        System.out.println("La cause de l'erreur est : " + exc.getMessage());
     }finally {
        //fermeture propres des flux
        fr.close();
        fw.close();
        bf.close();
        bw.close();
     }
   }
}
```

Remarques :

- Le bloc de code qui effectue le traitement des fichiers est balisé par le bloc try { .. }

- Le bloc catch { .. } traite l'exception IOException qui pourrait survenir suite à une impossibilité de lire l'un ou les deux fichiers des fichiers. Et dans ce bloc, on imprime le message que la JVM renverra dans l'objet exec au travers de sa méthode getMessage()

- Finalement dans le bloc finally { ..}, puisque des flux de données ont été ouverts dans le bloc try { .. }, ils doivent être refermés impérativement qu'une erreur se soit produite ou non. On ajoute le bloc finally uniquement si l'on veut réaliser impérativement un traitement après le bloc try..catch.

3 Le traitement des exceptions (3/3)

- Exemple 2 :

```
import java.io.*;

public class Exemple3LectureEcritureFichier {

   public static void main(String[] args) throws IOException {
     FileReader fr = null; FileWriter fw = null;
     BufferedReader bf = null; BufferedWriter bw = null;
     try{
        fr = new FileReader("C:/tmp/bidule/file1.txt");
        fw = new FileWriter("C:/tmp/bidule/file2.txt");

        bf = new BufferedReader(fr);
        bw = new BufferedWriter(fw);

        String ligne = bf.readLine();
        ligne = ligne.trim();
        while(ligne != null) {
            bw.write(ligne);
            bw.newLine();
            ligne = bf.readLine();
        }
        bw.flush();
     }catch(IOException | RuntimeException ex) {
        if(ex instanceof IOException ) {
          System.out.println("Erreur issue de IOException : " + ex.getMessage());
        }else {
          System.out.println("Erreur issue de RuntimeException : " + ex.getMessage());
        }
     }finally {
        //fermeture propres des flux
        fr.close();
        fw.close();
        bf.close();
        bw.close();
     }
   }
}
```

Remarques :

Il s'agit du même programme précédent, à la différence que dans le bloc try {…}, lorsqu'on lit une ligne dans le premier fichier, on l'y supprime les espaces avant et après la ligne avant de l'écrire dans le 2ème fichier (cf. *ligne.trim()*)

Du coup, cette opération a un risque : c'est que s'il arrivait qu'une ligne nulle soit lue du primière fichier, la tentative d'appliquer la méthode trim(), sur un objet null lèvera une exception, qui cette fois n'est pas prévisible par le compilateur.

Dans ce cas, le développeur doit le prévoir et le traiter et puisse qu'il s'agit d'un unchecked exception, il peut utiliser le RuntimException pour le gérer

1 Intérêt et définition

2 Hiérarchie organisationnelle des exceptions

3 Le traitement des exceptions

4 Le propagation et la levée d’exception

5 Les exceptions personnalisées

4 Le propagation et la levée d'exception (1/3)

Propagation d'exceptions (1/2)

- Il peut arriver que pour une raison ou une autre, l'on écrive une méthode m qui manipule des instructions sensées provoquer la survenance d'une ou de plusieurs exceptions prévisibles (beaucoup plus de type checked, mais aussi unchecked), mais que l'on ne peut ou veut pas directement traiter dans le corps de cette méthode à l'aide des blocs *try…catch*
  - Dans ce cas, Java oblige que la méthode m propage ces exceptions qu'elle ne traite pas
  - La propagation d'exceptions signifie que la méthode m déclare que c'est une méthode quelconque m' qui la consommera qui se chargera de les traiter à ce moment là ou de les propager aussi à son tour. La propagation est transitive
  - Java utilise le mot clé **throws** pour réaliser une déclaration de propagation d'exceptions

```
public void methode1() throws FileNotFoundException {
   //Corps de la méthode pouvant provoquer
   //l'exception FileNotFoundException
}

public String methode2() throws ParseException, IOException {
   //Corps de la méthode pouvant provoquer
   //les exceptions PaseException, IOException
}
```

```
public void methodeCons1() {
   //méthode appelant methode1() avec traitement de son exception
   try{
     obj.methode1();
   }catch(FileNotFoundException ex) {
      ex.printStackTrace();
   }
}

public String methodeCons2() throws ParseException, IOException {
   //méthode appelant methode2(), mais au lieu de traiter
   //l'exception, la propage à son tour
   return obj.methode2();
}
```

4 Le propagation et la levée d'exception (2/3)

Propagation d'exceptions (2/2)

- Exemple :

```
import java.io.*;

public class ExempleLectureEcritureFichier {

  FileReader fr = new FileReader(f1);
  FileWriter fw = new FileWriter(f2);

  public void recopierFichier(String file1, String file2) throws IOException {
     File f1 = new File(file1);
     File f2 = new File(file2);

     //lecture du contenu du fichier file1 et recopie dans file2
     char[] b = new char[10];
     int var = fr.read(b);
     while(var != -1) {
         fw.write(b);
         var = fr.read(b);
     }
  }

  public static void main(String[] args) throws IOException  {
    ExempleLectureEcritureFichier elef = new ExempleLectureEcritureFichier();
    try{
        elef.recopierFichier("C:/tmp/bidule/file1.txt", "C:/tmp/bidule/file2.txt");
     } catch(IOException e) {
        e.printStackTrace();
     } finally {
        fr.close();
        fw.close();
     }
  }
}
```

4 Le propagation et la levée d'exception (3/3)

Levée d'exception

- Il peut arriver que dans une méthode m d'un programme, l'on souhaite expressément provoquer la survenue d'une exception si une condition non souhaitée arrive
  - Dans ce cas, on crée une instance de la classe d'exception concernée et on utilise le mot clé **throw** pour la levée
  - La levée d'une exception dans m c'est l'action de mettre le programme dans un état anormal à des fins de provoquer son arrêt cette dernière n'est pas traiter dans m ou dans la méthode consommatrice de m .
  - Une méthode qui lève une exception mais ne la traite pas, doit utiliser le mot clé **throws** pour la propager

```
    public void methode1() throws FileNotFoundException {
      //traitements divers
      if(isConditionNonReuniePourTraitementFichier){
         throw new FileNotFoundException();
      }
    }

    public String methode2(String s) {
      try{
        //opération de converion du paramètre s en un nombre
        if(isConversionImpossible){
           throw new NumberFormatException ();
        }
      } catch (NumberFormatException e){
         System.out.println(e.getMessage());
      }
    }
}
```

```
    public void methodeCons1() {
       //méthode appelant methode1() avec traitement de son exception FileNotFoundException
       try{
         obj.methode1();
       }catch(FileNotFoundException ex) {
          ex.printStackTrace();
       }
    }

    public String methodeCons2() throws FileNotFoundException {
       //méthode appelant methode1() avec propagation de l'exception FileNotFoundException levée
       return obj.methode1();
    }

    public String methodeCons3() {
       //méthode appelant methode2() sans besoin de traitement d'exception, car
       //methode2() lève et traite elle-même son exception NumberFormatException
       return obj.methode2();
    }
```

1 Intérêt et définition

2 Hiérarchie organisationnelle des exceptions

3 Le traitement des exceptions

4 Le propagation et la levée d’exception

5 Les exceptions personnalisées

5 Les exceptions personnalisées (1/2)

- Java offre la possibilité aux programmeurs de définir leurs propres types d'exception
- La seule condition est qu'une exception personnalisée doit hériter d'une exception native de Java
- En général, la plupart des exceptions personnalisées sont des sous-classes d'Exception ou de RuntimeException
- Le but de définir une exception personnalisée est d'apporter un niveau de précision fine dans les erreurs qui peuvent être soulevées dans un programme et qui sont liées au besoin métier

  Exemples :

  - Dans le traitement d'un compte bancaire, on peut vouloir levée une exception particulière avec un message bien précis à renvoyer si ce compte dépassait son découvert autorisé
  - Dans le traitement des dates dans un programme, on peut vouloir levée une exception particulière avec un message bien précis à renvoyer si le format attendu en entrée n'est pas le format souhaité

- Toute exception personnalisée se gère exactement pareil comme les exceptions natives étudiées précédemment
- Dans une exception personnalisée, on peut définir plusieurs constructeurs qui appelleront les constructeusr de sa super-classe

5 Les exceptions personnalisées (2/2)

- Exemple :

```
public class DepassementDecouvertException extends Exception {

    public DepassementDecouvertException() {
        super();
    }

   public DepassementDecouvertException(String message) {
         super(message) ;
   }

   public DepassementDecouvertException(Throwable cause) {
        super(cause) ;
   }

   public DepassementDecouvertException(String message, Throwable cause) {
        super(message, cause);
   }

}
```

```
public class GestionCompteBancaire {

   final double SEUIL_DECOUVERT = -100.0d;

   public gererDecouvert(double solde)  throws DepassementDecouvertException {

      if(solde < SEUIL_DECOUVERT) {
          double diff = SEUIL_DECOUVERT – solde;
          throw new DepassementDecouvertException("Solde en dépassement de : " + diff);
      }

    }

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

           GestionCompteBancaire gcb = new GestionCompteBancaire ();
           gcb.gererDecouvert(-200.0d);

        } catch(DepassementDecouvertException ex) {

            System.out.println("L'erreur est : " + ex.getMessage());

        }
    }
}
```


SUCCESS