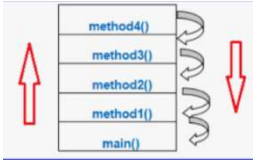
Gestion des exceptions

Afin d'écrire des programmes robustes, le langage Java, proposent un **mécanisme de prise en compte des erreurs, fondé sur la notion d'*exception***.

Une exception est un objet qui peut être émis par une méthode si un événement d'ordre "exceptionnel" (les erreurs rentrent dans cette catégorie) se produit. **La méthode en question ne renvoie alors pas de valeur de retour, mais émet une exception expliquant la cause de cette émission**.

La propagation d'une émission se déroule ainsi :



A. Une exception est générée à l'intérieur de la méthode 4;

B. Si la méthode prévoit un traitement de cette exception, on va au point D, sinon au point C ;

C. L'exception est renvoyée à la méthode 3 ayant appelé la méthode 4, on retourne au point B ; ***Throw***

D. L'exception est traitée et le programme reprend son cours. ***Try … catch***

La gestion d'erreurs par propagation d'exception présente deux atouts majeurs :

- **Une facilité de programmation et de lisibilité** : il est possible de regrouper la gestion d'erreurs à un même niveau. Try …catch...
- **Une gestion des erreurs propre et explicite** : La dissociation d'une valeur de retour « Exception » et de l'exception JAVA permet à cette dernière de décrire précisément la ligne de code ayant provoqué l'erreur et la nature de cette erreur.

5.1 Déclaration

Il est nécessaire de déclarer, pour chaque méthode, les classes d'exception qu'elle est susceptible d'émettre. Cette déclaration se fait à la fin de la signature d'une méthode par le mot-clé **`throws`** à la suite duquel les classes qui peuvent être générées sont précisées.

Exemple :

```
public static int parseInt(String s) throws NumberFormatException;IOException {
 ...
}
```

Une exception peut-être émise dans une méthode de deux manières :

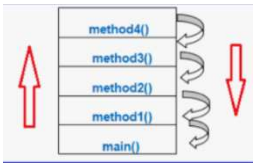
- par la création d'un objet instanciant la classe `Exception` (ou la classe `Throwable`) et la levée explicite de l'exception en utilisant le mot-clé `throw`.
- par une autre méthode appelée dans le corps de la première méthode

L'exemple ci-dessous illustre ce premier cas :

```
public class ExempleException {
   /*Cette méthode renvoie le nom du mois correspondant au chiffre donné en paramètre.Si celui-ci n'est pas
   valide une exception de classe  IndexOutOfBoundsException est levée. */
   public static String month(int mois) throws IndexOutOfBoundsException {
      if ((mois < 1) || (mois > 12)) {
         throw new IndexOutOfBoundsException(
                 "le numero du mois qui est "
                    + mois
                        + " doit être compris entre 1 et 12");
      }
      if (mois == 1) return
         "Janvier";
      else
         return "Autre";
      }
   }
```

5.2 Interception et traitement

Avant de coder le traitement de certaines exceptions, il faut **préciser l'endroit où elles vont être interceptées**.



La méthode `main()` appelle une méthode `1` qui appelle une méthode `2` qui appelle une méthode `3`, qui appelle une méthode `4`, et que cette méthode `4` lève une exception, celle-ci est d'abord transmise à `3` qui peut l'intercepter ou la transmettre à `2`, qui peut aussi l'intercepter ou la transmettre à `1`, qui peut aussi l'intercepter ou la transmettre à `main()`.

L'interception d'exceptions se fait par une sorte de "mise sur écoute" d'une portion de code.

Pour cela on utilise le mot-clé **`try`** suivi du bloc à surveiller. Si aucune exception ne se produit dans le bloc correspondant, le programme se déroule normalement comme si l'instruction **`try`** était absente. Par contre, si une exception est levée, le traitement de l'exception est exécuté puis l'exécution du programme reprend son cours après le bloc testé.

Il est également nécessaire de préciser quelles classes d'exception doivent être interceptées dans le bloc testé. L'interception d'une classe d'exception s'écrit grâce au mot-clé **`catch`** suivi de la classe concerné, d'un nom de variable correspondant à l'objet exception, puis du traitement. Si une exception est levée sans qu'aucune interception ne soit prévue pour sa classe, celle-ci est propagée à la méthode précédente.

Dans l'exemple ci-dessous, le programme demande à l'utilisateur de saisir le numéro d'un mois et affiche à l'écran le nom de ce mois. Les exceptions qui peuvent être levées par ce programme sont traitées.

```
public class ExempleTraitementException {

   public static void main(String[] args) {
      System.out.print("Entrez le numero d'un mois : ");
      try {
         BufferedReader input = new BufferedReader(
                 new InputStreamReader(System.in));
         String choix = input.readLine(); int
         numero = Integer.parseInt(choix);
         System.out.println(ExempleException.month(numero));
      } catch (IndexOutOfBoundsException e) {
         System.err.println("Numero incorrect : "
            + e.getMessage());
      } catch (NumberFormatException e) {
         System.err.println("Entrée incorrecte : "
            + e.getMessage());
      } catch (IOException e) {
         System.err.println("Erreur d'accès : "
            + e.getMessage());
      }
   }
}
```

Trois classes d'exception sont ici traitées pour afficher le message d'erreur :

- `IndexOutOfBoundsException` (levé par la méthode `month`) se produit si le numero entré par l'utilisateur est inférieur à 1 ou supérieur à 12 ;
- `NumberFormatException` (levé par la méthode `parseInt`) qui se produit si le texte entré par l'utilisateur n'est pas convertible en entier ;
- `IOException` (levé par la méthode `readLine`) qui se produit s'il y a eu une erreur d'accès au périphérique d'entrée.

5.3 Classes d'exception

Une classe est considérée comme une classe d'exception dès lors qu'elle hérite de la classe `Throwable`. Un grand nombre de classes d'exception sont proposées dans l'API pour couvrir les catégories d'erreurs les plus fréquentes. Les relations d'héritage entre ces classes permettent de lever ou d'intercepter des exceptions décrivant une erreur à différents niveaux de précision. Les classes d'exception les plus fréquemment utilisées sont récapitulées dans le tableau 5.1.

TABLE 5.1 – Classes d'exception fréquentes

Classe	Description
`ClassCastException`	Signale une erreur lors de la conversion d'un objet en une classe incompatible avec sa vraie classe.
`FileNotFoundException`	Signale une tentative d'ouverture d'un fichier inexistant.

`IndexOutOfBoundsException`	Se produit lorsque l'on essaie d'accéder à un élément inexistant dans un ensemble.
`IOException`	Les exceptions de cette classe peuvent se produire lors d'opérations d'entrées/ sorties.
`NullPointerException`	Se produit lorsqu'un pointeur `null` est reçu par une méthode n'acceptant pas cette valeur, ou lorsque l'on appelle une méthode ou une variable à partir d'un pointeur `null`.

Si aucune des classes d'exception ne correspond à un type d'erreur que vous souhaitez exprimer, vous pouvez également écrire vos propres classes d'exception.

Pour cela, il suffit de faire hériter votre classe de la classe `java.lang.Exception`.

5.4 Classification des erreurs en Java

On peut finalement distinguer quatre types de situations d'erreurs en Java :

- Erreurs de compilation. Avant même de pouvoir exécuter le programme, notre code source génère des erreurs par le compilateur. Il faut alors réviser et corriger le code pour ne plus avoir d'erreurs.
- Erreurs d'exécution. Alors que notre programme est en cours d'exécution, la JVM étant mal configurée ou corrompue, le programme s'arrête ou se gèle. A priori, c'est une erreur non pas due à notre programme, mais à la **configuration ou l'état de l'environnement d'exécution** de notre programme.
- Exception non vérifiée. Alors que notre programme est en cours d'exécution, une trace de la pile des exceptions est affichée, pointant vers une partie de notre code sans gestion d'exception. Visiblement, nous avons utilisé du code qui est capable de lever une exception non vérifiée (comme **`NullPointerException`**). Il faut modifier le programme pour que cette situation ne survienne pas.
- Exception vérifiée. Alors que notre programme est en cours d'exécution, une trace de la pile des exceptions est affichée, pointant vers une partie de notre code avec gestion d'exception. Visiblement, nous avons produit du code qui est capable de lever une exception vérifiée (comme `FileNotFoundException`) mais les données passées à notre programme ne valident pas ces exceptions (par exemple, lorsque l'on essaie d'ouvrir un fichier qui n'existe pas). Il faut alors revoir les données passées en paramètre du programme. Notre code a bien détecté les problèmes qu'il fallait détecter. Le chapitre suivant sur les entrées/sorties présentent de nombreux exemples relatifs à ce cas.

Gestion des entrées/sorties simples

Le package java.io propose un ensemble de classes permettant de gérer la plupart des entrées/sorties d'un programme. En java, les notions de flux d'entrée et flux de sortie sont utilisées.

Deux cas sont illustrés dans ce chapitre :
- les interactions avec un utilisateur (entrée clavier et sortie écran)
- les accès en lecture ou écriture à un fichier.

6.1 Flux d'entrée

Un flux d'entrée est une instance d'une sous-classe de InputStream. Les classes les plus couramment utilisées sont :

- **ByteArrayInputStream** permet de lire le flux d'entrée sous la forme d'octets (*byte*) ;
- **DataInputStream** permet de lire le flux d'entrée sous la forme de types de données primitifs de Java. Il existe des méthodes pour récupérer un entier, un réel, un caractère,…
- **FileInputStream** est utilisé pour lire le contenu d'un fichier. Les objets de cette classe sont souvent encapsulés dans un autre objet de classe `InputStream` qui définit le format des données à lire.
- **ObjectInputStream** permet de lire des objets (c-à-d des instances de classes Java) à partir du flux d'entrée, si ces objets implémentent les interfaces `java.io.Serializable` ou `java.io.Externalizable`.
- **Reader** n'est pas une sous-classe de `InputStream` mais représente un flux d'entrée pour chaînes de caractères. Plusieurs sous-classes de `Reader` permettent la création de flux pour chaînes de caractères.
- **Scanner** n'est pas une sous-classe de `InputStream`, mais un `Iterator` qui permet de lire un flux (fichier ou chaîne de caractères par exemple) "mot" par "mot" en définissant le délimiteur entre les mots (espace par défaut).

La lecture de données à partir d'un flux d'entrée suit le déroulement suivant :

1. **Ouverture du flux :** Elle se produit à la création d'un objet de la classe `InputStream`. Lors de l'appel au constructeur, on doit préciser quel élément externe est relié au flux (par exemple un nom de fichier ou un autre flux).
2. **Lecture de données :** Des données provenant du flux sont lues au moyen de la méthode `read()`) ou d'une méthode équivalente. La méthode précise à employer dépend du type de flux ouvert.
3. **Fermeture du flux :** Quand le flux n'est plus nécessaire, il doit être fermé par la méthode `close()`.

6.1.1 Lecture des entrées clavier

Les données provenant de l'utilisation du clavier sont transmises dans un flux d'entrée créé automatiquement pour toute application Java. On accède à ce flux par la variable statique de la classe java.lang.System qui s'appelle in. La sous-classe de Reader est utilisée pour récupérer les entrées de l'utilisateur sous la forme d'une chaîne de caractères.

Exemple :

```
import java.io.*;

public class Clavier {

public static void main(String[] args) {
  try {
       BufferedReader flux = new BufferedReader(
         new InputStreamReader(System.in));
       System.out.print("Entrez votre prenom : ");
       String prenom = flux.readLine();
       System.out.println("Bonjour " + prenom);
       flux.close();
     } catch (IOException ioe) {
          System.err.println(ioe);
     }
  }
}
```

6.1.2 Lecture à partir d'un fichier

Un fichier est représenté par un objet de la classe java.io.File. Le constructeur de cette classe prend en paramètre d'entrée le chemin d'accès du fichier. Le flux d'entrée est alors créé à l'aide de la classe FileInputStream sur lequel on peut lire caractère par caractère grâce à la méthode read().

L'exemple suivant présente une méthode pour afficher le contenu d'un fichier :

```
import java.io.*; public

class LectureFichier {

  public static void main(String[] args) {
    try {
      File fichier = new File(”monFichier.txt”);
      FileInputStream flux = new FileInputStream(fichier);
      int c;
      while ((c = flux.read()) > -1) {
        System.out.write(c);
      }
      flux.close();
    } catch (FileNotFoundException e) {
      e.printStackTrace();
    } catch (IOException e) {
      e.printStackTrace();
    }
  }
}
```

Il arrive souvent d’enregistrer des données dans des fichiers textes. Il peut alors être utile d’utiliser un `BufferedReader` ou un `Scanner` pour effectuer la lecture. Dans les exemples suivants, on considère une matrice 10×10 enregistrée dans un fichier texte `matrice.txt` ligne par ligne, avec les colonnes séparées par des espaces :

```
import java.io.*;

public class LectureMatrice{

  public static void main(String[] args) {
    try {
      FileReader fileReader = new FileReader(”matrice.txt”);
      BufferedReader reader = new BufferedReader(fileReader);
      while (reader.ready()) {
        String[] line = reader.readLine().split(” ”);
        for (String s : line) {
          System.out.print(s);
        }
        System.out.println();
      } reader.close();
      fileReader.close();
    } catch (FileNotFoundException e) {
      e.printStackTrace();
    } catch (IOException e) {
      e.printStackTrace();
    }
  }
}
```

On peut effectuer un traitement similaire avec un `Scanner` :

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

public class LectureMatriceScanner {
  public static void main(String[] args) {
     try {
       Scanner fileScanner = new Scanner(new File(”matrice.txt”));
       while (fileScanner.hasNextLine()) {
            System.out.print(fileScanner.next());
       }
     } catch (FileNotFoundException e) {
       e.printStackTrace();
     }
   }
  }
```

6.1.3 Lecture d’objets enregistrés

Il est parfois utile d’enregistrer l’état d’objet (de n’importe quelle classe implémentant les interfaces `java.io.Serializable` ou `java.io.Externalizable`) pour des exécutions futures. Le flux d’entrée est encore créé à l’aide de la classe `FileInputStream` et est ensuite encapsulé dans un autre flux spécifiant le format des données à lire. L’exemple suivant illustre la lecture d’un objet de la classe `Date` dans un fichier nommé `monFichier.dat` :

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

public class LectureDate { public static
  void main(String[] args) {
    try {
      File fichier = new File(”monFichier.dat”);
      ObjectInputStream flux = new ObjectInputStream(
      new FileInputStream(fichier));
      Date laDate = (Date) flux.readObject();
      System.out.println(laDate);
      flux.close();
    } catch (IOException ioe) {
      System.err.println(ioe);
    } catch (ClassNotFoundException cnfe) {
      System.err.println(cnfe);
    }
  }
}
```

L’objet qui est lu dans le fichier doit être une instance de la classe `java.util.Date`.

6.2 Flux de sortie

Un flux de sortie est une instance d'une sous-classe de OutputStream. Comme pour les flux d'entrée, chaque classe de flux de sortie a son propre mode d'écriture de données. Les classes les plus couramment utilisées sont :

- **ByteArrayOutputStream** permet d'écrire des octets vers le flux de sortie ;
- **DataOutputStream** permet d'écrire des types de données primitifs de Java vers le flux de sortie.
- **FileOutputStream** est utilisé pour écrire dans un fichier. Les objets de cette classe sont souvent encapsulés dans un autre objet de classe `OutputStream` qui définit le format des données à écrire.
- **ObjectOutputStream** permet d'écrire des objets (c-à-d des instances de classes Java) vers le flux de sortie, si ces objets implémentent les interfaces `Serializable` ou `Externalizable`.
- **Writer** n'est pas une sous-classe de `OutputStream` mais représente un flux de sortie pour chaînes de caractères. Plusieurs sous-classes de `Writer` permettent la création de flux pour chaînes de caractères.

L'écriture de données vers un flux de sortie suit le même déroulement que la lecture d'un flux d'entrée

1. **Ouverture du flux :** Elle se produit lors de la création d'un objet de la classe `OutputStream`.
2. **Ecriture de données :** Des données sont écrites vers le flux au moyen de la méthode `write()` ou d'une méthode équivalente. La méthode précise à employer dépend du type de flux ouvert.
3. **Fermeture du flux :** Quand le flux n'est plus nécessaire, il doit être fermé par la méthode `close()`.

6.2.1 Ecriture sur la sortie standard "écran"

Comme pour les entrées du clavier, l'écriture vers l'écran fait appel à la variable statique out de la classe System. On appelle généralement la méthode System.out.printouSystem.out.println comme cela a été fait dans de nombreux exemples de ce livret.

6.2.2 Ecriture dans un fichier

L'écriture dans un fichier se fait par un flux de la classe `FileOutputStream` qui prend en entrée un fichier (instancedelaclasse `File`).Ce flux de sortie permet d'écrire des caractères dans le fichier grâce à la méthode `write()`. L'exemple suivant présente une méthode pour écrire un texte dans un fichier :

```
import java.io.*; public class

EcritureFichier {

  public static void main(String[] args) {
    try {
      File fichier = new File(”monFichier.txt”);
      FileOutputStream flux = new FileOutputStream(fichier);
      String texte = ”Hello World!”;
      for (int i = 0; i < texte.length(); i++) {
      flux.write(texte.charAt(i));
      } flux.close();
    } catch (FileNotFoundException e) {
      e.printStackTrace();
    } catch (IOException e) {
      e.printStackTrace();
    }
  }
}
```

On peut également utiliser un `Writer` comme un `FileWriter` pour écrire des chaînes de caractères dans un fichier assez simplement. Dans l'exemple suivant, on écrit une série de 10 lignes de 10 entiers aléatoires séparés par des espaces dans un fichier pouvant être lu par la classe `LectureMatrice` :

```
import java.io.*; import

java.util.Random; public

class EcritureMatrice {

  public static void main(String[] args) {
    try {
      FileWriter writer = new FileWriter(”random.txt”);
      Random generator = new Random(System.currentTimeMillis());
      for (int i = 0; i < 9; i++) {
        for (int j = 0; j < 9; j++)
           writer.write(generator.nextInt() + ” ”);
        writer.write(”\n”);
      }
      writer.close();
    } catch (FileNotFoundException e) {
      e.printStackTrace();
    } catch (IOException e) {
      e.printStackTrace();
    }
  }
}
```

6.2.3 Ecriture d'objets

Le flux de sortie peut également être encapsulé dans un flux de type `ObjectOutputStream`, comme le montre l'exemple suivant pour écrire la date courante dans un fichier :

```
import java.io.*; import
java.util.Date; public
class EcritureDate {

  public static void main(String[] args) {
    try {
      File fichier = new File(”monFichier.dat”);
      ObjectOutputStream flux = new ObjectOutputStream(
      new FileOutputStream(fichier));
      flux.writeObject(new Date()); flux.close();
    } catch (IOException ioe) {
      System.err.println(ioe);
    }
  }
}
```

Ce fichier peut ensuite être lu par la classe `LectureDate`.

* **Remarque importante :** Nous avons présenté dans ce chapitre plusieurs exemples d'entrée/sortie utilisant différents modes de lecture/écriture (avec ou sans flux). Nous conseillons toutefois d'utiliser en *priorité* un `Scanner` pour la lecture dans un fichier, et un `FileWriter` pour l'écriture dans un fichier.

TP : Reality Tips

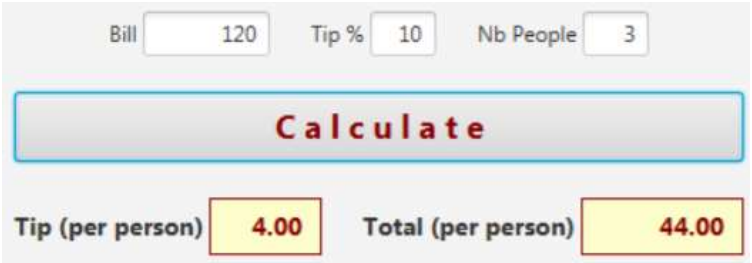
Exercice 1 : Application

Créer une application en mode graphique pour gérer le calcul des tips (Pourboires).
L'utilisateur entre le montant « Bill », le % du Tip et le nombre de personne (Utilisation de classes).

Le système calcule :

- Le montant du tip par personne
- Le total à payer par personne

Les erreurs possibles doivent être traitées au travers d'Exception Java.



Exercice 2 : Optimisation de l'applicatif

Ajouter un champ « Date du calcul » de type Date « dd/mm/yyyy » et un label en haut d'application qui affichera le message d'erreur.
Au click sur le bouton « Calculate », l'application doit vérifier via le mécanisme d'exception Java que :

- les champs numériques sont bien de type numérique
- le champ Date est bien de type date
- les saisies du nombre 0

Si ce n'est pas le cas, il faut remplir le label avec un message d'erreur.

Exercice 3 : Enregistrement de l'historique

Ajouter un traitement sur le bouton calculate permettant de stocker dans un fichier texte pour la date les informations Bill, Tip et NbPeople. Il doit y avoir un enregistrement par Date. Si la date existe dans le fichier, les informations seront remplacées.
Exemple de fichier :
20/10/2020 ;20 ;10 ;3
21/10/2020 ;40 ;8 ;2

Exercice 4 : Initialisation de l'application

Ajouter un traitement lors de la saisie d'une date qui renseigne les 3 champs Bill, Tip et NbPeople avec les données remontées du fichier si un historique a été enregistré dans le fichier à cette date.

Exercice 5 : Jeu en réseau communiquant par fichier (En binôme)

Créer un Morpion en mode graphique qui communique par fichier avec le morpion de votre binôme.
Le fichier sera écrit et lu par chacun des 2 binômes lors de son tour.
La lecture du fichier mettra à jour l'interface de l'utilisateur avec les données récupérées dans le fichier.
Lors d'un click sur une case vide, le fichier sera mis à jour et le binôme pourra à son tour récupérer les données. Le fichier contiendra donc les données (O ou X) de chaque joueur.