Gestion des exceptions et des entrées-sorties

Christelle CAILLOUET
(christelle.caillouet@unice.fr)

Gestion des exceptions

Gestion des exceptions

- Objectifs :
  - Rendre le programme robuste
  - Limiter les instructions de traitement des cas exceptionnels

- Mécanisme très souple de prise en compte fondé sur la notion d'**exception**, qui permet :
  - De dissocier la **détection** d'une anomalie de son **traitement**;
  - De séparer la gestion des anomalies du reste du code

L'exception

- Définition : c'est un **objet** qui peut être émis par une méthode si un évènement d'ordre *exceptionnel* se produit.

- La méthode émet alors une exception expliquant la cause de cette émission.

Mot-clés

- Lancer une exception : **`throw (throws)`**
- Surveiller une partie du code : **`try`**
- Traiter l'exception : **`catch`**

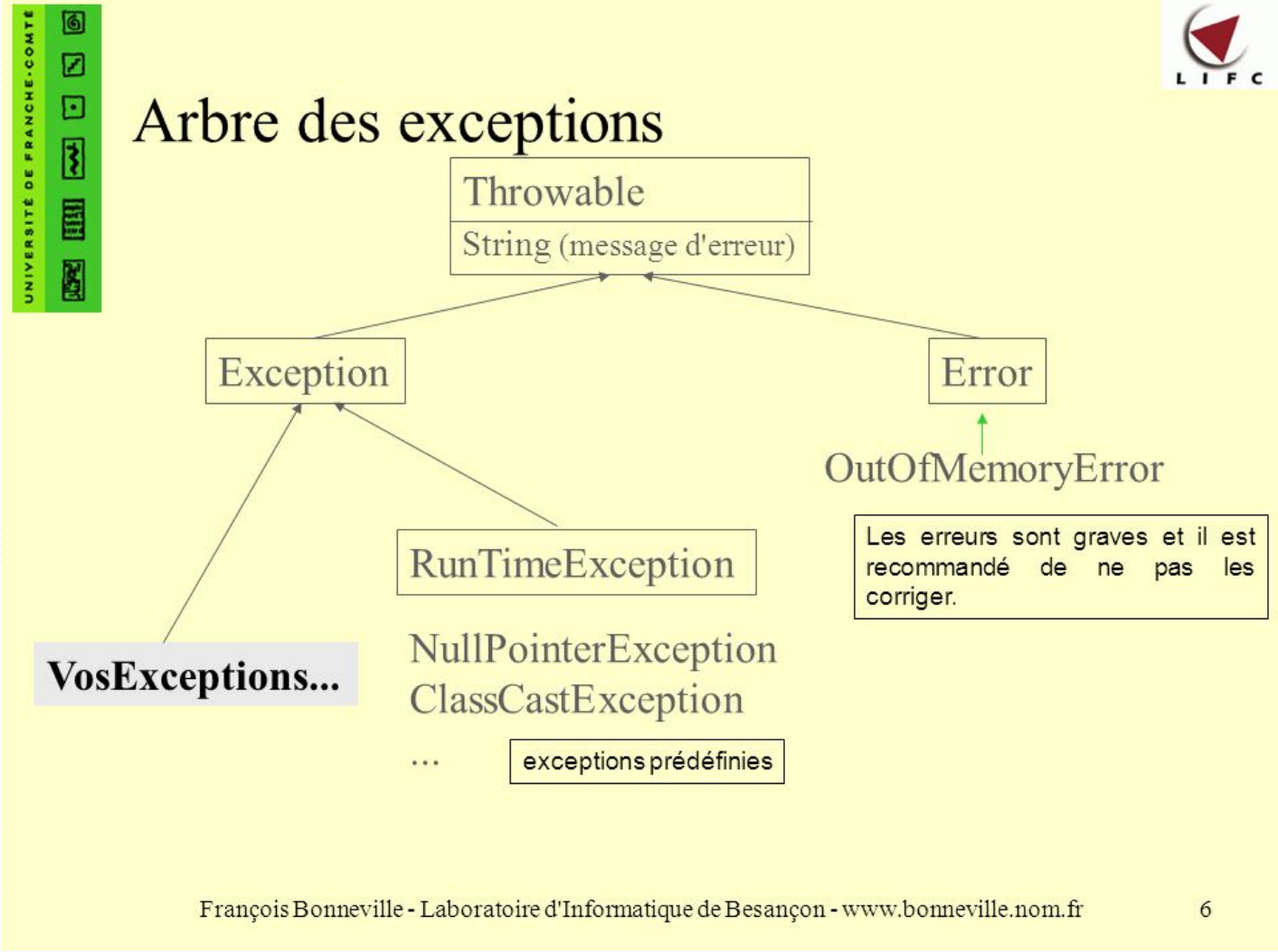
- Classes d'exception standards prédéfinies :
  - **Throwable**
  - **Exception**
  - **RuntimeException**

- Toutes les exceptions dérivent de la classe *Exception* qui dérive de la classe *Throwable*

Classes d'exceptions

- Classes qui héritent de la classe `Throwable`
- Un grand nombre de classes d'exceptions existent dans l'API pour couvrir les erreurs les plus fréquentes
- Si aucune classe d'exception ne correspond au type d'erreur rencontré, on peut écrire de nouvelles classes d'exceptions ➔ elles doivent impérativement hériter de `java.lang.Exception`

Hiérarchie des classes d'exceptions



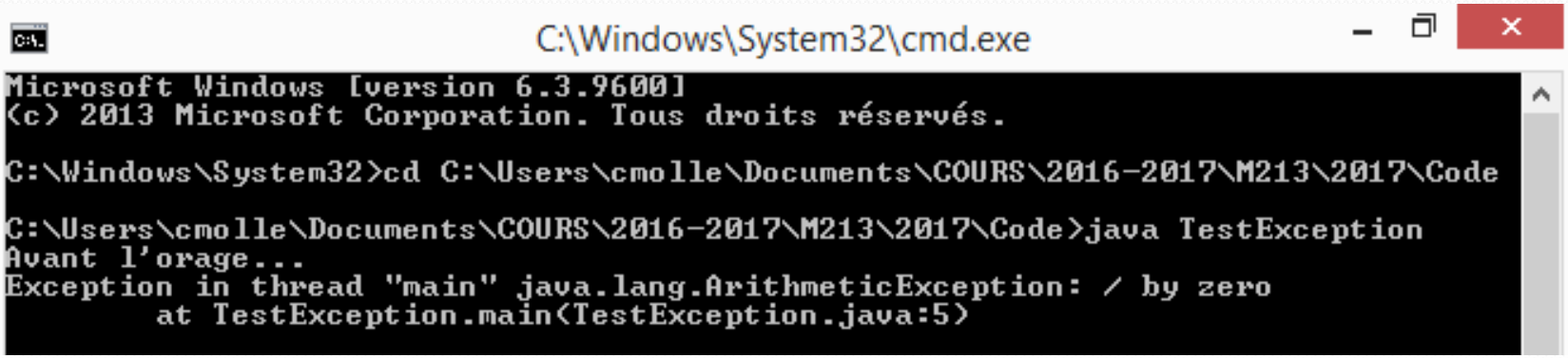
Exceptions levées par la JVM

- Erreur de compilation ou d'exécution
  - `NoClassDefFoundError, ClassFormatError`
- Problème d'entrée/sortie
  - `IOException, AWTException`
- Problème de ressources
  - `OutOfMemoryError, StackOverflowError`
- Erreurs de programmation (Runtime)
  - `NullPointerException, ArrayIndexOutOfBoundsException, ArithmeticException, …`

```java
public class TestException{
    public static int divisionEntiere(int x, int y){
        return x/y;
    }

    public static void main(String[] args){
        int a = 2, b = 0;
        System.out.println("Avant l'orage...");
        divisionEntiere(a,b);
        System.out.println("Après l'orage...");

    }
}
```



L'exécution du programme est interrompue.

Déclaration

- Instruction `throw` :
  - Lancement d'une exception : `throw objet;`

```
if (k<0)
    throw new RuntimeException("k négatif");
```

Déclaration

- Pour chaque méthode : quelles classes d'exception est-elle susceptible de déclencher ? (elle ou les méthodes appelées)

- La méthode qui lève une exception doit déclarer cette action *potentielle*
  - À la fin de la déclaration de la méthode : **`throws`**

```
import java.io.IOException;
public void maMethode(int entier) throws IOException
{
    //code de la methode
}
```

```java
public class TestException{
    public static int divisionEntiere(int x, int y) throws ArithmeticException{
        if (y == 0) throw new ArithmeticException("Division par 0 !");
        return x/y;
    }


    public static void main(String[] args){
        int a = 2, b = 0;
        System.out.println("Avant l'orage...");
        divisionEntiere(a,b);
        System.out.println("Apres l'orage...");

    }
}
```

```
C:\Users\cmolle\Documents\COURS\2016-2017\M213\2017\Code>java TestException
Avant l'orage...
Exception in thread "main" java.lang.ArithmeticException: Division par 0 !
        at TestException.divisionEntiere(TestException.java:3)
        at TestException.main(TestException.java:11)
```

Gestion d'erreurs par propagation

1. Une exception est générée à l'intérieur d'une méthode

2. Si la méthode prévoit un traitement de cette exception, on va au point 4 ; sinon au point 3

3. L'exception est renvoyée à la méthode ayant appelé la méthode courante, on retourne au point 2

4. L'exception est traitée et le programme reprend son cours après le traitement de l'exception

Exemple

- Une méthode A appelle une méthode B qui appelle une méthode C qui appelle un méthode D.
- La méthode D lève une exception.
- L'exception est transmise à C qui peut l'intercepter ou la transmettre à B.
- B peut aussi l'intercepter ou la transmettre à A.

- L'interception se fait par une *mise sur écoute* d'une portion du code…

Interception

- Surveillance des exceptions éventuelles lancées par une séquence cible d'instructions

- Surveiller ne veut pas dire traiter !
  (mais obligation de traiter si on surveille…)

Comment intercepter une exception

- Exécuter un bloc d'instructions **A** suspect
- Si **A** ne fonctionne pas, l'erreur est prise en charge par un bloc d'instructions **B** (cf. traitement)

```
try{
      // A
} catch(Exception e) {
      // B
}
```

- Si aucune erreur ne se produit dans **A**, **B** n'est pas exécuté

Traitement

```
try{
      // A
} catch(Exception e) {
      // B
}
```

- Si un exception est levée dans **A**, l'exception levée est transmise au gestionnaire d'exceptions (**catch**) qui va exécuter la séquence de traitement **B**.

- La séquence de traitement peut elle-même :
  - Appeler des méthodes
  - Lever des exceptions

```
try {
      // A
} catch (Exception e){
      // B
      throw new Exception();
}
```

```java
public class TestException{
    public static int divisionEntiere(int x, int y) throws ArithmeticException{
        return x/y;
    }


    public static void main(String[] args){
        int a = 2, b = 0;
        System.out.println("Avant l'orage...");
        try {
            divisionEntiere(a,b);
        } catch (ArithmeticException e){
            System.out.println(e.getMessage());
        }
        System.out.println("Apres l'orage...");

    }
}
```

```
C:\Users\cmolle\Documents\COURS\2016-2017\M213\2017\Code>javac TestException.jav
a

C:\Users\cmolle\Documents\COURS\2016-2017\M213\2017\Code>java TestException
Avant l'orage...
/ by zero
Apres l'orage...
```

```java
public class TestException{
    public static int divisionEntiere(int x, int y) throws Exception{
        if (y == 0) throw new Exception("Division par 0 !");
        return x/y;
    }


    public static void main(String[] args){
        int a = 2, b = 0;
        System.out.println("Avant l'orage...");
        try {
            divisionEntiere(a,b);
        } catch (Exception e){
            System.out.println(e.getMessage());
        }
        System.out.println("Apres l'orage...");

    }
}
```

```
C:\Users\cmolle\Documents\COURS\2016-2017\M213\2017\Code>java TestException
Avant l'orage...
Division par 0 !
Apres l'orage...
```

Reprise après exception

- Si une exception est levée et capturée, l'exécution reprend à la suite du catch (et non là où l'exception a été levée !)

```
public static void main(String[] args){
    int a = 2, b = 0;
    System.out.println("Avant l'orage...");
    try {
        divisionEntiere(a,b);
        System.out.println("Suite...");
    } catch (Exception e){
        System.out.println(e.getMessage());
    }
    System.out.println("Apres l'orage...");

}
```

Ne s'affichera pas !

Redéclenchement d'une exception

- Malgré la phase de traitement, on peut décider de propager l'exception à un niveau supérieur

➔Propagation directe

```
try {
    // A
} catch (Exception e){
    // B
    throw e;
}
```

- Très utile si l'on ne peut résoudre localement qu'une partie du problème

Plusieurs gestionnaires d'exception

- Plusieurs bloc **catch** peuvent se succéder

```
try {
    divisionEntiere(a,b);
} catch (ArithmeticException e){
    System.out.println(e.getMessage());
} catch (Exception e) {
    System.out.println(e.getMessage());
}
```

- Attention aux relations d'héritage !

Bloc finally

- Le bloc finally est toujours exécuté lorsque le bloc try se termine (sauf si la JVM crashe entre temps…)

```
try{
        System.out.println("begin");
        throw new Exception();

} catch (Exception e) {
        System.out.println("in catch");
} finally {
        System.out.println("hello world");
}
```

- Affiche :

begin
in catch
hello world

Gestion des entrées - sorties

Clavier, écran, fichiers, erreurs, ...

Le package `java.io`

https://docs.oracle.com/javase/8/docs/api/java/io/package-summary.html

- Ensemble de classes qui gèrent la plupart des entrées-sorties d'un programme
- Gestion des entrées-sorties = créer un **objet** *flux* dans lequel transitent les données à envoyer ou recevoir
- Un **flux** connecte un objet Java à un autre élément
- 2 cas étudiés :
  - Interactions avec un utilisateur (entrée clavier, sortie écran)
  - Lecture, écriture dans un fichier

Etapes des échanges

1. **Ouverture du flux**
2. **Lecture/écriture des données**
3. **Fermeture du flux**

Un flux (*stream*)

- Représente un **canal de communication**

- Dans lequel circulent des données
  - octets (Byte), caractères(Character), …
  - Codage des caractères UNICODE sur 2 octets

- Ce flux peut être en entrée (ou lecture), ou en sortie (ou écriture)
  - Peut utiliser un buffer pour le traitement de lots

Classes java.io.*

- Flux de données :
  - Classes **`InputStream`** et **`OutputStream`**
  - Classes **`Reader`** et **`Writer`**

- Système de fichiers : Classe **`File`**

- Noms des classes :
  - Origine ou destination du flot : tampon, fichier, tableau, tube
  - Sens (lecture ou écriture) : Input, Output, Reader, Writer
  - Octets ou Caractères : Stream ou Reader/Writer

`BufferedInputStream` : tampon/Lecture/octet
`ByteArrayOutputStream` : tableau/Ecriture/octet
`BufferedWriter` : tampon/caractère

Types de flux

- **Flux d'octets**
  - Toutes les classes héritent de **`InputStream`** ou **`OutputStream`**
  - Flux d'E/S standard : **in, out** et **err**
- **Flux de caractères**
  - Les classes dérivent des classes abstraites **`Reader`** et **`Writer`**
- Méthode de lecture : `read()`

Exemple :
lire un octet au clavier

```
try{
  byte b;
  int val = System.in.read();
  if(val != -1) b = (byte)val;
  System.out.write(b);
}
catch(IOException e){}
```

Flux tampon

- améliorent les performances

- classe **`BufferedReader`**

```
public String readLine() throws IOException
```

➔ lit une ligne de texte et la retourne comme un objet String

Lecture de caractères au clavier

```
import java.io.*;

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

Création de flux pour chaînes de caractères

Flux d'entrée : variable statique *in* de la classe *java.lang.System*

Lecture des données

Fermeture du flux

```
C:\Users\cmolle\Documents\COURS\2016-2017\M213\2017\Code>java Clavier
Entrer votre prenom : Bob
Bonjour Bob
```

Flux sur fichiers

- Un fichier est un objet de `java.io.File` construit à partir du chemin d'accès du fichier

- Mêmes types de flux sur fichiers :
  - **Flux d'octets** : `FileInputStream/ FileOutputStream` à partir d'un objet de type `File`
  - **Flux de caractères** : `BufferedReader/ BufferedWriter` à partir d'un objet de type `FileReader/FileWriter`

Lecture par octets

```
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();
        }
    }
}
```

Lecture par ligne de caractères

```
import java.io.*;

public class LectureFichier{
    public static void main(String[] args){
        try {
            FileReader fichier = new FileReader("monfichier.txt");
            BufferedReader reader = new BufferedReader(fichier);
            while (reader.ready()){
                String line = reader.readLine();
                System.out.println(line);
            }
            reader.close();
            fichier.close();
        } catch (FileNotFoundException e) {
            e.printStackTrace();
        } catch (IOException e){
            e.printStackTrace();
        }
    }
}
```

La classe `Scanner`

https://docs.oracle.com/javase/8/docs/api/java/util/Scanner.html

- Classe du package `java.util`
- Autre manière d'effectuer la lecture
- Découpe le fichier (ou la ligne) « en morceaux » en fonction d'un délimiteur
- Peut lire un entier, double, String, …

Lecture avec Scanner

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

public class LectureFichier{
    public static void main(String[] args){
        try {
            Scanner fileScanner = new Scanner(new File("monFichier.txt"));
            while (fileScanner.hasNextLine()){
                Scanner lineScanner = new Scanner(fileScanner.nextLine());
                while (lineScanner.hasNext()){
                    System.out.println(lineScanner.next());
                }
            }
        } catch (FileNotFoundException e) {
            e.printStackTrace();
        }
    }
}
```

Ecriture dans un fichier

```
import java.io.*;

public class EcritureFichier{
    public static void main(String[] args){
        try {
            FileWriter fichier = new FileWriter("monfichier.txt");
            BufferedWriter writer = new BufferedWriter(fichier);
            String s = "Hello world !";
            writer.write(s);
            writer.close();
            fichier.close();
        } catch (FileNotFoundException e) {
            e.printStackTrace();
        } catch (IOException e){
            e.printStackTrace();
        }
    }
}
```

Flux d'objets

- 3 niveaux :

1. Une instance de `ObjectInputStream/ ObjectOutputStream`
   2. se construit à partir d'un objet de type `FileInputStream/ FileOutputStream`
      3. qui se construit à partir d'un objet de type `File`

- On écrit/lit l'objet dans le fichier à l'aide de la méthode
  `writeObject(Object obj)`
  `readObject()`

Exemple

```
File file = new File("toto.txt");
ObjectOutputStream stream;
// Ouverture du fichier
stream = new ObjectOutputStream(new FileOutputStream(file));
// Ecriture dans le fichier
stream.writeObject(new Etudiant("Ivan", "Wilfried"));
// Fermeture du fichier
stream.close();
```

La fin de fichier

- La classe `ObjectInputStream` ne possède pas de méthode pour détecter la fin du fichier.

➔ Utilisation du mécanisme des exceptions

- Tentative de lecture en fin de fichier : `EOFException`

- Pour détecter la fin de fichier, il suffit d'attraper cette exception !