Introduction à Java

Cours 5: gestion des exceptions et entrées sorties

Stéphane Airiau

Université Paris-Dauphine

Gestion des Exceptions

Gérer l'inattendu !

Exemple

```
public static Random generator = new Random();

public static int randInt(int low, int high){
   return low + (int) (generator.nextDouble() * (high-low +1));
}
```

Que se passe-t-il si l'utilisateur écrit

```
randInt(10,5);
```

On pourrait modifier le code de `randInt` pour prendre en compte ce type d'erreur (mais peut être que l'utilsateur a fait une erreur d'interprétation plus profonde...)

Exemple (suite)

`Java` nous permet de lever une **exception** adaptée, par exemple :

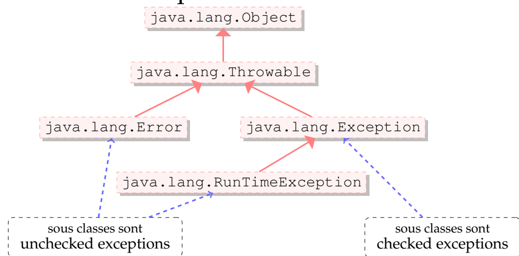
```
if (low > high)
   throws new IllegalArgumentException(
     "low should be <= high but low is " + low + " and high is " + high);
```

On lève une exception d'une classe qui porte un nom parlant `IllegalArgumentException` avec un message qui aidera au deboggage.

Quand une exception est levée, l'exécution "normale" est interrompue. Le contrôle de l'exécution passe à un gestionnaire d'erreur.

Gestion des erreurs

Java possède un mécanisme de gestion des erreurs, ce qui permet de renforcer la sécurité du code. On peut avoir différents niveaux de problèmes :

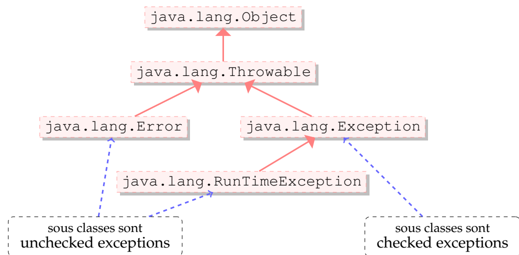


- Error représente une erreur grave intervenue dans la machine virtuelle (par exemple OutOfMemory)
- La classe Exception représente des qui sont reportées au développeur.

➡ le développeur a la possibilité de **gérer** de telles erreurs et **éviter** que l'application ne se termine

➡ on veut gérer les erreurs que l'on peut anticiper

Gestion des erreurs



- `checked Exception` représente des cas où on peut anticiper le problème (par exemple entrée-sortie).
- `RunTimeException` ne peuvent pas être vérifiée lors de la compilation
  plutôt des erreurs causées par le programmeur (`NullPointerException`) que des erreurs causées par une source incontrolable (ex entrée-sortie).

Gestion des erreurs

- Utilisez des exceptions pour des conditions exceptionnelles
- checked exception : conditions pour lesquelles on peut raisonnablement espérer pouvoir récupérer et poursuivre l'exécution
  ➯ force le développeur à gérer une condition exceptionnelle (éviter qu'il ne la néglige trop)
  ➯ *"exception d'utilisation"* : ces erreurs doivent être déclarées dans la signature de la fonction si elles ne sont pas capturées par la fonction.
- running time exception : indique une erreur de programmation `NullPointerException` est "unchecked" (sinon, il faudra toujours traiter cette erreur car elle peut intervenir partout !)

Code reuse : utilisez au plus les exceptions de la bibliothèque standard

- `IllegalArgumentException` : argument dont la valeur n'est pas appropriée
- `IllegalStateException` l'objet n'est pas "prêt" pour faire cet appel
- ... on pourrait utiliser seulement ces deux exceptions car la plupart des erreurs correspondent à un argument ou un état illégal
  - `NullPointerException`
  - `IndexOutOfBoundsException`
  - `ArithmeticException`
  - `NumberFormatException`
- pas une science exacte :
  ex : on possède une méthode pour tirer au sort k dominos dans la pioche, k étant un argument de ma méthode. Supposons qu'un utilisateur demande de tirer k dominos alors qu'il y a $k-1$ dominos dans la pioche. On peut utiliser
  - `IllegalArgumentException` : k est trop grand
  - `IllegalStateException` pas assez de dominos dans la pioche

Déclarer des checked exceptions

On peut utiliser une clause **throws** pour déclarer une méthode dont on peut anticiper l'échec.

```
public void write(Object obj, String filename)
    throws IOException, ReflectiveOperationException
```

Dans la déclaration, on peut grouper les erreurs dans une super classe, ou bien les lister. Dans l'exemple, on lève des exceptions de type `IO`, mais on pourrait nommer plus précisemment toutes les types possibles (sous classes de `IOException`).

Levée d'exception

Lors de la détection d'une erreur

- un objet qui hérite de la classe `Exception` est créé
- ⇒ ce qui s'appelle **lever une exception**
- l'exception est propagé à travers la pile d'exécution jusqu'à ce qu'elle soit traitée.

```java
int[] tab = new int[5];
tab[5]=0;
```

```
Exception in thread "main"
java.lang.ArrayIndexOutOfBoundsException: 5
at Personnage.main(Personnage.java:2)
```

```java
int d=10,t1=5,t2=5;
System.out.println("vitesse:" + d / (t2-t1));
```

```
Exception in thread "main"
java.lang.ArithmeticException: / by zero
at Personnage.main(Personnage.java:21)
```

Le bloc `try` ... `catch`

- bloc `try` : le code qui est susceptible de produire des erreurs
- on récupère l'exception créée avec le `catch`.
- on peut avoir plusieurs blocs `catch` pour capturer des erreurs de types différentes.
- en option, on peut ajouter un bloc `finally` qui sera toujours exécuté (qu'une exception ait été levée ou non)

Lorsqu'une erreur survient dans le bloc `try`,

- la suite des instructions du bloc est abandonnée
- les clauses `catch` sont testés **séquentiellement**
- le premier bloc `catch` correspondant à l'erreur est exécuté.
  ➫ l'ordre des blocs `catch` est donc de l'erreur la plus spécifique à la plus générale !
  ➫on a un mécanisme de **filtre**

Choisir où traiter une exception

Lorsqu'on utilise un morceau de code qui peut lever une exception, on peut

- traiter l'exception immédiatement dans un bloc `try` ... `catch`
- propager l'erreur :

```
public Domino[] tirer(int nb) throws IllegalStateException {
   ...
}
...
public void distribuer() {
  ...
  Domino[] jeu = tirer(7);
  ...
}
```

Ici, on a une erreur, car l'exception qui peut être levée dans `tirer` n'est pas traitée dans la méthode `distribue`.

Choisir où traiter une exception

Lorsqu'on utilise un morceau de code qui peut lever une exception, on peut

- traiter l'exception immédiatement dans un bloc `try` ... `catch`
- propager l'erreur :

```
public Domino[] tirer(int nb) throws IllegalStateException {
   ...
}
...
public void distribuer() throws IllegalStateException {
  ...
  Domino[] jeu = tirer(7);
  ...
}
```

Dans l'exemple, on ne traite pas l'exception dans la méthode `distribuer`, on la propage à la méthode qui appelle la méthode `distribue`.

➩ choisir judicieusement l'endroit où l'erreur est traitée.

Déclarer et bien Documenter les exceptions

- une méthode peut lever plusieurs exceptions
  ➡ on peut être tenté de les regrouper dans une classe exception mère (mais on perd un peu d'information)
  ➡ on peut déclarer chaque exception individuellement
  (N.B. le langage ne requiert pas la déclaration que chaque exception que le code peut lever)

➡ permet au développeur d'être plus attentif sur certains aspects

- **@throws** balise `javadoc` permet de documenter chaque exception
- Depuis `Java`7, on peut gérer plusieurs exceptions dans une même clause catch

```
catch(ExceptionType1 | ExceptionType2 | ExceptionType 3 e){
  ...
}
```

N’ignorez pas les exceptions

- si une méthode peut lever une exception
  ➡ le développeur de la méthode n’a pas fait ce choix sans raison
  ➡ ne l’ignorez pas !
- il est facile d’ignorer une exception sans rien faire.
- au minimum, écrivez un message indiquant pourquoi vous vous êtes permis d’ignorer l’exception.

Pour les unchecked exceptions

- on peut utiliser la sortie erreur (différente de la sortie standard) pour afficher des message
- La méthode `printStackTrace()` de la classe `Throwable` donne le parcours complet de l'exception du moment où elle a été levée jusqu'à celui où elle a été capturée.

Exemple

```java
int d=10,t1=5,t2=5;
try{
  System.out.println("vitesse:" + d / (t2-t1));
}
catch(ArithmeticException e){
   System.out.println(" vitesse non valide ");
}
catch(Exception e){
   e.printStackTrace();
}
```

Créer sa propre exception

- La classe `MyException` hérite de la classe `Exception`.
- Une méthode qui risque lever une exception de type `MyException` l'indique à l'aide de **throws**

```
1  public class PotionMagiqueException extends Exception {
2     public PotionMagiqueException(){
3       super();
4     }
5     public PotionMagiqueException(String s){
6       super(s);
7     }
8  }
9
10 public class GourdePotionMagique {
11    private int quantite, gorgee=2,contenance=20;
12    public GourdePotionMagique(){ quantite=0;}
13
14    public boolean bois() throws PotionMagiqueException {
15      if (quantite-gorgee <0)
16        throw new PotionMagiqueException
17                    (" pas assez de potion magique!");
18    }
19 }
```

Entrée et sortie

Entrée / sortie

Entrée/sortie : échange de données entre le programme et une source :

- entrée : au clavier, lecture d'un fichier, communication réseau
- sortie : sur la console, écriture d'un fichier, envoi sur le réseau

➾ `Java` utilise des flux (stream en anglais) pour abstraire toutes ses opérations.

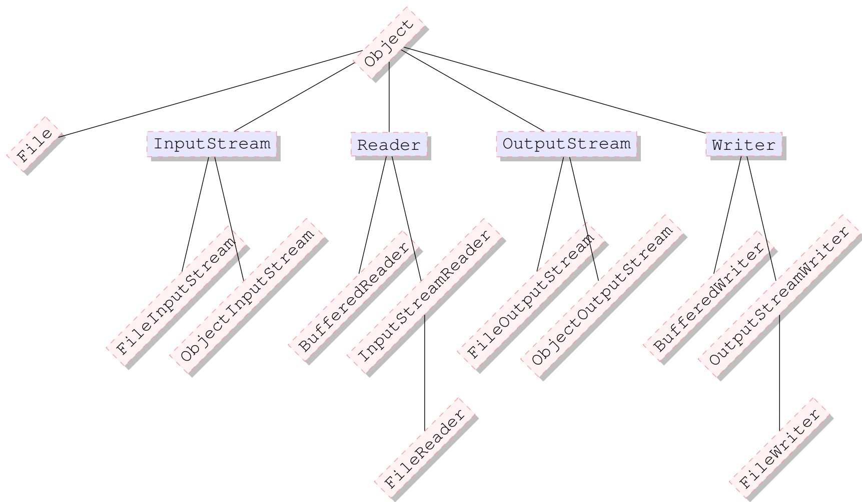
de manière générale, on observera trois phases :

1- ouverture du flux

2- lecture/écriture du flux

3- fermeture du flux

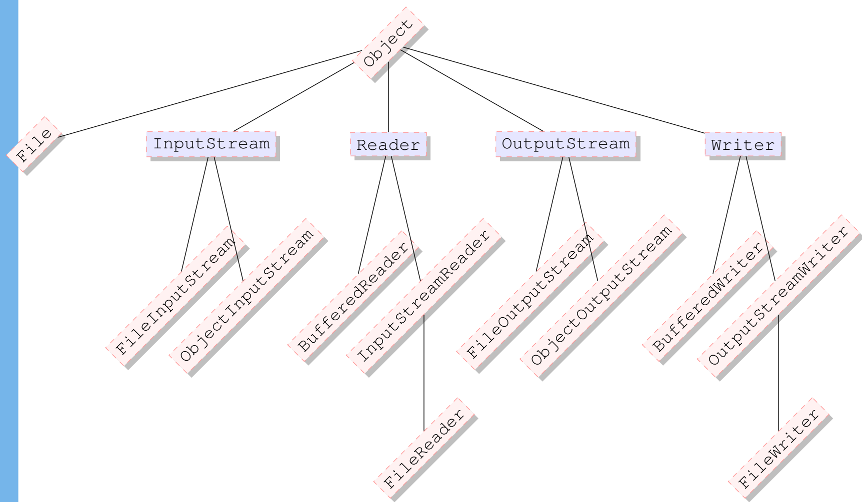
le package `java.io` (fragment)



La classe `Files`

La classe `Files` permet de manipuler des fichiers et répertoire

- nom, chemin absolu, répertoire parent
- s'il existe un fichier d'un nom donné en paramètre
- droit : l'utilisateur a-t-il le droit de lire ou d'écrire dans le fichier
- la nature de l'objet (fichier, répertoire)
- la taille du fichier
- obtenir la liste des fichiers
- effacer un fichier
- créer un répertoire
- accéder au fichier pour le lire ou l'écrire


Object
File
InputStream
Reader
OutputStream
Writer
FileInputStream
ObjectInputStream
BufferedReader
InputStreamReader
FileOutputStream
ObjectOutputStream
BufferedWriter
OutputStreamWriter
FileReader
FileWriter

Flux

Les flux transportent des `bytes` ou des `char`.

Direction du Flux :

- objets qui gèrent des flux d'entrée : **`in`**
  - ➩ **`In`**`putStream`, `File`**`In`**`putStream`, `File`**`In`**`putStream`
- objets qui gèrent des flux de sortie : **`out`**
  - ➩ **`Out`**`putStream`, `File`**`Out`**`putStream`, `File`**`Out`**`putStream`

Source du flux :

- **fichiers** : on pourra avoir des flux vers ou à partir de fichiers
  - ➩ **`File`**`InputStream` et **`File`**`OutputStream`
- **objets** : on pourra envoyer/recevoir un objet via un flux
  - ➩ **`Object`**`InputStream` et **`Object`**`OutputStream`

Obtenir un flux de Bytes

Obtenir un flux : le plus facile, avec une méthode **static**.

- flux de ou vers un **fichier**

```
InputStream in = Files.newInputStream(path);
OutputStream out = Files.newOutputStream(path);
```

  path est un chemin sur votre ordinateur.

```
Path p = Paths.get("/", "home", "stephane");
```

  La méthode get de la classe Paths va former un chemin avec les chaînes passées en argument (et va séparer avec le séparateur pour les répertoire) et va donner un objet de type Path.
  Il y a des choses pour regarder les chemins relatifs ou absolus.

- flux de ou vers **internet**

```
URL url = new URL("http://www.lamsade.dauphine.fr");
InputStream in = url.openStream();
```

- venant d'un tableau de **byte**

```
byte[] bytes = ...
InputStream in = new ByteArayInputStream(bytes);
```

Lire un flux de bytes

- `read()` lit un seul byte ! (retourne -1 si la fin de l'output n'a pas été atteinte)
- `readAllBytes()` lit toutes les bytes possibles et les place dans un tableau

```
byte[] tab = in.readAllBytes();
```

  Pour un fichier, on a directement

```
byte[] tab = Files.readAllBytes(path);
```

- lire quelques bytes `read(byte[], int, int)` ou `readNBytes(byte[],int, int)` `readNbbytes` va attendre d'avoir lu *n* bytes, alors que `read` va retourner moins de bytes s'il échoue

Ecrire un flux de bytes

- écrire un seul byte

```java
int b = 17;
out.write(b);
```

- écrire tous les éléments d'un tableau de bytes

```java
byte[] tab = ...
out.write(tab);
out.write(tab, start, length);
```

- il existe pas mal de méthodes utilitaires : par exemple, transférer depuis un fichier

```java
Files.copy(path, out);
```

Un mot sur l'encodage

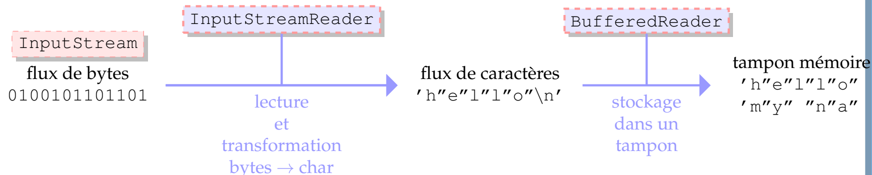
comment les caractères sont encodés en bytes ?

- java utilise le standard Unicode `UTF`-8, `UTF`-16 encode les "points code" en bit.
  `Java`utilise `UTF`-16 qui utilise un ou deux mots de 16bit chacun.
- quelques interprétations différentes du standard. Quelques programmes (dont MS NotePad) ajoute un byte en début d'un fichier pour aider à décoder, mais `Java`ne le prend pas en compte...
- il n'y a pas de moyen fiable de reconnaître un encodage
  ⇒essayer de bien spécifier l'encodate (ex : `Content-Type` dans le header d'une page web)
- classe `Charset`

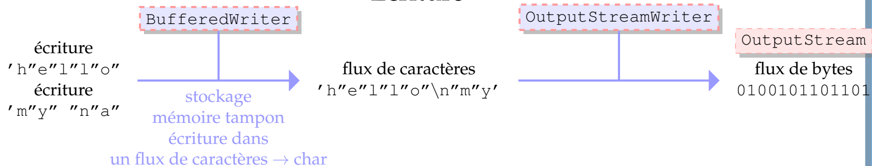
```
String s = new String(bytes, StandardCharSets.UTF_8);
```

Processus de lecture et d'écriture

Lecture



Ecriture



Selon le type de la source ou de la destination (fichier, objet), on utilisera

- `FileReader` à la place de `InputStreamReader`
- `FileOutputStream` ou `ObjectOutputStream` comme implémentation de la classe abstraite `OutputStream`

Lire du Texte

```java
Reader reader = new InputStreamReader(inStream, charset);
```

- lire les symboles un par un

```java
int s = in.read();
```

  pas très utilisé en pratique

- lire tout un texte

```java
String contenu = new String(Files.readAllBytes(path), charset);
```

- récupérer une séquence de ligne

```java
List<String> lines = Files.readAllLines(path,charset);
```

  N.B. On expliquera ce List<String> bientôt, c'est juste une liste contenant des chaînes de caractères.

Lire du Texte

Pour lire des nombres, utilisez la classe `Scanner`

```
Scanner scan = new Scanner(path,"UTF-8");
double value = scan.nextDouble();
```

Pour lire du texte petit à petit, on peut utiliser un `BufferedReader`. En particulier, la classe possède une méthode `readLine()` pour lire ligne par ligne.

Ecrire du Texte

On utilise un `Writer`

```
OutputStream out = ...
Writer write = new OutputStreamWriter(out, charset);
out.write("We all live in a yellow submarine");
```

- pour écrire dans un fichier

```
Writer write = Files.newBufferedWriter(path,charset);
```

- Il est plus pratique d'utiliser `PrintWriter` car il contient des méthodes familières

  `print,println,printf`

  - pour écrire dans un fichier

```
PrintWriter writer =
    new PrintWriter(Files.newBufferedWriter(path,charset));
```

  - pour écrire dans un autre flux

```
PrintWriter writer =
    new PrintWriter(new OutputStreamWriter(out,charset));
```

Expressions régulières

On peut utiliser des expressions régulières pour lire ou chercher du texte.

ici, on donne un seul exemple pour découper du texte avec split, une méthode de la classe `String`.

```
String line = ...
 String[] parts = line.split(";”)
```

cette line va mettre dans un tableau de chaîne de caractères les morceaux délimités par `;`.

Envoi d'objets

But : envoyer toute l'information d'un objet

⇨ mécanisme de « sérialisation »

- la classe doit implémenter l'interface `Serializable`
- l'interface `Serializable` **n**'a **pas** de méthodes : c'est juste un marqueur.
- `Java` transforme l'objet automatiquement en un code pas lisible pour les humains

NB : Si un attribut de la classe est un objet d'une classe `MaClasse`

- `MaClasse` est « sérialisable » : ✔
- `MaClasse` n'est pas « sérialisable » : on peut utiliser le mot-clé **`transient`** pour indiquer de ne pas enregistrer cet attribut

Exemple

```
IrreductibleGaulois panoramix =
        new IrreductibleGaulois("Panoramix", 1.75);

ObjectOutputStream oos =
        new ObjectOutputStream(
          new FileOutputStream(
            new File("panoramix.txt")));

oos.writeObject(panoramix);
oos.close();

ObjectInputStream ois =
        new ObjectInputStream(
          new FileInputStream(
            new File("panoramix.txt")));

IrreductibleGaulois copyPanoramix =
        (IrreductibleGaulois) ois.readObject();
System.out.println(copyPanoramix.nom);
ois.close();
```

N.B. Le code n'est pas correct (gestion des exceptions)

Lire depuis la console, afficher sur la console

- `System.in` :
  - entrée « standard »
  - objet de type `InputStream`
- `System.out` :
  - sortie « standard »
  - objet de type `PrintStream` qui hérite de `OutputStream`

La classe `Scanner` permet de récupérer ce que vous tapez

```java
1 Scanner scan = new Scanner(System.in);
2 int n = scan.nextInt();
3 double x = scan.nextDouble();
4 String s = scan.nextLine();
```

Exceptions et entrée/sortie

```java
try {
  FileInputStream fis = new FileInputStream(new File("test.txt"));
  byte[] buf = new byte[8];
  int nbRead = fis.read(buf);
  System.out.println("nb bytes read: " + nbRead);
  for (int i=0;i<8;i++)
    System.out.println(Byte.toString(buf[i]));
  fis.close();

  BufferedReader reader =
      new BufferedReader(new FileReader(new File("test.txt")));
  String line = reader.readLine();
  while (line!= null){
    System.out.println(line);
    line = reader.readLine();
  }
  reader.close();
} catch (FileNotFoundException e) {
  e.printStackTrace();
}
catch(IOException e){
  e.printStackTrace();
}
```