NFP121 — Programmation Avancée — Exceptions

Xavier Crégut
<Prénom.Nom@enseeiht.fr>

Département Télécommunications & Réseaux
ENSEEIHT

Motivation

- Soit la classe Fraction dont un extrait est :

```
public class Fraction {
   private int numerateur;
   private int denominateur;
   public Fraction(int n, int d)    { ... }
   public Fraction inverse()        { ... }
   ...
}
```

- Peut-on toujours calculer l'inverse d'une fraction ?
  - Il faut que la fraction soit non nulle !
- Comment le sait on (en général) ?
  - en lisant la documentation de inverse !
- Comment ceci est traduit au niveau de la programmation ?
  1. Programmation par contrat **MAIS**
     - peut-on faire confiance à l'appelant ?
     - pas en Java sur les méthode publiques (chargement dynamique des classes)
  2. Programmation défensive :
     - prévoir explicitement les cas anormaux dans le code de la méthode
     - que faire dans un cas anormal (ex : la fraction est nulle) ?

Programmation défensive

Principe : Les cas anormaux sont testés dans le sous-programme.

```
public Fraction inverse() {
   if (this.estNulle()) {
      // Que faire ?
   }
   ...
}
```

Traitements possibles :

- signaler un message d'erreur et continuer l'exécution ;
- signaler un message d'erreur et arrêter l'exécution ;
- renvoyer un code d'erreur ;
- particulariser une valeur de retour pour indiquer l'erreur ;
- réaliser un traitement par défaut (renvoyer une valeur valide) ;

Questions :

- Qui détecte l'erreur (ou le problème) ?
- Qui est capable de le résoudre ?

Meilleure solution : Signaler l'erreur en levant une **exception**.

Motivation

Exemple introductif

Exception

Spécification des exceptions

Définition d’une exception

Une exception est une classe

Conclusion

Exemple introductif

```java
import java.io.*; // io pour input/output
public class Moyenne {
   /** Afficher la moyenne des valeurs réelles du fichier args[0] */
   public static void main (String args []) {
      try {
         BufferedReader in = new BufferedReader(new FileReader(args[0]));
         int nb = 0;    // nb de valeurs lues
         double somme = 0; // somme des valeurs lues
         String ligne;  // une ligne du fichier
         while ((ligne = in.readLine()) != null) {
            somme += Double.parseDouble(ligne);
            nb++;
         }
         in.close();
         System.out.println("Moyenne = " + (somme / nb));
      } catch (IOException e) {
         System.out.println("Problème d’E/S : " + e);
         e.printStackTrace();
      } catch (NumberFormatException e) {
         System.out.println("Une donnée non numérique : " + e);
      }
   }
}
```

Exemples d'utilisation de Moyenne

```
> java Moyenne Données1  # exécution nominale (10 15 20)
Moyenne = 15

> java Moyenne Données2  # (10 15 20 quinze)
Une donnée non numérique : java.lang.NumberFormatException: quinze

> java Moyenne          # pas de fichier en argument
Exception in thread "main"java.lang.ArrayIndexOutOfBoundsException
      at Moyenne.main(Moyenne.java:6)

> java Moyenne Données4  # fichier contenant des lignes vides
Une donnée non numérique : java.lang.NumberFormatException: empty String

> java Moyenne Données5  # fichier inexistant
Problème d'E/S : java.io.FileNotFoundException: Données5 (Aucun fichier
ou répertoire de ce type)
java.io.FileNotFoundException: Données5 (No such file or directory)
      at java.io.FileInputStream.open(Native Method)
      at java.io.FileInputStream.<init>(FileInputStream.java:106)
      at java.io.FileInputStream.<init>(FileInputStream.java:66)
      at java.io.FileReader.<init>(FileReader.java:41)
      at Moyenne.main(Moyenne.java:6)

> java Moyenne Données3  # pas de valeur dans le fichier
Moyenne = NaN
```

Motivation

Exemple introductif

Exception
- Intérêt
- Principe
- Hiérarchie des exceptions
- Lever une exception
- Récupérer une exception
- Traiter une exception
- finally
- Tests unitaires
- Documentation

Spécification des exceptions

Définition d'une exception

Une exception est une classe

Intérêt des exceptions

Les exceptions permettent de :

- transférer le flot de contrôle :
  - de la partie du programme qui détecte le problème (lève l'exception)
  - vers la partie du programme capable de la traiter (récupère l'exception) ;
- éviter de surcharger le code d'une méthode avec de nombreux tests concernant des cas anormaux ;
- regrouper le traitement des cas anormaux et erreurs ;
- différencier les anomalies (différents types d'exception) ;

Attention : Ne pas abuser des exceptions et les réserver aux cas réellement anormaux ou d'erreurs.

Principe

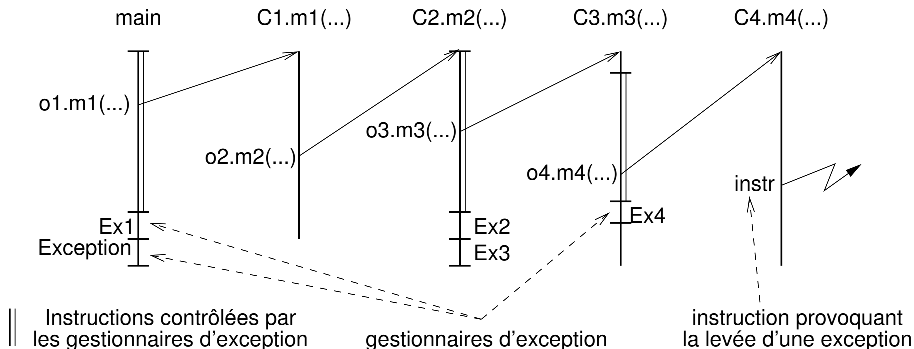
Motivation : Mécanisme pour le traitement des erreurs et/ou des cas anormaux.

Principe : Le mécanisme repose sur trois phases :

- une exception est levée quand une erreur ou anomalie est détectée ;
- l'exception est propagée : l'exécution séquentielle du programme est interrompue et le flot de contrôle est transféré aux gestionnaires d'exception ;
- L'exception est (éventuellement) récupérée par un gestionnaire d'exception. Elle est traitée et l'exécution « normale »reprend avec les instructions qui suivent le gestionnaire d'exception.

Remarque : Une exception non récupérée provoque l'arrêt du programme (avec affichage de la trace des appels de méthodes depuis l'instruction qui a levé l'exception jusqu'à l'instruction appelante de la méthode principale).

Mécanisme de propagation d'une exception



Exercice 1 Indiquer la suite de l'exécution de ce programme lorsque `instr` lève Ex4, Ex3, Ex1, Ex5 et `Err`.

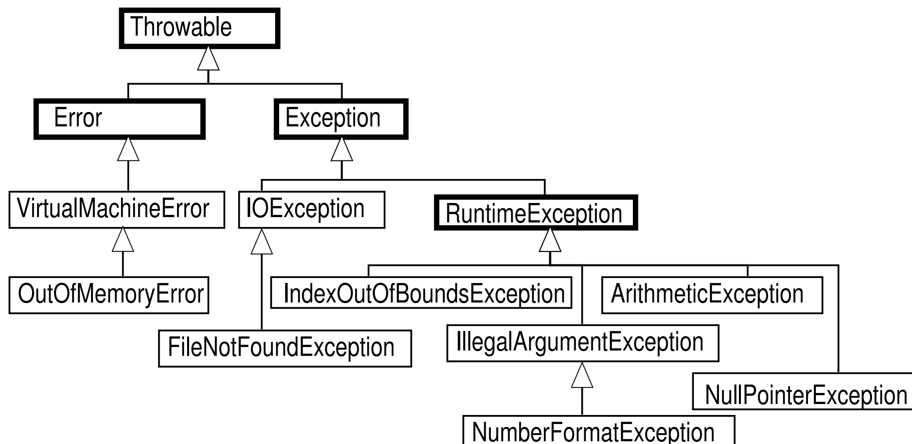
Illustration du mécanisme de propagation

```java
1   /** Illustrer le mécanisme de propagation des exceptions */
2   public class ExceptionPropagation {
3      public static void m1() {
4         m2();
5      }
6      public static void m2() {
7         m3();
8      }
9      public static void m3() {
10        throw new RuntimeException("ERREUR␣!");
11     }
12     public static void main(String[] args) {
13         m1();
14     }
15  }
```

```
Exception in thread "main" java.lang.RuntimeException: ERREUR !
      at ExceptionPropagation.m3(ExceptionPropagation.java:10)
      at ExceptionPropagation.m2(ExceptionPropagation.java:7)
      at ExceptionPropagation.m1(ExceptionPropagation.java:4)
      at ExceptionPropagation.main(ExceptionPropagation.java:13)
```

Hiérarchie des exceptions en Java

Principe : Toutes les exceptions en Java héritent de la classe Throwable.



Classification des exceptions

Les exceptions sont forcément descendantes de la classe `Throwable`.
En Java, les exceptions sont classées en deux catégories :

- les **exceptions hors contrôle** du compilateur :
  - classes descendantes de `java.lang.Error` : ce sont des erreurs non accessibles, qui ne peuvent généralement pas être récupérées (`OutOfMemoryError`, `AssertionError`, `NoClassDefFoundError`...) ;
  - classes descendantes de `java.lang.RuntimeException` : ce sont des erreurs de programmation ; elles ne devraient donc pas se produire (`NullPointerException`, `IndexOutOfBoundsException`...) ;
- les **exceptions sous contrôle** du compilateur. Ce sont les classes qui ne sont sous-types ni de `java.lang.Error` ni `java.lang.RuntimeException`.
  Exemples : `java.io.IOException`, `Exception` ou `Throwable`.
  Le compilateur veut être sûr que le programmeur en a tenu compte.
  Elles correspondent à la notion de *robustesse*.

La classe `java.lang.Throwable`

- `Throwable(String message)` : constructeur avec un message expliquant la cause de l'exception ;
- `Throwable()` : constructeur par défaut (message == **`null`**) ;
- `Throwable(Throwable cause)` : constructeur avec cause ;
- `Throwable(String message, Throwable cause)` ;
- `printStackTrace()` : afficher la trace des appels de méthodes ;
- `getMessage()` : le message passé en paramètre du constructeur ;
- `getCause()` : la cause de l'exception ;
- ...

Les classes `java.lang.Exception` et `java.lang.Error`

- `Exception` et `Error` n'ajoutent aucune nouvelle caractéristique. Elles permettent simplement de classifier les anomalies.
- Elles définissent des constructeurs de même signature que `Throwable`.

Lever une exception

L'opérateur **throw** permet de lever une exception. Une exception est une instance d'une classe descendant de Throwable (souvent de Exception).

Forme générale :

```
1  if (<condition anormale>) {
2     throw new TypeException(<paramètres effectifs>);
3  }
```

Exemples : En considérant la classe Fraction :

```
1  public Fraction(int num, int dén) {
2     if (dén == 0) {
3        throw new ArithmeticException("Division␣par␣zéro");
4     }
5     ...
6  }
7  public Fraction(Fraction autre) {
8     if (autre == null) {
9        throw new IllegalArgumentException("Poignée␣nulle");
10    }
11    ...
12 }
```

Récupérer une exception

Le bloc **try** {...} peut être suivi de plusieurs gestionnaires d'exception :

```
catch (TypeExc1 e) { // gestionnaire de l'exception TypeExc1
    // instructions à exécuter quand l'exception TypeExc1 s'est produite
} catch (TypeExc2 e) { // e ≈ paramètre formel (peu importe le nom)
    // instructions à exécuter quand l'exception TypeExc2 s'est produite
} catch (TE3 | TE4 e) { // ici, TE3 ou TE4 (depuis Java7)
    // instructions à exécuter quand l'exception TypeExc2 s'est produite
} catch (Exception e) { // toutes les exceptions (utiles au programmeur)
    // instructions à exécuter si une exception se produit
} catch (Throwable e) {
    // Toutes les erreurs et autres ! Utile ?
}
```

- L'ordre des **catch** est important (principe de substitution).
- Après l'exécution des instructions d'un **catch**, l'exécution continue après le dernier **catch** (sauf si une exception est levée).
- **Conseil :** Ne récupérer une erreur que si vous savez comment la traiter (en totalité ou partiellement).

Traiter une exception

On ne récupère une exception que si on sait la traiter (au moins partiellement).

Ce traitement est fait dans les instructions du **catch**. Il peut consister à :

- Réparer le problème et exécuter de nouveau l'opération (cf transparent suivant)
- Rétablir un état cohérent et continuer l'exécution sans recommencer
- Calculer un autre résultat remplaçant celui de la méthode
- Réparer localement le problème et propager l'exception (voir **finally**)

```
catch (TypeException e) {
  faire des choses; // par exemple rétablir la cohérence de l'état
  throw e;        // propager l'exception
  throw new ExcQuiVaBien(e); // OU Chaînage des exceptions (Java ≥ 1.4)
}
```

- Réparer localement le problème et lever une nouvelle exception
- Terminer le programme

Traiter une exception : exemple du réessai

```
1  static public int readMois() {
2     int mois = 0;
3     java.util.Scanner scanner = new java.util.Scanner(System.in);
4     boolean reussi = false;
5     do {
6        try {
7           System.out.print("Mois␣:␣");
8           mois = scanner.nextInt();
9           verifierMois(mois);
10          reussi = true;
11       } catch (java.util.InputMismatchException e) {
12          System.out.println("Il␣faut␣saisir␣un␣entier␣!");
13          scanner.nextLine();
14       } catch (IllegalArgumentException e) {
15          System.out.println("Erreur␣:␣" + e.getMessage());
16       }
17    } while (! reussi);
18    return mois;
19 }
20 static private void verifierMois(int m) {
21    if (m < 1 || m > 12) {
22       throw new IllegalArgumentException(m + "␣n'est␣pas␣dans␣[1..12]");
23    }
24 }
```

Exercice 2 Adapter cet algorithme pour limiter le nombre de réessais à 5.

Exceptions : la clause finally

- Un bloc **finally** peut être mis à la fin d'un bloc **try ... catch**.
- Les instructions du bloc **finally** seront toujours exécutées
  - qu'il y ait ou non une exception levée,
  - qu'elle soit ou non récupérée.
- Exemple :

```
try {
   instructions_qui_peuvent_échouer();
} catch (TypeExc1 e) {
   traiter_TypeExc1();
} catch (TypeExc2 e) {
   traiter_TypeExc2();
} finally {
   instructions_toujours_exécutées();
}
```

- **Intérêt :** Être sûr de libérer une ressource (libérer les ressources associées à un composant graphique (dispose), fermer un fichier (close) – à condition que l'ouverture ait réussi...).
- **Remarque :** Java7 ajoute la notion de AutoCloseable (encore plus pratique !)

Exceptions et tests unitaires

- Tester une méthode, c'est aussi vérifier qu'elle lève bien l'exception attendue dans les cas anormaux.
- **JUnit 3 : Possible mais lourd !**

```
1  public class ExceptionJUnit3Test extends junit.framework.TestCase {
2        public void testerException() {
3              boolean reussi = false;
4              try {
5                    Integer.parseInt("quinze");
6              } catch (NumberFormatException e) {
7                    reussi = true;
8              } catch (Throwable e) {
9              }
10             assertTrue("NumberFormatException␣attendue␣!", reussi);
11       }
12 }
```

- **JUnit 4 : simple grâce à expected**

```
1  import org.junit.*;
2  import static org.junit.Assert.*;
3  public class ExceptionJUnit4Test {
4        @Test(expected=NumberFormatException.class)
5        public void testerException() {
6              Integer.parseInt("quinze");
7        }
8  }
```

Documenter les exceptions : javadoc bien sûr !

L'étiquette javadoc `@throws` signale que la méthode peut lever une exception.

Justification : L'appelant de la méthode connait ainsi les exceptions potentielles

exception = paramètre en sortie... généralement non disponible.

Exemple : Spécification de la méthode `java.util.Collection.remove(Object)`

```
/**
 * Removes a single instance of the specified element from this
 * collection, if it is present (optional operation). More formally,
 * removes an element <tt>e</tt> such that
 * <tt>(o==null ? e==null : o.equals(e))</tt>, if
 * this collection contains one or more such elements. Returns
 * <tt>true</tt> if this collection contained the specified element (or
 * equivalently, if this collection changed as a result of the call).
 *
 * @param o element to be removed from this collection, if present
 * @return <tt>true</tt> if an element was removed as a result of this call
 * @throws ClassCastException if the type of the specified element
 *         is incompatible with this collection (optional)
 * @throws NullPointerException if the specified element is null and this
 *      collection does not permit null elements (optional)
 * @throws UnsupportedOperationException if the <tt>remove</tt> operation
 *      is not supported by this collection
 */
boolean remove(Object o);
```

Motivation

Exemple introductif

Exception

Spécification des exceptions

Définition d'une exception

Une exception est une classe

Conclusion

Spécification des exceptions

- Une exception est un **résultat (exceptionnel)** transmis par une méthode. Aussi, Java permet de **spécifier les exceptions** propagées par une méthode.
- Java n'impose (et le compilateur ne vérifie) que la spécification des exceptions sous contrôle (ni `Error`, ni `RuntimeException`).
- **Syntaxe :** Toutes les exceptions sous contrôle qui sont (levées ou) propagées par une méthode doivent être déclarées en utilisant le mot-clé **`throws`**.

```
<modifieurs> Type maMéthode(Type1 a) throws TypeExc1, TypeExc2 { ... }
```

```
public static double racineCarrée(double x) throws MathException { ... }
```

- Le compilateur vérifie que toutes les exceptions (sous-contrôle) produites par les instructions du code de la méthode sont :
  - soit récupérées et traitées par la méthode (clause **`catch`**) ;
  - soit déclarées comme étant propagées (clause **`throws`**).

Spécification des exceptions : illustration (et intérêt)

```
1  import java.io.FileReader;
2
3  class A {
4     void m1() {
5        FileReader f = new FileReader("info.txt");
6     }
7  }
```

Résultat de la compilation

```
ExceptionSpecificationReader.java:5: error: unreported exception
    FileNotFoundException; must be caught or declared to be thrown
     FileReader f = new FileReader("info.txt");
                  ^
1 error
```

Exceptions et sous-typage

Une méthode peut propager (donc lever) toute exception qui est :

- une descendante de `RuntimeException` ou `Error` (**`throws`** implicite) ;
- une descendante d'une des exceptions listées dans la clause **`throws`**.

Exemple : Voir le transparent précédent.

Sous-typage : L'héritage entre exceptions permet (principe de substitution) :

- de limiter le nombre d'exceptions à déclarer (**`throws`**) ;
- de récupérer dans un même **`catch`** plusieurs exceptions.

Attention : Dans les deux cas, on perd en précision !

Exceptions et redéfinition de méthode

Règle sur la redéfinition de méthode : Une méthode redéfinie ne peut lever que des exceptions qui ont été spécifiées par sa déclaration dans la classe parente.

$\Longrightarrow$ Elle ne peut donc pas lever de nouvelles exceptions (contrôlées).

```java
1  class E1 extends Exception {}
2  class E2 extends E1 {}
3  class E3 extends E2 {}
4  class F1 extends Exception {}
5
6  class A {
7      void m() throws E2 { };
8  }
```

```java
1  class B1 extends A {
2      void m() throws E1 { };
3  }
4  class B2 extends A {
5      void m() throws E3 { };
6  }
7  class B3 extends A {
8      void m() throws F1 { };
9  }
```

```
ExceptionSpecificationRedefinition.java:14: error: m() in B1 cannot override m() in A
   void m() throws E1 { };
       ^
  overridden method does not throw E1
ExceptionSpecificationRedefinition.java:20: error: m() in B3 cannot override m() in A
   void m() throws F1 { };
       ^
  overridden method does not throw F1
2 errors
```

Motivation

Exemple introductif

Exception

Spécification des exceptions

Définition d'une exception

Une exception est une classe

Conclusion

Définition d'une exception : exception utilisateur

- **Exception :** Une exception est tout objet instance d'une classe qui hérite de Throwable. Cependant, une exception utilisateur hérite généralement de Exception ou de l'une de ses descendantes.
- **Conseil :** Choisir soigneusement la classe parente de son exception : l'exception doit-elle être sous contrôle ou hors contrôle du compilateur ?
- **Définition type d'une exception :**

```
public class MathException extends Exception {
   public MathException(String s) {
      super(s);
   }
   public MathException() { // utile ?
   }
}
```

- Cette classe, comme toute autre, peut avoir des attributs et des méthodes.

Exemple d'exception : la racine carrée

```
1  public class RacineCarree {
2    public static double RC(double x) throws MathException {
3      if (x < 0) {
4        throw new MathException("Paramètre␣de␣RC␣strictement␣négatif␣:␣" + x);
5      }
6      return Math.sqrt(x);
7    }
8
9    public static void main (String args []) {
10     try {
11       double d = Console.readDouble("Donnez␣un␣réel␣:␣");
12       System.out.println("RC(" + d + ")␣=␣" + RC(d));
13       for (int i = 0; i < 10; i++) {
14         System.out.println("RC(" + i*i + ")␣=␣" + RC(i*i));
15       }
16     } catch (MathException e) {
17       System.out.println("Anomalie␣:␣" + e);
18     }
19   }
20 }
```

Exercice 3 Peut-on sortir la boucle **for** du **try catch** ?

Exemples d'exceptions sur la classe Fraction

```java
public class DivisionParZeroException extends Exception {
    public DivisionParZeroException(String message) {
        super(message);
    }
}
```

```java
public class Fraction {
        private int numérateur;
        private int dénominateur;
        public Fraction(int num, int dén) throws DivisionParZeroException {
                set(num, dén);
        }
        public void set(int n, int d) throws DivisionParZeroException {
                if (d == 0) {
                        throw new DivisionParZeroException("Dénominateur␣nul");
                }
                ...
        }
        public Fraction inverse() throws DivisionParZeroException {
                return new Fraction(dénominateur, numérateur);
        }
        ...
}
```

Utilisation des exceptions de la classe Fraction

```
1  public class TestFractionExceptions {
2        public static void main (String args []) {
3              try {
4                    // Saisir une fraction
5                    int n = Console.readInt("Numérateur␣:␣");
6                    int d = Console.readInt("Dénominateur␣:␣");
7                    Fraction f = new Fraction(n, d);
8                    System.out.println("f␣=␣" + f);
9                    System.out.println("inverse␣de␣f␣=␣" + f.inverse());
10             }
11             catch (DivisionParZeroException e) {
12                   System.out.println("Le␣dénominateur␣d'une␣fraction␣"
13                                     + "ne␣doit␣pas␣être␣nul");
14                   // Est-ce un message correct ?
15             }
16       }
17 }
```

Exercice 4 Peut-on connaître, dans le gestionnaire d'exception, la ligne du bloc **try** qui est à l'origine de l'exception (cf commentaire) ?

Motivation

Exemple introductif

Exception

Spécification des exceptions

Définition d'une exception

Une exception est une classe

Conclusion

Une exception est une classe

Exercice 5 : Somme d'entiers
L'objectif est de calculer la somme des entiers donnés en argument de la ligne de commande en signalant les arguments incorrects. On indiquera le caractère incorrect (qui n'est donc pas un chiffre) et sa position dans l'argument comme dans les exemples suivants :

```
java Somme 10 15 20
Somme : 45

java Somme 10 15.0 20.0
Caractère interdit : >.< à la position 3 de 15.0
```

5.1 Écrire le programme qui réalise cette somme en signalant les erreurs éventuelles.

5.2 Comment faire pour indiquer le numéro de l'argument incorrect ?

5.3 Comment faire pour indiquer tous les arguments incorrects ?

Le programme calculant la somme des arguments

```
1  /** Programme sommant les entiers en argument de la ligne de commande. */
2  class Somme {
3
4     public static void main (String args []) {
5        try {
6           int somme = 0;
7           for (int i = 0; i < args.length; i++) {
8              somme += Nombres.atoi(args[i]);
9           }
10          System.out.println("Somme␣:␣" + somme);
11       }
12       catch (FormatEntierException e) {
13          System.out.println("Caractère␣interdit␣:␣>"
14             + e.getCaractereErrone()
15             + "<␣à␣la␣position␣" + e.getPositionErreur()
16             + "␣de␣" + e.getChaine());
17       }
18    }
19
20 }
```

L'exception FormatEntierException

```
1  /** Exception indiquant une représentation erronée d'un entier en base 10 */
2  public class FormatEntierException extends Exception {
3     private String chaine; // la chaîne contenant l'entier en base 10
4     private int iErreur; // indice de l'erreur dans chaine
5
6     public FormatEntierException(String chaine, int indice) {
7        super("Caractère␣invalide");
8        this.chaine = chaine;
9        this.iErreur = indice;
10    }
11
12    /** La chaîne contenant l'entier en base 10 */
13    public String getChaine() { return this.chaine; }
14
15    /** Position du premier caractère interdit (1 si c'est le premier) */
16    public int getPositionErreur() { return this.iErreur + 1; }
17
18    /** Premier caractère erroné. */
19    public char getCaractereErrone() { return this.chaine.charAt(this.iErreur);
20
21    public String toString() {
22       return "FormatEntierException:␣Erreur␣dans␣" + this.chaine
23          + "␣à␣la␣position␣" + (this.iErreur+1);
24    }
25 }
```

La méthode convertissant une chaîne en entier

```
1   /** Opérations sur les nombres */
2   public class Nombres {
3
4      /** Conversion de chaîne de caractères en entier naturel.
5        * @param s représentation d'un entier naturel en base 10
6        * @return l'entier correspondant à s
7        * @exception FormatEntierException la chaîne est mal formée
8        */
9      public static int atoi(String s) throws FormatEntierException {
10        int resultat = 0;
11        for (int i = 0; i < s.length(); i++) {
12           char c = s.charAt(i);
13           if (c >= '0' && c <= '9') {
14              resultat = resultat * 10 + (c - '0');
15           } else {
16              throw new FormatEntierException(s, i);
17           }
18        }
19        return resultat;
20     }
21  }
```

Motivation

Exemple introductif

Exception

Spécification des exceptions

Définition d'une exception

Une exception est une classe

Conclusion

Quelques conseils

- Une même méthode ne devrait pas lever et récupérer une exception.
  Contre-exemple : Gestion des interactions avec l'extérieur du programme.
- La gestion des exceptions n'est pas supposée remplacer un test simple.
- Ne pas faire une gestion ultrafine des exceptions : multiplier les blocs **`try`** pénalise le programme en terme de performance.
- Éviter de récupérer les exceptions de type `RuntimeException` :
  il est souvent difficile d'identifier leur origine (ex : `NullPointerException`)
- Ne pas museler les exceptions.

  ```
  try { beaucoup de code }
  catch (Exception e) {}
  ```

  Le compilateur signale les exceptions sous contrôle non prises en compte pour vous aider ! Les cacher ne résout pas le problème !
- Éviter d'imbriquer les blocs **`try`** (faire des méthodes auxiliaires).
- Ne pas avoir honte de propager une exception.
  Si vous n'êtes pas capable de la traiter complètement, il est nécessaire de la propager vers l'appelant.