Technologie Objet

—

Exceptions

Xavier Crégut
<Prénom.Nom@enseeiht.fr>

Département Sciences du Numérique
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()        { ... }
   private set(int n, int d)        { ... } // normalise la fraction
   ...
}
```

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 offensive (précondition) **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()) { // détection d'un cas anormal (anomalie)
      // Que faire ?
   }
   ...
```

Mais que fait-on quand une anomalie est détectée ?

Problème posé : Lorsqu'une méthode **C** (cliente) appelle une méthode **F** (fournisseur), il faut aussi tenir compte, dans le cas d'une anomalie, de :

- **D** : Celui qui détecte l'anomalie (ou le problème)
- **T** : Celui qui la traite (la résout).

Exemple : On s'intéresse à une calculatrice sur des fractions

- F est la fonction `inverse` de la classe Fraction
- C est la méthode `inverser` de la calculette qui appelle « `inverse` »
- T est la calculette qui appelle « `inverser` »
- D pourrait être la méthode « `set` » de `Fraction` qui est appelée par « `inverse` »

Calculette $\longrightarrow$ inverser $\longrightarrow$ inverse $\longrightarrow$ Fraction $\longrightarrow$ set

Que faire quand une anomalie est détectée ?

- S_1 Afficher un message d'erreur et arrêter l'exécution
  - Le programme s'est arrêté ! Pas de reprise possible !
  - Programme non robuste ! L'utilisateur final appelle le service après vente !
- S_2 Afficher un message d'erreur et continuer l'exécution
  - Qui le lira ? Préférer la journalisation (logging) !
  - L'exécution peut-elle vraiment continuer ? Possible si couplé avec un traitement par défaut (S_3)
- S_3 Réaliser un traitement par défaut
  - est-ce que le traitement convient pour tous les appelants ?
- S_4 Renvoyer un code d'erreur
  - Pas toujours facile de particulariser une valeur dans le cas d'une requête
  - Les constructeurs n'ont pas de valeur de retour !
  - Aucune garantie que l'appelant tiendra compte du code d'erreur
  - Le code d'erreur devra être transmis de D vers T
- S_5 Signaler le problème en levant une exception
  - L'exception ressemble à une valeur de retour qui n'existe qu'en cas d'anomalie
  - L'exception est créée par D, se propage automatiquement jusqu'à T qui la récupère et l'utilise
  - L'appelant ne peut pas invonlontairement ignorer (et cacher) l'exception
  - Une partie du programme devra la traiter (robustesse) !
  - Tout ceci est détaillé dans la suite de ce cours !

Comparaison des stratégies

- Dans les cas S_1, S_2 et S_3 : T est D
  La manière de traiter l'anomalie est choisie par D et est imposée à tous les C.
- Dans les cas S_4 et S_5 : il doit y avoir communication de D vers T !

Critère	S_1	S_2	S_3	S_4	S_5
T ne connait pas D	oui	oui	oui	oui	oui
C sait si F a fait son travail normalement ou pas	non	non	non	oui	oui
T peut choisir le traitement	non	non	non	oui	oui
Pas d'actions spécifiques entre D et T	—	—	—	non	oui
Anomalie non involontairement ignorée	oui	oui	oui	non	oui

Lever une exception

L'opérateur **throw** lève objet exception :

```
throw new ClasseException(<paramètres effectifs>);
```

Exemple :

```
99   public Fraction inverse() {
100     if (this.estNulle()) { // détection d'un cas anormal (problème)
101        throw new ArithmeticException("/ by zero");
102           // lever d'une exception de type ArithmeticException
103           // avec le message "/ by zero"
104     }
105
106     return new Fraction(den, num);
107  }
```

Explications :

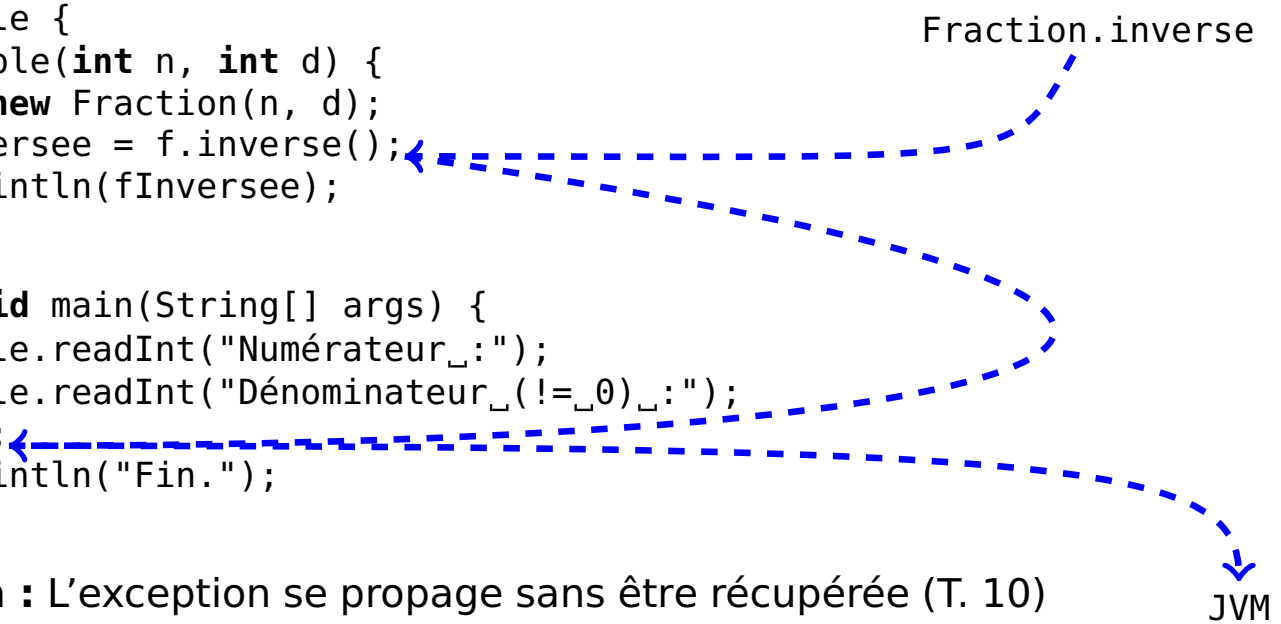
- **new** `ArithmeticException("/ by zero")` crée un objet exception :
  - le paramètre est l'explication de l'anomalie détectée, le message : "/ by zero"
  - la création de l'objet capture la trace des appels
- L'exécution de **throw** lève l'exception précisée.
  - Ceci interrompt l'exécution du code et l'exception se propage (voir T. 8).
  - **throw** est proche de **return** ou **break**.
- Ne pas mettre de **else** et sauter une ligne : le code nominal est ainsi bien visible.

Propagation d'une exception

Propagation : Une fois levée, l'exception remonte (- - >) les blocs et les appels de méthodes

Exemple

```
1   class FractionExemple {
2      static void exemple(int n, int d) {
3         Fraction f = new Fraction(n, d);
4         Fraction fInversee = f.inverse();
5         System.out.println(fInversee);
6      }
7
8      public static void main(String[] args) {
9         int n = Console.readInt("Numérateur : ");
10        int d = Console.readInt("Dénominateur (!= 0) :");
11        exemple(n, d);
12        System.out.println("Fin.");
13  }  }
```



Résultat de l'exécution : L'exception se propage sans être récupérée (T. 10)

```
Numérateur : 0
Dénominateur (!= 0) : 1
Exception in thread "main" java.lang.ArithmeticException: / by zero
      at Fraction.inverse(Fraction.java:101)
      at FractionExemple.exemple(FractionExemple.java:4)
      at FractionExemple.main(FractionExemple.java:11)
```

Analyser l'affichage lié à une exception non récupérée

```
Exception in thread "main" java.lang.ArithmeticException: / by zero
      at Fraction.inverse(Fraction.java:101)
      at FractionExemple.exemple(FractionExemple.java:4)
      at FractionExemple.main(FractionExemple.java:11)
```

1. « Exception in thread "main" » : le **programme n'est pas robuste**.
   Un problème est survenu, et n'a pas été traité par le programme.
2. `java.lang.ArithmeticException` : l'**exception qui signale l'anomalie**.
   le nom est significatif : exception liée à un problème arithmétique
3. « / by zero » : le **message** qui explique pourquoi l'exception est survenue.
4. Les lignes suivantes correspondent aux **appels de méthodes** qui, partant du programme principal (dernière ligne), conduisent à l'instruction qui a levé l'exception (première ligne).

   En lisant la **trace des appels** de bas en haut :
   - la méthode `main` de la classe `FractionExemple` a été exécutée jusqu'à la ligne 11 (fichier FractionExemple.java)
   - où la méthode `exemple` de la classe `FractionExemple` a été appelée et exécutée jusqu'à la ligne 4.
   - où la méthode `inverse` de `Fraction` a été appelée et exécutée jusqu'à la ligne 101 (Fraction.java)
   - où l'exception `ArithmeticException` est levée avec le message « / by zero »

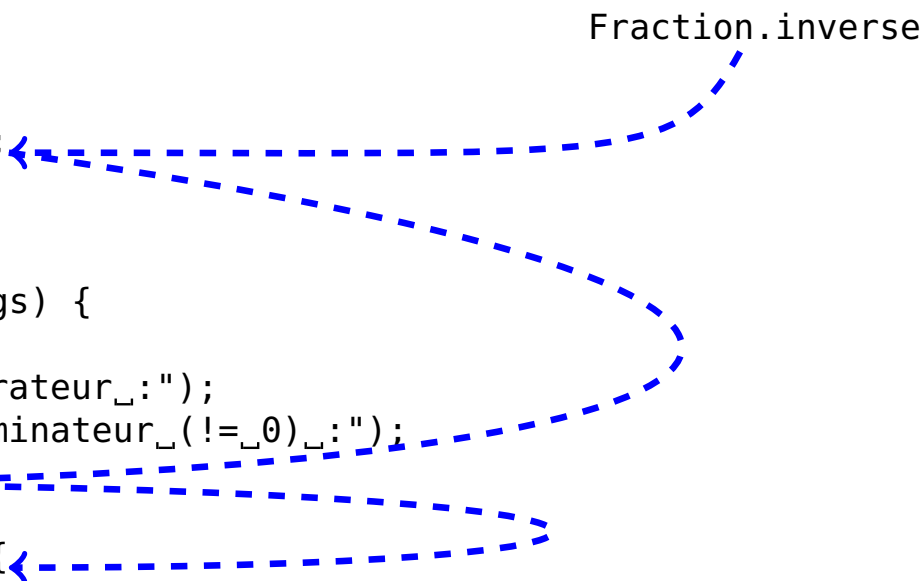
Récupérer une exception

Pour récupérer une exception, il faut :

1. placer les instructions susceptibles de la lever dans un bloc **try**
2. définir un gestionnaire d'exception (**catch**) pour le type d'exception considéré.

Exemple

```
1   class FractionExempleTry {
2      static void exemple(int n, int d) {
3         Fraction f = new Fraction(n, d);
4         Fraction fInversee = f.inverse();
5         System.out.println(fInversee);
6      }
7
8      public static void main(String[] args) {
9         try {
10           int n = Console.readInt("Numérateur : ");
11           int d = Console.readInt("Dénominateur (!= 0) : ");
12           exemple(n, d);
13           System.out.println("Fin.");
14        } catch (ArithmeticException e) {
15           System.out.println("Oups ! Problème arithmétique.");
16           System.out.println("Message : " + e.getMessage());
17           // e.printStackTrace();
18  }  }  }
```



Résultat de l'exécution : L'exception se propage et est récupérée dans le `main`.

```
> java FractionExempleTry
Numérateur : 0
Dénominateur (!= 0) : 1
Oups ! Problème arithmétique.
Message : / by zero
```

Explications :

- le flot de contrôle est transféré de la partie du programme qui détecte l'anomalie (inverse) vers la partie du programme qui sait la traiter (main).
- `getMessage()` : le message associé à l'exception.
- `printStackTrace()` : affiche la trace des appels.

```
java.lang.ArithmeticException: / by zero
   at Fraction.inverse(Fraction.java:101)
   at FractionExempleTry.exemple(FractionExempleTry.java:4)
   at FractionExempleTry.main(FractionExempleTry.java:12)
```

Conseil : Mettre le maximum d'instructions dans le **`try`** pour éviter de morceler le code (et gagner en lisibilité)

Bilan sur le mécanisme d'exception

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

Principe : Le mécanisme repose sur trois phases :

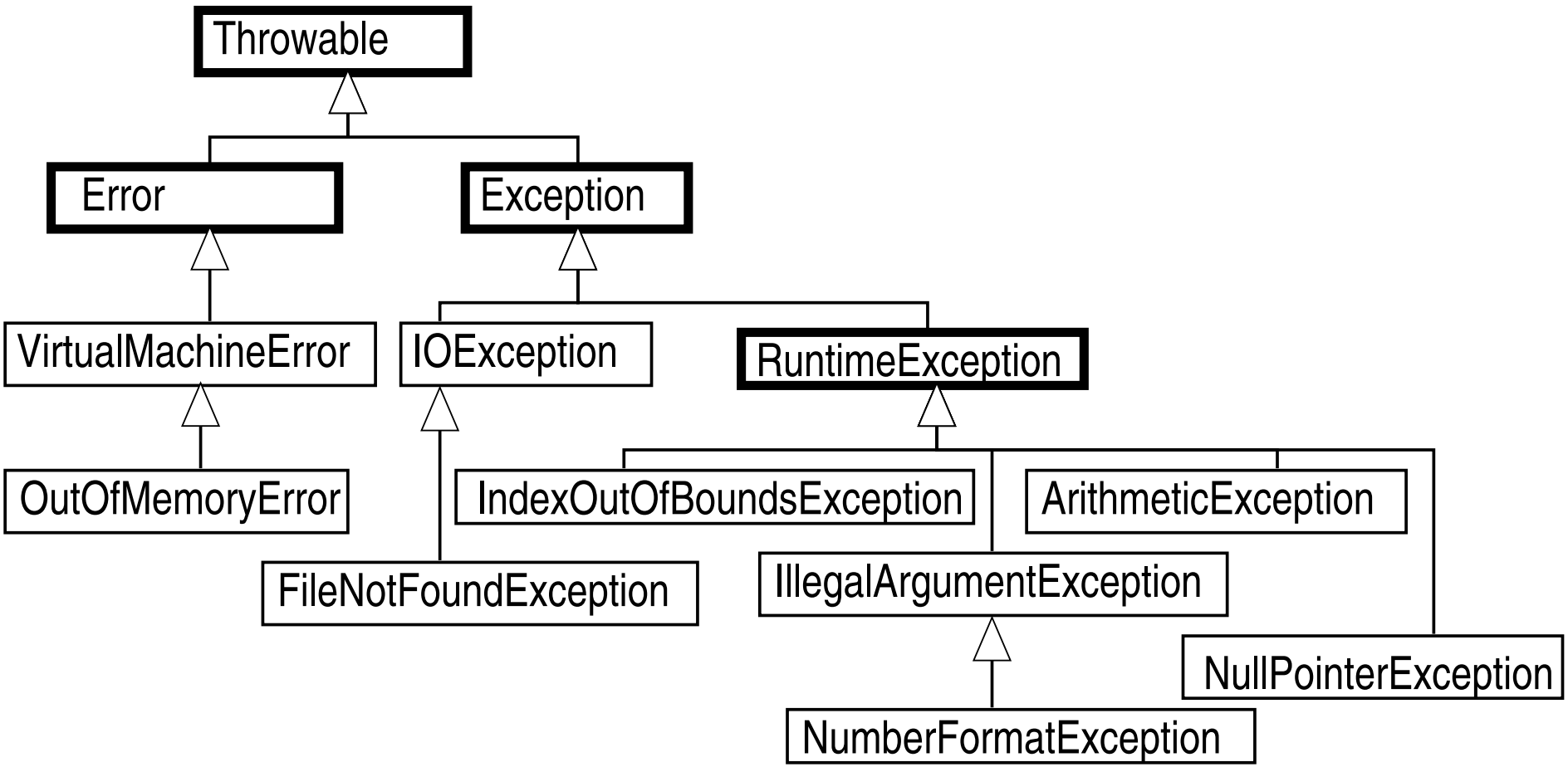
- une exception est **levée** quand un problème ou anomalie est détecté ;
- l'exception est **propagée** : l'exécution séquentielle du programme est interrompue et le flot de contrôle est transféré aux gestionnaires d'exception ;
- L'exception est **récupérée** par un gestionnaire d'exception. L'exécution « normale » reprend avec les instructions qui suivent le gestionnaire d'exception.
  Une exception non récupérée provoque l'arrêt du programme (programme non robuste).

Intérêt : Les exceptions permettent de :

- transférer le flot de contrôle :
  - de la partie du programme qui détecte l'anomalie (lève l'exception)
  - vers la partie du programme capable de la traiter (récupère l'exception) ;
- d'avoir un code plus simple et lisible :
  - code *optimiste* sans test des cas anormaux
  - regroupement des traitements des anomalies (**`catch`**)
  - *It is often easier to ask for forgiveness than to ask for permission.* - Grace Hopper
- différencier les anomalies (différents types d'exception) ;

Hiérarchie des exceptions en Java

Principe : Toute exception en Java doit être un sous-type de la classe Throwable



Classification des exceptions

`Throwable` : L'ancêtre de toute exception.

- `Error`, `Exception`, `RuntimeException` et leurs descendants : **classification** des anomalies
- On ne devrait pas hériter directement de `Throwable` !

`Error` : Anomalies qui ne peuvent pas être traitées pendant l'exécution du programme

- `OutOfMemoryError`, `AssertionError`, `StackOverflowError`, `NoClassDefFoundError`...
- le programme doit être corrigé, lancé avec plus de ressources, etc.

`RuntimeException` : Plutôt des erreurs de programmation (**correction** du programme)

- `NullPointerException`, `IndexOutOfBoundsException`, `ArithmeticException`...
- elles peuvent se produire presque partout !
- Souvent, elles sont anticipées et ne se produisent pas !
- Sauf : `NumberFormatException`...

`Exception` : Anomalies qui concourent à la **robustesse** du programme

- `FileNotFoundException`, `IOException`, etc.
- elles permettent de signaler une anomalie qui normalement ne se produit pas
- elles doivent être traitées par le programmeur (aide fournie pas Java voir T. 24)

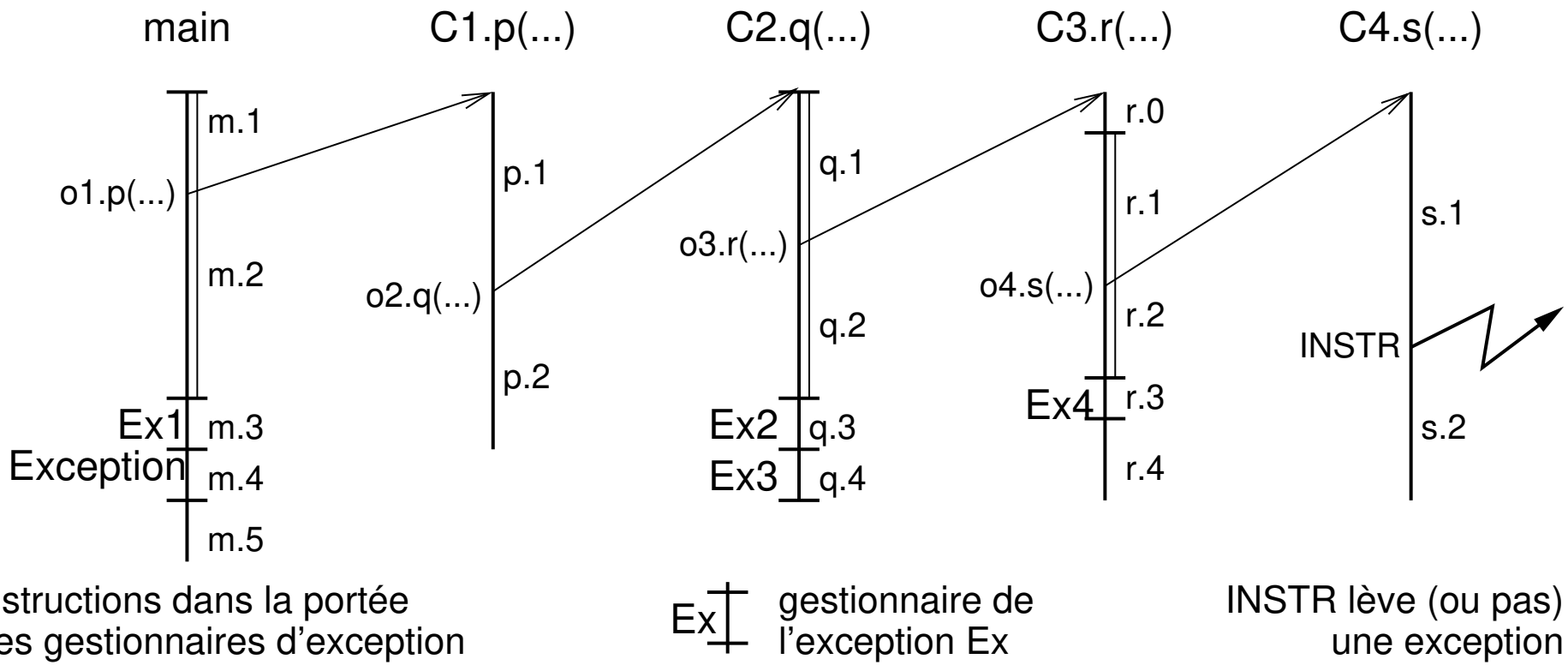
La classe `java.lang.Throwable`

- `Throwable(String message)` : constructeur avec un message sur l'origine de l'exception
- `Throwable()` : constructeur par défaut (message == **`null`**)
- `Throwable(Throwable cause)` : une exception résultant d'une autre exception (sa 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`.
- Présentes pour des raisons de classification (voir T. 15)

Mécanisme de propagation d'une exception



Exercice 1 Indiquer la suite de l'exécution de ce programme lorsque `instr` lève Ex4, Ex3, Ex1, Ex5 ou `Err` (dire les blocs de code exécutés, les blocs sont nommés lettre.chiffre : m.1, m.2...).

Réponses

Voici les réponses. La première ligne correspond au cas où aucune exception ne se produit.

```
--- -> m.1 p.1 q.1 r.0 r.1 s.1 INSTR s.2 r.2 r.4 q.2 p.2 m.2 m.5
Ex4 -> m.1 p.1 q.1 r.0 r.1 s.1 INSTR r.3 r.4 q.2 p.2 m.2 m.5
Ex3 -> m.1 p.1 q.1 r.0 r.1 s.1 INSTR q.4 p.2 m.2 m.5
Ex1 -> m.1 p.1 q.1 r.0 r.1 s.1 INSTR m.3 m.5
Ex5 -> m.1 p.1 q.1 r.0 r.1 s.1 INSTR m.4 m.5
Err -> m.1 p.1 q.1 r.0 r.1 s.1 INSTR Exception in thread "main" Err
      at C4.INSTR(ExceptionComprendre.java:88)
      at C4.s(ExceptionComprendre.java:95)
      at C3.r(ExceptionComprendre.java:70)
      at C2.q(ExceptionComprendre.java:54)
      at C1.p(ExceptionComprendre.java:44)
      at ExceptionComprendre.main(ExceptionComprendre.java:27)
      at ExceptionComprendre.main(ExceptionComprendre.java:21)
```

- Si pas d'exception, tous les blocs sont exécutés.
- L'exécution est la même jusqu'à `instr`. Normal !
- Une exception interrompt bien l'exécution, par exemple s.2 n'est plus exécutée
- Ex5 : Aucun gestionnaire ne correspond mais le nom sonne comme Exception (sous-type)
- Err : Ne sonne pas comme Exception (pas sous-type), non récupérée.

Forme générale du try ... catch ... finally

```
try {
   instructions   // seules les exceptions propagées par ces instructions seront
                  // considérées par les gestionnaires d'exception qui suivent.
} catch (TE1 e) { // gestionnaire de l'exception TE1
   instructions // exécutées quand l'exception se propageant est de type TE1
} catch (TE2 e) { // e ≈ paramètre formel (peu importe le nom)
   // instructions à exécuter quand l'exception est de type TE2
} catch (TE3 | TE4 e) { // ici, TE3 ou TE4 (depuis Java7)
   // instructions à exécuter quand l'exception est de type TE3 ou TE4
} catch (Exception e) { // toutes les exceptions (utiles au programmeur)
   // instructions à exécuter quand l'exception est de type Exception
} catch (Throwable e) {
   // Toutes les erreurs ! Utile ?
} finally {
   // instructions TOUJOURS exécutées.
}
```

- Les **catch** ne sont considérés que si une exception est propagée par une instruction du **try**
- L'ordre des **catch** est important (principe de substitution). Le compilateur le vérifie.
- Après l'exécution des instructions d'un **catch**, l'exécution continue après le dernier **catch** (sauf si une exception est levée dans ce **catch**).
- Si la clause **finally** est présente, ses instructions sont forcément exécutées, qu'il y ait une exception ou non, qu'elle soit récupérées ou non ! **Utile pour libérer une ressource.**
- **Règle :** Ne récupérer une erreur que si on sait la traiter (en totalité ou partiellement).

Compréhension du mécanisme d'exception

```
1  public class ExceptionExemple {
2     static NumberFormatException nfe = new NumberFormatException();
3     static void m(int i) {
4        if (i == 0) throw new IllegalArgumentException();
5        if (i == 1) throw new NullPointerException();
6        if (i == 2) throw new ArithmeticException();
7        if (i == 3) throw new IndexOutOfBoundsException();
8        if (i == 4) throw new StringIndexOutOfBoundsException();
9        if (i < 0) throw nfe;
10       System.out.println("m(int) : OK");
11    }
12    public static void main(String[] args) {
13       try {
14          m(Integer.parseInt(args[0]));
15          if (args[0].equals("5")) return;
16          System.out.println("appel ok");
17       } catch (NullPointerException | ArithmeticException e) {
18          System.out.println(e.getClass().getName());
19       } catch (IndexOutOfBoundsException e) {
20          System.out.println("IndexOutOfBoundsException ?");
21       } finally {
22          System.out.println("F");
23 }  }  }
```

Exercice 2 Expliquer ce qui s'affiche quand on exécute java ExceptionExemple avec comme arguments successifs : 1, 2, 3, 4, 5, 6, 0, -1.

Résultat des différentes exécutions

```
> java ExceptionExemple 1
java.lang.NullPointerException
F
```

```
> java ExceptionExemple 2
java.lang.ArithmeticException
F
```

```
> java ExceptionExemple 3
IndexOutOfBoundsException ?
F
```

```
> java ExceptionExemple 4
IndexOutOfBoundsException ?
F
```

```
> java ExceptionExemple 5
m(int) : OK
F
```

```
> java ExceptionExemple 6
m(int) : OK
appel ok
F
```

```
> java ExceptionExemple 0
F
Exception in thread "main" java.lang.IllegalArgumentException
      at ExceptionExemple.m(ExceptionExemple.java:4)
      at ExceptionExemple.main(ExceptionExemple.java:14)
```

```
> java ExceptionExemple -1
F
Exception in thread "main" java.lang.NumberFormatException
      at ExceptionExemple.<clinit>(ExceptionExemple.java:2)
```

1. Quelle relation entre IndexOutOfBoundsException ou StringIndexOutOfBoundsException ? Voir exécutions 3 et 4.
2. Quand est créée la trace d'appels ? Voir exécutions 0 et -1.

Exceptions et tests unitaires

- Tester une méthode, c'est aussi vérifier qu'elle lève bien l'exception attendue !
- **JUnit 4 : simple grâce à expected**
  - préciser la classe d'exception attendue comme valeur de expected
  - le test réussi si et seulement si l'exception attendue se produit

```
1  public class ExceptionJUnit4Test {
2        @Test(expected=NumberFormatException.class)
3        public void testerException() {
4              Integer.parseInt("quinze");
5  }     }
```

- **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 }     }
```

Documenter les exceptions : javadoc bien sûr !

- Les exceptions doivent être documentées dans la spécification d'une méthode : `@throws`
- *Justification :* L'appelant doit savoir à quelles exceptions s'attendre
- Une exception est équivalent un 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);
```

Spécification des exceptions

Idée :

- Une exception est un **résultat exceptionnel** transmis par une méthode.
- Il devrait donc apparaitre dans la signature de la méthode (comme le type de retour)

Mise en œuvre : spécifier les exceptions propagées par une méthode avec **`throws`**

```
1  public class FileReader {
2     public FileReader(String fileName) throws FileNotFoundException;
3        // le constructeur de FileReader peut donc laisser se propager
4        // l'exception FileNotFoundException
```

- On peut spécifier plusieurs exceptions : **`throws`** `E1, E2, E3`

Intérêt :

- Le compilateur vérifie que le programmeur a bien pris en compte toutes les *exceptions vérifiées* propagées par les instructions du code d'une méthode.
  - **Exception vérifiée :** exception qui est ni `Error`, ni `RuntimeException`
  - **Exception non vérifiée :** sous-type de `Error` ou `RuntimeException`
- Le programmeur doit : (c'est le « **`catch or throws`** `requirements` »)
  - soit récupérer (et traiter) l'exception : **`catch`**
  - soit la spécifier comme se propageant : **`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
```

- L'erreur signale que le programmeur n'a pas dit ce qu'il faisait de `FileNotFoundException`
- Le programmeur doit réfléchir et décider ce qu'il fait !
  - S'il sait traiter l'exception signalée, il ajoutera un **`try ... catch`**.
  - S'il ne sait pas la traiter, il devra dire qu'il la laisse se propager avec **`throws`**.

Exceptions et sous-typage

Une **méthode ne peut propager** (donc lever) que les exceptions qui sont :

- descendantes de `RuntimeException` ou `Error` (exceptions non vérifiées, **`throws`** implicite)
- descendantes d'une des exceptions listées dans la clause **`throws`**

Exemple : Voir le transparent précédent

- `m1` doit spécifier `FileNotFoundException` avec **`throws`** ou la récupérer avec **`catch`**

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 !

- Dans un **`throws`**, spécifier un type plus général cache à l'appelant les exceptions réelles. Il devra envisager tous les descendants même s'ils ne se produisent jamais.
- Dans un **`catch`**, plus le type est général, plus grand sera le nombre d'exceptions récupérées et moins le traitement pourra être précis

Contraintes sur la redéfinition de méthode

Règle : Une méthode redéfinie ne peut propager que des exceptions spécifiées par sa déclaration dans la classe parente.

⇒ Elle ne peut donc pas lever de nouvelles exceptions vérifiées

- La redéfinition peut spécifier qu'elle laisse se propager moins d'exceptions

```
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  }
```

```
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
```

Définition d'une exception

Le programmeur peut **définir ses propres exceptions** :

- Il suffit d'écrire une classe qui est sous-type de `Throwable`
- Ne jamais hériter directement de `Throwable` !
- Comme pour toute classe, on peut définir attriuts, constructeur, méthodes...

Conseil : Choisir soigneusement la classe parente :

- Si on pense que l'appelant peut raisonnablement récupérer l'exception pour résoudre l'anomalie, l'exception doit être définie comme *vérifiée*
- Si l'anomalie traduit un appel incorrect de la méthode, on choisit une exception *non vérifiée* :
  - `RuntimeException` si l'utilisateur peut la récupérer (e.g. `NumberFormatException`)
  - `Error` s'il ne doit pas la récupérer (e.g. `AssertionError`)

Exemple :

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

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 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 positif : ");
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 }  }  }
```

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

Exercice 4 Est-ce une bonne idée ici d'avoir utilisé une exception vérifiée ?

Réponses

Peut-on sortir la boucle `for` du `try catch` ?

- On constate que dans le boucle **`for`** on calcule la racine carrée de i^2 qui est positif
- L'exception `MathException` ne se produira pas
- Mais on ne peut pas sortir du **`for`** car le compilateur n'a pas cette finesse
- Pour lui, l'exception pourrait se produire, elle est vérifiée, il veut un **`throws`** ou un **`catch`**

Est-ce une bonne idée ici d'avoir utilisé une exception vérifiée ?

- Ici, donner un paramètre effectif négatif est plutôt un appel incorrect
- Il serait plus judicieux de choisir une `RuntimeException`, par exemple `IllegalStateException`

Limites des exceptions vérifiées

```
1  public class DivisionParZeroException extends Exception {}
2        // Minimale. Sans constructeur. Pas une bonne idée !
```

```
1  public class Fraction {
2        private int numérateur, dénominateur;
3        public Fraction(int num, int dén) throws DivisionParZeroException {
4              set(num, dén);
5        }
6        public void set(int n, int d) throws DivisionParZeroException {
7              if (d == 0) {
8                    throw new DivisionParZeroException();
9              }
10             ...
11       }
12       public Fraction inverse() throws DivisionParZeroException {
13             return new Fraction(dénominateur, numérateur);
14       }
15       ...
```

- `set` propage l'exception si le dénominateur est nul, d'où le **`throws`**
- Le constructeur appelle `set` et ne sait pas traiter l'exception, d'où le **`throws`**
- `inverse` appelle le constructeur, ne sait pas traiter l'exception, d'où le **`throws`**

⇒ Le **`throws`** devra être mis jusqu'à la méthode qui trairera l'exception

- On marque le chemin de la méthode qui détecte le problème jusqu'à celle qui le traitera !

Utilisation des exceptions de la classe Fraction

```
1  public class ExempleFractionExceptions {
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("Que s'est-il passé ?");
13       }
14    }
15 }
```

Exercice 5 Qu'elles sont les instructions qui peuvent conduire dans le **catch** ?
Comment faire pour afficher un message plus explicite à l'utilisateur ?

Réponses

Qu'elles sont les instructions qui peuvent conduire dans le `catch` ?

- Si on arrive dans le **`catch`**, c'est que l'exception `DivisionParZeroException` s'est produite
- Deux instructions peuvent propager cette exception :
  1. le constructeur de Fraction : `new Fraction(n, d)`
  2. le calcul de l'inverse : `f.inverse()`

Comment faire pour afficher un message plus explicite à l'utilisateur ?

- Pour afficher un message clair, il faut savoir quelle instruction a levé l'exception
- Il nous faut plus d'information :
  - soit la valeur de `d` : si `d == 0`, c'est le constructeur, sinon l'inverse
  - soit la fraction `f` (si on l'initialise à **`null`** avant la création de la fraction)
- Dans les deux cas, il faut sortir les variables du **`try`** : les accolades définissent la portée !
  - On ne peut pas respecter la règle « ne déclarer une variable que quand on sait l'initialiser »
- Il faut les initialiser avant le **`try`** :
  - On fait `f = null;` et ainsi, dans le **`catch`**, `f == null` permet de conclure
  - On fait `d = 0;` mais peu d'importance car sa valeur sera remplacée

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 la même exception
}
```

- Réparer localement le problème et lever une nouvelle exception

```
catch (TypeException e) {
  faire des choses; // par exemple rétablir la cohérence de l'état
  throw new ExceptionQuiVaBien(e); // Chaînage des exceptions (Java4)
}
```

- Terminer le programme

Traiter une exception : exemple du réessai

Saisir un numéro de mois compris entre 1 et 12

```
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 6 Adapter cet algorithme pour limiter le nombre de réessais à 5.

Explications et réponse

Comprendre ce programme :

- Le répéter (**`do ... while`**) s'exécute tant que `reussi` est faux.
- reussi est mis à faux au début et ne passera à vrai qu'à la fin des instruction du **`try`**... si aucune exception n'interrompt son exécution !
- les exceptions peuvent être :
  - `InputMismatchException` levée par `nextInt` : l'utilisateur n'a pas saisi un entier
  - `IllegalStateException` levée par `verfierMois` : la valeur n'est pas entre 1 et 12
- Dans les deux cas, on « **répare** (message à l'utilisateur) **et on exécute à nouveau** »
- dans le cas `InputMismatchException`, on **rétablit un état cohérent** : `readLine()`

Limiter le nombre de réessais à 5

- On ajoute un compteur du nombre d'essais : `nbEssais`
- La condition de continuation sera alors : **`while`** `(! reussi && nbEssais < 5)`
- On initialise `nbEssais` à 0 avant la boucle
- On l'incrémente au tout début du **`try`** :
  - ne pas le faire avant le try pour ne pas morceler le code
  - le faire au début du **`try`** pour être sûr que ce sera fait !
  - si on le fait à la fin du **`try`**, une exception peut empêcher son exécution : nbEssais sera incohérent !
  - ne pas le faire après le **`try catch`** : des exceptions non récupérées (`NoSuchElementException`) !
- En sortie de boucle, on initialise `mois` à 1 (ou autre) si `reussi` vaut faux
- C'est un exemple de « **Calculer un résultat de remplacement** »

Moyenne des valeurs d'un fichier

```
1  import java.io.*;
2
3  class Util {
4     /** Retourner la moyenne des valeurs réelles présentent dans le fichier
5      * nommé nomFichierValeurs, une valeur par ligne.
6      * @param nomFichierValeurs le nom du fichier qui contient les valeurs
7      * @return la moyenne des valeurs de nomFichierValeurs
8      * @throws ... rubrique volontairement non renseignée ...
9      */
10    public static double moyenne(String nomFichierValeurs) throws IOException {
11       BufferedReader in = new BufferedReader(new FileReader(nomFichierValeurs));
12       int nb = 0;    // nb de valeurs lues
13       double somme = 0; // somme des valeurs lues
14       String ligne;  // une ligne du fichier
15       while ((ligne = in.readLine()) != null) { // encore des valeurs dans le fichier
16          // traitement de la valeur ligne
17          somme += Double.parseDouble(ligne);
18          nb++;
19       }
20       in.close();    // fermeture du fichier
21       return somme / nb; // la moyenne
22    }
23 }
```

Exercice 7 Commenter cette classe. Lister les problèmes qu'elle pose.

Réponse

C'est un **code optimiste** qui suppose que tout se passe bien.

- Soit on ajoute de nombreux tests, le code sera moins lisible
- soit on s'occupe des exceptions : on garde le code optimiste et on a un programme robuste

Voici les problèmes et les solutions retenues :

- Le fichier n'existe pas ⇒ à propager (c'est l'appelant qui a fourni le nom du fichier !)
- On n'a pas les droits en lecture sur le fichier ⇒ à propager (idem)
- Le fichier est mal formé : pas une valeur réelle par ligne (`NumberFormatException` signalé par parseDouble) ⇒ lever une nouvelle exception : `FichierMalFormeException`
- Le fichier est vide ⇒ lever une nouvelle exception `PasDeDonneesException`
- La fermeture du fichier ne sera pas faite si une exception se produit
  - c'est un problème car le fichier utilise des ressources du système
  - Cas de **réparer localement et propager l'exception ou lever une nouvelle exception**

Vérifions que la non fermeture du fichier est un problème et voyons comment le résoudre !

Classe principale : ne pas fermer le fichier est un problème

```
1  import java.io.*;
2  public class UtilMain {
3     public static void main(String[] args) throws java.io.IOException {
4        int nbErreurs = 0;
5        for (int i = 0; i < 30000; i++) {
6           // Créer un fichier qui contient 10 xxx i => mal formé !
7           File f = new File("/tmp/data" + i + ".txt");
8           f.deleteOnExit(); // le fichier sera supprimé à la fin du programme
9           PrintWriter pw = new PrintWriter(new FileWriter(f));
10          pw.println("10\nxxx\n" + i);
11          pw.close(); // le fichier a été créé
12          // Calculer la somme des valeurs du sujet
13          try {
14             double somme = Util.moyenne(f.getPath());
15          } catch (Exception e) {
16             nbErreurs++;
17       }  }
18       System.out.println("Nombre␣d'erreurs␣:␣" + nbErreurs);
19 }  }
```

```
1  > java UtilMain
2  Exception in thread "main" java.io.FileNotFoundException: /tmp/data1613.txt (Too many open files)
3     at java.base/java.io.FileOutputStream.open0(Native Method)
4     at java.base/java.io.FileOutputStream.open(FileOutputStream.java:298)
5     at java.base/java.io.FileOutputStream.<init>(FileOutputStream.java:237)
6     at java.base/java.io.FileOutputStream.<init>(FileOutputStream.java:187)
7     at java.base/java.io.FileWriter.<init>(FileWriter.java:96)
8     at UtilMain.main(UtilMain.java:9)
```

Première solution : lourd

```
1  import java.io.*;
2  class Util {
3     public static double moyenne(String nomFichierValeurs) throws IOException {
4        BufferedReader in = null; // null pour savoir si ouvert ou non
5        try {
6           in = new BufferedReader(new FileReader(nomFichierValeurs));
7           int nb = 0;    // nb de valeurs lues
8           double somme = 0; // somme des valeurs lues
9           String ligne;  // une ligne du fichier
10          while ((ligne = in.readLine()) != null) { // encore des valeurs dans le fichier
11             // traitement de la valeur ligne
12             somme += Double.parseDouble(ligne);
13             nb++;
14          }
15          in.close();
16          return somme / nb; // la moyenne
17       } catch (Exception e) {
18          if (in != null) {  // On a pu ouvrir le fichier
19             in.close();  // fermeture du fichier
20          }
21          throw e; // on lève la même exception !
22 }  }  }
```

```
1  > java UtilMain
2  Nombre d'erreurs : 30000
```

Discussion

Pour être sûr de fermer le fichier quand une exception se produit :

- on récupère toutes les exceptions
- on ferme le fichier
- on relève la même exception (donc mêmes informations, même trace d'appels)

On ne doit fermer le fichier que si on a réussi à l'ouvrir :

- il faut avoir accès à la variable `in` ⇒ déclarée avant le **`try`**
- on l'initialise à **`null`** pour déterminer s'il a pu être créé ou non
- dans le **`catch`** on ne fait `close` qui si `in != null`

Ça marche... mais c'est LOURD !

- surtout si on considère que d'autres exceptions doivent être traitées
- par exemple `NumberFormatException` doit être récupérée pour lever `FichierMalFormeException`
- Il faudra penser à faire un close dans ce cas aussi !
- « penser à ... » c'est donc qu'on risque d'oublier... et donc qu'on oubliera !

Solution 2 : finally pour libérer une ressource

```
1  import java.io.*;
2  class Util {
3     public static double moyenne(String nomFichierValeurs) throws IOException {
4        BufferedReader in = null; // null pour savoir si ouvert ou non
5        try {
6           in = new BufferedReader(new FileReader(nomFichierValeurs));
7           int nb = 0;    // nb de valeurs lues
8           double somme = 0; // somme des valeurs lues
9           String ligne;  // une ligne du fichier
10          while ((ligne = in.readLine()) != null) { // encore des valeurs dans le fichier
11             // traitement de la valeur ligne
12             somme += Double.parseDouble(ligne);
13             nb++;
14          }
15          return somme / nb; // la moyenne
16       } finally {
17          if (in != null) {  // On a pu ouvrir le fichier
18             in.close();  // fermeture du fichier
19 }  }  }  }
```

```
1  > java UtilMain
2  Nombre d'erreurs : 30000
```

- **finally** est la bonne solution quand on est sûr de vouloir faire quelque chose
- mais il faut encore gérer le fait que le fichier peut ne pas avoir été ouvert
- et il faut penser à ajouter le **finally** !

Meilleure solution : AutoCloseable (Java7)

```
1  import java.io.*;
2  class Util {
3     public static double moyenne(String nomFichierValeurs) throws IOException {
4
5        try (BufferedReader in = new BufferedReader(new FileReader(nomFichierValeurs))) {
6           int nb = 0;    // nb de valeurs lues
7           double somme = 0; // somme des valeurs lues
8           String ligne;  // une ligne du fichier
9           while ((ligne = in.readLine()) != null) { // encore des valeurs dans le fichier
10             // traitement de la valeur ligne
11             somme += Double.parseDouble(ligne);
12             nb++;
13          }
14          return somme / nb; // la moyenne
15 }  }  }
```

```
1  > java UtilMain
2  Nombre d'erreurs : 30000
```

- Java7 ajoute une interface `AutoCloseable` qui spécifie la méthode `close()`
- Le **`try`** évolue avec des parenthèses entre lesquelles sont déclarées et créés des objets de type `AutoCloseable` (syntaxe proche **`for`**) : *try-with-resources*
- Le compilateur Java ajoute automatiquement l'équivalent du **`finally`** sur ces objets
- On peut mettre plusieurs déclarations de variables dans **`try`** `(...)`, les séparer par « ; »
- **Conclusion :** Toujours utiliser le **`try`** `(...)` quand c'est possible !

Une exception est une classe

Exercice 8 : 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
```

8.1. Écrire le programme qui réalise cette somme en signalant les erreurs éventuelles.
8.2. Comment faire pour indiquer le numéro de l'argument incorrect ?
8.3. 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 : >" + e.getCaractereErrone()
14             + "< à la position " + e.getPositionErreur() + " de " + e.getChaine());
15 }  }  }
```

- On fait la somme de tous les arguments de la ligne de commande
- Chaque argument est traduit en entier
- On s'appuie sur un méthode `atoi` définie sur la classe `Nombres`
  - elle prend en paramètre une chaîne de caractères et retourne un entier
  - si la chaîne ne correspond pas à un entier, elle lève une exception `FormatEntierException`
  - cette exception transporte les informations utiles pour afficher le message d'erreur : `getCaractereErrone`, `getPositionErreur`, `getChaine`

La classe d'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 }
```

- C'est une exception vérifiée (hérite de Exception). C'est juste un choix.
- C'est une classe avec attributs, méthodes, constructeur.
- Les attributs permettent d'implanter les méthodes attendues (voir T. précédent)

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      * @throws 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       return resultat;
19 }  }
```

- On applique le schéma de Horner
- Si un caractère n'est pas un chiffre, alors c'est que la chaîne ne correspond pas
- On lève l'exception `FormatEntierException` :
  - la chaîne en cause est `s`
  - la position de l'erreur est `i`

Quelques conseils

- Une même méthode ne devrait pas lever et récupérer une exception.
  - On peut arriver au même résultat avec les structures de contrôle classiques
  - *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 **`try`** $\Rightarrow$ perte en lisibilité
- 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 vérifiées non prises en compte pour vous aider !
  - Les ignorer ne résout pas le problème !
  - Pour désactiver la vérif. des exceptions, faire dans le **`catch`** : **`throw new`** `RuntimeException(e);`
- Éviter d'imbriquer les blocs **`try`** : faire des méthodes auxiliaires
- Ne pas avoir honte de propager une exception :
  - Si vous ne savez pas la traiter complètement, il faut la propager vers l'appelant
  - Est-ce à vous de choisir le traitement ou à l'appelant ?