Aspects des responsabilités des classes
et de la programmation par contrats en Java

Assertions et Exceptions

Université de Montpellier
Faculté des sciences

HAI401 - Modélisation et programmation par objets

Responsabilités et contrats

Deux points de vue convergent en programmation par objets pour attribuer aux classes des responsabilités :

- Les classes mettent en œuvre des types abstraits de données, décrits formellement par :
  - un nom de type, parfois d'autres types associés
  - des opérations
  - des pré-conditions
  - des axiomes
- Les objets collaborent pour effectuer des traitements. Ces collaborations sont régies par des contrats [1] :
  - des pré-conditions
  - des post-conditions
  - des invariants d'algorithme
  - des invariants de classe

1. Le terme de programmation par contrats a été introduite par B. Meyer, 1985, dans le langage Eiffel

Responsabilités et contrats

En Java, deux mécanismes contribuent à la mise en place de ces responsabilités et de ces contrats par le biais du code (par opposition à des déclarations) :

- Les assertions, plutôt réservées à la mise au point
- Les exceptions, plutôt réservées à la récupération des erreurs à l'exécution

Ils ont leurs limites, mais l'important est de comprendre le rôle qu'ils jouent et leur intérêt.

Assertions

- Les assertions permettent de vérifier des propriétés sur les classes et les algorithmes
- Elles aident à la mise au point (débogage)
- Le programme est interrompu en cas de non respect d'une assertion
- Elles peuvent être activées ou inhibées au niveau des classes ou des paquetages

Syntaxe

```
assert conditionQuiDoitEtreVraie ;
assert conditionQuiDoitEtreVraie : objet ;
```

Assertions

Première syntaxe

L'âge d'une personne est compris entre 0 et 140

```
public class Personne{
  private int age;
  ...
  public void setAge(int a){
        ....
      assert(this.age >=0 && this.age < 140);
  }
}
```

Difficulté de mise en œuvre du procédé de Java : trouver les endroits pertinents du programme pour placer l'assertion.

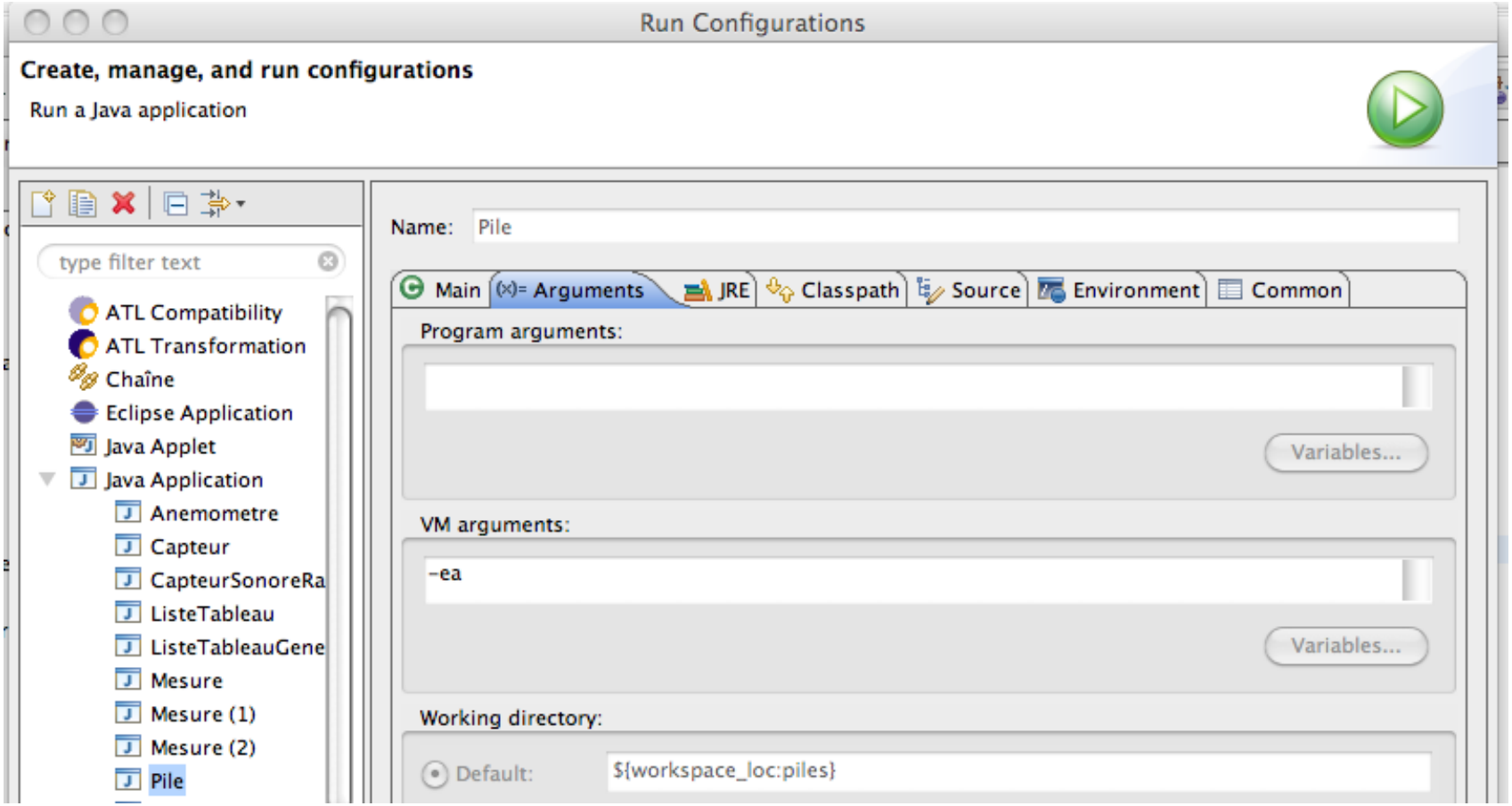
Assertions

Deuxième syntaxe

L'âge d'une personne est compris entre 0 et 140

```
public class Personne{
  private int age;
  ...
  public void setAge(int a){
        ....
        assert(this.age >=0 && this.age < 140)
                                  : "age hors des bornes";
  }
}
```

Assertions : Configuration sous eclipse pour activer les assertions



Commandes : `-ea` (activer) `-da` (désactiver)
L'activation/désactivation peut être générale, ou bien seulement sur une classe ou un paquetage

Assertions

En cas de non respect d'une assertion, l'exécution du programme s'arrête de manière fatale et on obtient un message d'erreur de type :

```
Exception in thread "main" java.lang.AssertionError
        at ...
        at ...
```

Assertions

Bonnes pratiques pour les assertions : contrôle lors du développement et des tests

- invariant d'algorithme
  - à la fin de l'itération `i` dans la recherche du minimum, la variable `min` contient la plus petite valeur rencontrée entre 0 et `i`
- invariant de flux
  - ce point de programme ne peut être atteint
- post-conditions
  - à la fin d'une fonction de tri le tableau est trié
- invariants de classe
  - l'âge d'une personne est compris entre 0 et 140
  - une personne mariée est majeure

Exceptions : classes et objets représentant des erreurs

Les exceptions sont destinées au contrôle des erreurs à l'exécution

- invariant de classe
  - l'âge d'une personne est compris entre 0 et 140
  - une personne mariée est majeure
- préconditions
  - dépiler() seulement si pile non vide
- postcondition remplie (hors erreur de logique)
  - après empiler(a), l'élément est dans la pile (car il y avait assez de place)
- abstraction/encapsulation des exceptions provenant des parties ou des éléments de l'implémentation
  - Point mal formé –> Rectangle mal formé
  - Tableau interne de pile plein –> impossible d'empiler

Il y a un certain recouvrement donc il n'est pas toujours facile de choisir entre assertion ou exception.

Détaillons les exceptions

Exception = Un objet qui représente une *erreur* à l'exécution

due à :

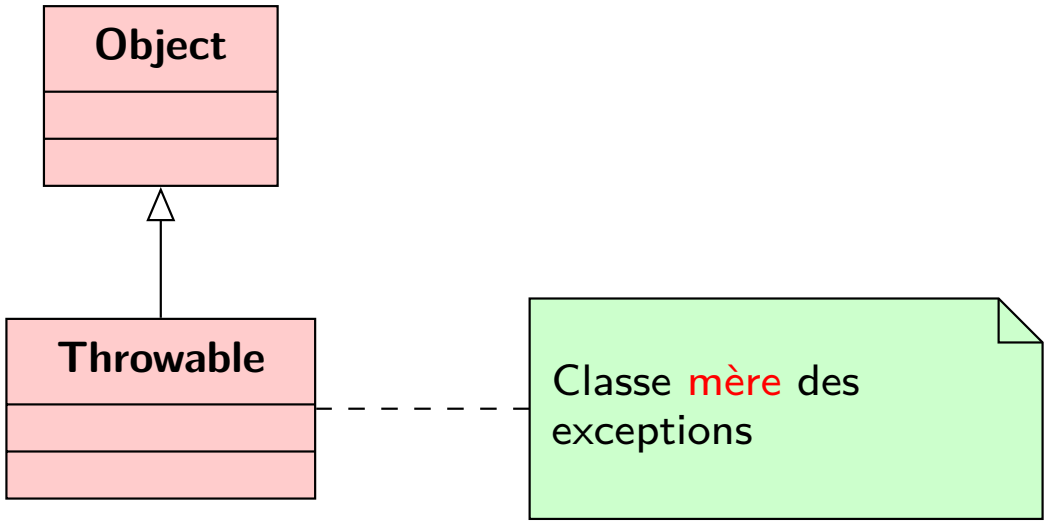
- une faute de saisie
- un problème matériel

} Causes externes au programme

- une faute de programmation observée à l'exécution

`T[i] avec i vaut T.length` ArrayIndexOutOfBoundsException
`o.f() avec o vaut null` NullPointerException

Hiérarchie des exceptions en Java



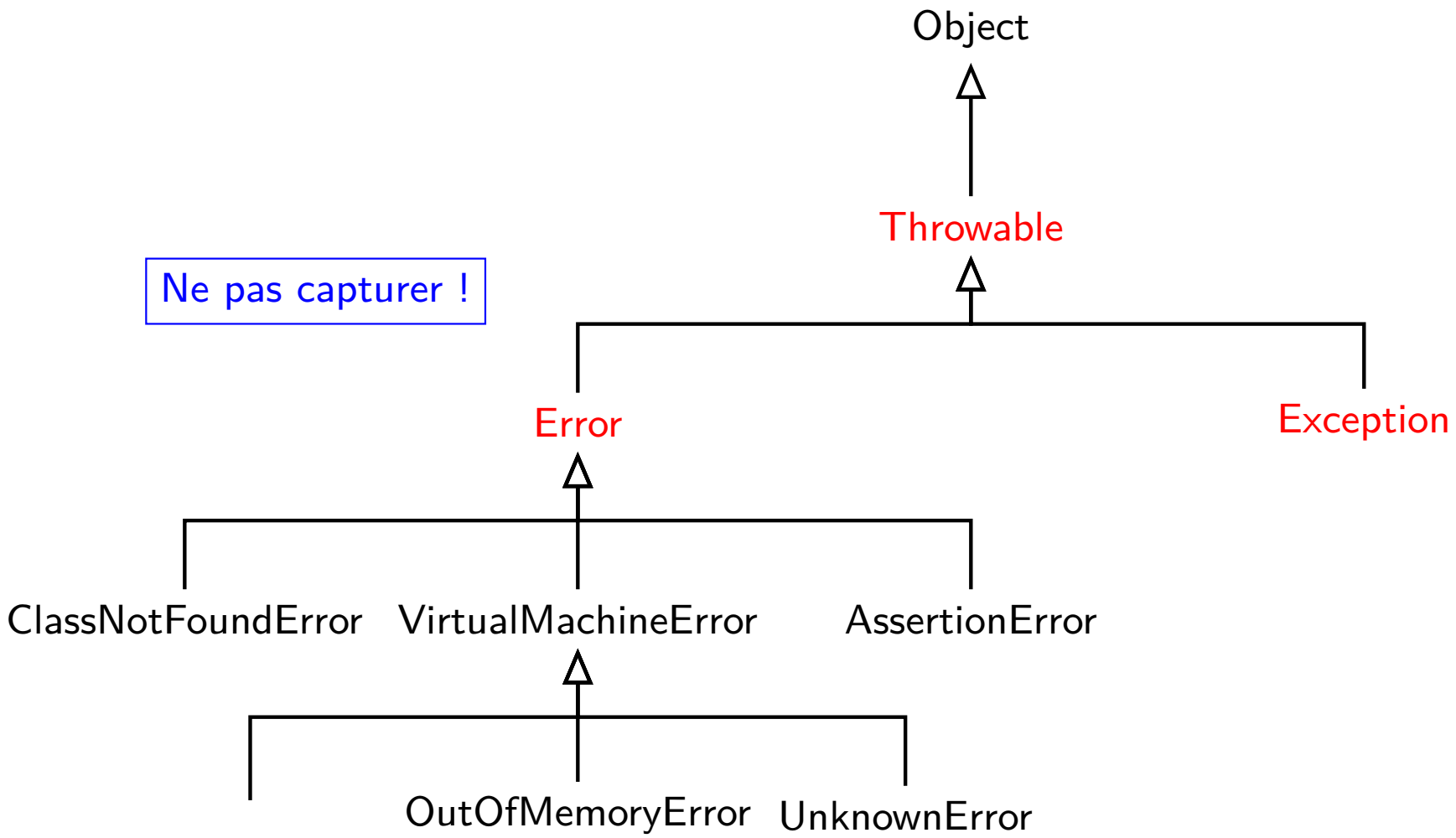
Attributs :

- message d'erreur (une String)
- état de la pile des appels

Méthodes :

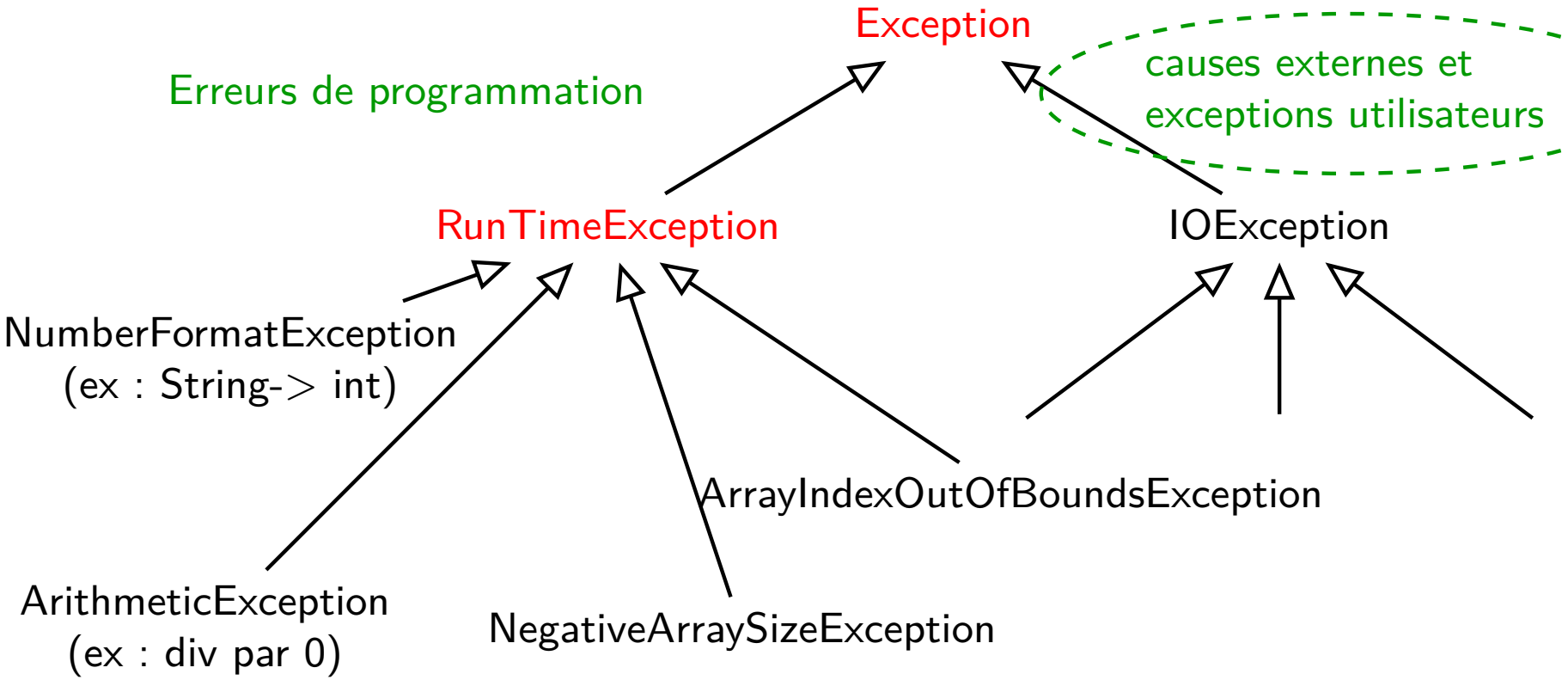
- public Throwable()
- public Throwable(String message)
- public String getMessage()
- public void printStackTrace()

Hiérarchie des exceptions



Error = problème de la machine virtuelle : interne ou manque de ressources

Hiérarchie des exceptions



Méthodes génératrices d'exceptions

Toute méthode doit déclarer les exceptions qu'elle est susceptible de lancer/transmettre :

classe java.io.BufferedReader

```
1 public final String readLine() throws IOException
2 {
3         ..........
4 }
```

...sauf si ce sont des RuntimeException ou Error

classe java.lang.Integer

```
1 public static int parseInt(String s) throws NumberFormatException
2 {
3         ..........
4 }
```

pas obligatoire

Philosophie générale

Exceptions hors contrôle

- les Error car leur traitement ne nous est pas accessible
- les Runtime car on n'aurait pas dû les laisser survenir

Exceptions sous contrôle

- Toutes les autres !

Signalement

Signalement

```
public final String readLine() throws IOException
{
        if (...) throw new IOException();
}
```

- crée un objet d'une certaine classe d'exception (ici IOException)
- signale (lève) cette exception : throw

Exceptions dans une classe utilisateur : Point

```
1  public class Point {
2          private int x, y; //coordonnees
3          public Point(int x, int y) {
4                  ...
5          }
6          public String toString() {
7                  return x + " " + y;
8          }
9          public void deplace (int dx, int dy) {
10         // ajoute dx et dy a x et y
11                 ...
12         }
13 } // fin classe
```

On ajoute la contrainte : un Point doit avoir des coordonnées positives ou nulles.

Au-delà de mettre des assertions, comment assurer le respect de cet invariant de classe tout en maintenant le programme actif ?

```java
public Point(int x, int y) {
        // si x < 0 ou y < 0, que faire ?
        this.x = x;
        this.y = y;
}
```

```java
public void deplace (int dx, int dy) {
        // si (x + dx) < 0 ou (y + dy) < 0,
        // que faire ?
        x += dx;
        y += dy;
}
```

Une classe d'exception pour Point

```
1 public class PointCoordException extends Exception {
2         public PointCoordException() {
3                 super();
4         }
5
6         public PointCoordException(String s) {
7                 super(s);
8         }
9 }
```

On pourrait aussi créer une hiérarchie de classes d'exception pour Point

```java
public Point(int x, int y) {
        // si x < 0 ou y < 0,
        //    generer une PointCoordException

        this.x = x;
        this.y = y;
}
```

devient

```java
public Point(int x, int y) throws PointCoordException {
 if ((x < 0) || (y < 0))
  throw new PointCoordException ("coord invalides "+ x + ' ' + y);
 this.x = x;
 this.y = y;
}
```

```java
public void deplace (int dx, int dy) {
        //  si (x + dx) < 0 ou (y + dy) < 0,
        //  generer une PointCoordException
        x += dx;
        y += dy;
}
```

devient

```java
public void deplace (int dx, int dy) throws PointCoordException {
        if ((x + dx < 0) || (y + dy < 0))
                throw new PointCoordException
                    ("deplacement invalide "+dx+' '+dy);
        x += dx;
        y += dy;
}
```

Capture versus transmission d'exception

```
1 public static int lireEntier() {
2         BufferedReader clavier =
3             new BufferedReader (new InputStreamReader(System.in));
4         String s = clavier.readLine();
5         int ilu = Integer.parseInt(s);
6         return ilu;
7 }
```

Erreur de compilation

lireEntier() doit capturer l'exception susceptible d'être transmise par readLine() ou déclarer qu'elle peut transmettre une exception (la laisser passer)

Solution 1 : la laisser passer

```
1 public static int lireEntier() // * ajout ici d'une clause throws
2 {
3         ...
4 }
```

* throws IOException
* throws IOException, NumberFormatException
* throws Exception (pas très informatif !)

Attention

La clause throws doit "englober" tous les types d'exception à déclaration obligatoire susceptibles d'être transmis de la manière la plus spécifique possible

Solution 2 : la capturer (et la traiter)

1. surveiller l'exécution d'un bloc d'instructions : try
2. capturer des exceptions survenues dans ce bloc : catch

```
1  public static int lireEntier () {
2      BufferedReader clavier = ... ; int ilu = -1;
3      try {
4       String s = clavier.readLine();
5       ilu = Integer.parseInt(s);
6      }
7      catch (IOException e) {
8        ilu = 0; //par defaut
9      }
10     return ilu;
11 }
```

```java
public static int lireEntier () {
    BufferedReader clavier = ... ; int ilu = -1;
    try {
     String s = clavier.readLine();
     ilu = Integer.parseInt(s);
    }
    catch (IOException e) {
      ilu = 0; //par defaut
    }
    return ilu;
}
```

Que se passe-t-il si :

- une exception est générée par readLine ? (IOException)
- par parseInt ? (NumberFormatException)

```java
public static int lireEntier () {
    BufferedReader clavier = ... ; int ilu = -1;
    try {
     String s = clavier.readLine(); // 1
     ilu = Integer.parseInt(s); // 2
    }
    catch (IOException e) {
      ilu = 0; //par defaut
    }
    return ilu; // 3
}
```

Si une exception IOException est générée par readLine :

- 2 n'est pas exécuté
- la clause catch capture l'exception et exécute `ilu = 0;`

l'exécution continue en 3 : ilu retourné (avec valeur 0)

```java
public static int lireEntier () {
    BufferedReader clavier = ... ; int ilu = -1;
    try {
     String s = clavier.readLine(); // 1
     ilu = Integer.parseInt(s); // 2
    }
    catch (IOException e) {
      ilu = 0; //par defaut
    }
    return ilu; // 3
}
```

Si une exception est générée par parseInt (NumberFormatException) :

- aucune clause catch ne capture l'exception ;
- elle est donc transmise à l'appelant ;
- 3 n'est donc pas exécuté.

try + une (ou plusieurs) clause(s) catch

Si une exception est générée dans un bloc try,

- l'exécution s'interrompt
- les clauses catch sont examinées dans l'ordre, jusqu'à en trouver une qui "englobe" la classe de l'exception
- s'il en existe une : le bloc du catch est exécuté, et l'exécution reprend juste après les clauses catch
- sinon : l'exception n'est pas capturée ; elle est donc transmise à l'appelant, et le reste de la méthode n'est pas exécuté

On peut ajouter une clause finally par laquelle on passe toujours.

Try avec ressource (depuis Java 7)

- Constat avant Java 7
  - Certaines ressources (fichiers, flux, etc) doivent être fermées explicitement pour les libérer
- Parade avant Java 7
  - utilisation du bloc finally pour fermer le flux même si une exception est levée
- Ce qui impliquait
  - Déclaration de la ressource en dehors du bloc try
  - La méthode close() de la ressource peut lever une IOException à gérer (autre bloc try/catch ou propagation)

Try avec ressource (depuis Java 7)

- Depuis Java 7, définition possible d'une ressource avec l'instruction try
- La ressource sera automatiquement fermée à la fin de l'exécution du bloc try

Try avec ressource (depuis Java 7)

```
1  try (BufferedReader bufferedReader =
2       new BufferedReader(new FileReader("myFile.txt")))
3  {
4          String line=null;
5          while ((line = bufferedReader.readLine()) != null)
6          {
7                          System.out.println(line);
8          }
9  }
10 catch (IOException ioe) {
11         ioe.printStackTrace();
12 }
```

`bufferedReader` sera fermé proprement à la fin normale ou anormale des traitements -> appel automatique à close.

Try avec ressource (depuis Java 7)

- Ce qu'on peut mettre comme ressource dans le try :
  - Un objet d'une classe implémentant `java.lang.AutoCloseable`
- `java.lang.AutoCloseable` définit une seule méthode : `close()` qui lève une exception de type Exception
- `java.io.Closable extends AutoCloseable`
- Le close de Closable signale une `IOException`

Try avec ressource (depuis Java 7)

- Il est possible de créer de nouveaux types de ressources autoclosable -> implémenter `AutoClosable`
- La ressource doit être déclarée et créée dans le `try`
- Possibilité de spécifier plusieurs ressources :

try (Ressource a=... ; Ressource b=... ; Ressource c=...)
A la fin -> fermeture automatique de c, puis de b, puis de a

Capturer plusieurs types d'exceptions (depuis Java 7)

```
1 try
2 {
3         ...
4 }
5 catch(IOException | SQLException ex)
6 {
7         ex.printStackTrace();
8 }
```

Retour à lireEntier()

Essayons de trouver une bonne façon de gérer les erreurs

```
1  public static int lireEntier () {
2      BufferedReader clavier = ... ;
3      int ilu = 0;
4      try {
5       String s = clavier.readLine(); // 1
6       ilu = Integer.parseInt(s); // 2
7      }
8      catch (IOException e) {
9        ilu = 0; //par defaut
10     }
11     return ilu; // 3
12 }
```

Qu'en penser ?

Problème...

L'erreur n'est pas vraiment réparée : si 0 est retourné, l'appelant ne peut pas savoir que ça ne correspond pas *forcément* à une valeur saisie

Si on ne sait pas comment traiter une exception, il vaut mieux ne pas l'intercepter

Trouvons un traitement plus approprié...

```java
public static int lireEntier()throws IOException {
  BufferedReader clavier = ......;
  int ilu = 0;
  boolean succes = false ;
  while (! succes) {
    try {
      String s = clavier.readLine();
      ilu = Integer.parseInt(s);
      succes = true;
    }
    catch (NumberFormatException e)  {
      System.out.println("Erreur : " + e.getMessage());
      System.out.println("Veuillez recommencer... ");
    }
  } // end while
  return ilu;
}
```

lireEntier()

Changer de niveau d'abstraction

```
1  public static int lireEntier()
2      throws IOException, MauvaisFormatEntierException
3  {
4          BufferedReader clavier = ......;
5          int ilu = 0;
6          try
7          {
8                  String s = clavier.readLine();
9                  ilu = Integer.parseInt(s);
10         }
11         catch (NumberFormatException e)  {
12                 throw new MauvaisFormatEntierException();
13         }
14         return ilu;
15 }
```

L'erreur de bas-niveau retournée est interceptée
et transformée en erreur du niveau de lireEntier()

À ne pas faire

Chercher à traiter des exceptions à tout prix

```
1 public void f (String nomFichier)
2 {
3         // On essaye d'ouvrir le fichier dont le
4         // nom est passe en parametre
5         // Si une FileNotFoundException surgit,
6         // que faire ?
7 }
```

À faire : transmettre l'exception à l'appelant jusqu'à arriver à la méthode qui a décidé du nom de fichier

Règles assurant la substituabilité

Dans la redéfinition (contrôlée par `@Override`) :

- Ajouter une déclaration n'est pas possible
- Retirer une exception est possible
- Spécialiser une exception est possible

Philosophie : la redéfinition ne doit pas surprendre

```
Dessiner() throws RectangleException, IOException
```

Peut être redéfinie en

```
Dessiner() throws NegativeWidthException
```

Si NegativeWidthException est une sous-classe de RectangleException

Synthèse

Assertions

Conditions à vérifier en certains points du programme

Erreurs de logique du programme

Pour la mise au point du programme

Interruption du programme

Exceptions

Objets représentant les erreurs pouvant survenir à l'exécution

Erreurs imprévisibles

Hors du contrôle du programmeur (liées aux ressources extérieures)

À accepter et gérer à l'exécution

Poursuite du programme