Chapitre 4
Les Exceptions

Voir Liskov&Guttag,
ch. 4 « Exceptions », pp 57-75

Procédures partielles et robustesse des programmes

- On n'est pas toujours en mesure de définir des output sensés pour toutes les valeurs d'inputs d'une procédure
- Alors, on définit des procédures partielles
- Exemple

```
/**
 * @requires n,d > 0
 * @return le PGCD de n et d
 */
public static int gcd(int n, int d)
{ ... }
```

- Une procédure partielle ne garantit rien dans le cas où les inputs ne vérifient pas la contrainte **`@requires`** (elle peut boucler, « planter », renvoyer un résultat quelconque)
- Ces procédures partielles mènent à des programmes non robustes
- Un programme robuste fournit un comportement raisonnable et bien défini même en présence d'erreurs. Au pire, il fournit un message d'erreur clair et s'arrête.

Des procédures partielles aux procédures totales

- Pour améliorer la robustesse, on peut avoir recours à des procédures **totales**
- Lorsqu'elle ne peut traiter les inputs reçus de la manière attendue, une procédure totale doit en informer l'appelant
- Une première solution : retourner une valeur spéciale du type de la valeur de retour attendue
- Exemple

```
/**
  * @return n! si n>0, 0 sinon
  */
public static int fact(int n)
{ ... }
```

Retour d'une valeur particulière du type de retour

- Cette manière de faire comporte divers inconvénients
  - risque pour l'appelant de ne pas remarquer la survenance d'un cas particulier
    - Ex : `z = x + Num.fact(y);`
  - complication du code appelant
    - Ex:

```
int r = Num.fact(y);
if (r>0) z = x + r; else...
```

  - impossible si toutes les valeurs du type de retour sont déjà des résultats possibles dans le cas normal
    - Ex: la méthode `E get(int index)` de l'interface `List<E>`

- Le mécanisme des exceptions résout ces 3 problèmes

Principes des exceptions

- Si la procédure appelée se termine normalement, elle renvoie une valeur du type de retour
- Si un problème survient pendant son exécution une **exception** (objet d'un type spécial) est renvoyée
- A différentes catégories de terminaisons exceptionnelles correspondent différents types d'exceptions
- Les noms des types d'exceptions doivent être choisis de manière à communiquer de l'information sur la nature du problème
  - Ex: la méthode `E get(int index)` de l'interface `List<E>` peut retourner des `IndexOutOfBoundsException`

Exceptions et spécifications

- Le fait qu'une procédure puisse se terminer autrement qu'en renvoyant une valeur du type de retour (i.e. en renvoyant une exception) est indiqué par la présence de l'instruction **`throws`** dans sa signature
  - `[`*`modificateurs`*`] type_de_retour nomProc (typeParam_1 nomParam_1, ... , typeParam_m nomParam_m,) [throws typeException_1, ..., typeException_n]`

- Exemple

```
public static int fact (int n) throws NonPositiveException
```

  - c'est à dire que `fact` renvoie :
    - soit un entier
    - soit un objet de type NonPositiveException

Exceptions et spécifications

- Conséquence de la présence d'exceptions sur la spécification
  - **Au niveau de la signature (`throws`)**, on exigera que l'on renseigne **tous** les types des exceptions que la procédure peut renvoyer dans le cadre de son comportement « ordinaire » i.e. **sur des inputs admis par la clause `@requires`**
  - **Au niveau de la clause `@throws`**, on exigera que soit clairement indiqué **la cause de chaque type d'exception**
  - **Au niveau de la cause `@effects`,** on exigera **une description complète des effets dans chacun des cas**. En particulier, les inputs apparaissant dans la clause `@modifies` sont-ils également modifiés si telle ou telle exception est renvoyée et de quelle manière?

Exceptions et spécifications

Exemple 1

```
/**
* @requires v != null
* @modifies v
* @throws NullPointerException si x est null
* @throws NotSmallException si v contient un élément > x
* @effects sinon, ajoute x dans v
*/
public static void addMax(List<Integer> v, Integer x)
    throws NullPointerException, NotSmallException {
    ...
}
```

- Ici, la spécification ne dit rien du cas où `v` ne ferait référence à aucun objet. Il est en dehors des hypothèses.
- Si aucun comportement (« normal » ou d'exception) n'est prévu pour une combinaison d'inputs donnée, celle-ci doit être exclue par la clause **`@requires`**. Le comportement dans tous les autres cas doit être clairement spécifié dans la **`postcondition.`**

Exceptions et spécifications

Exemple 2

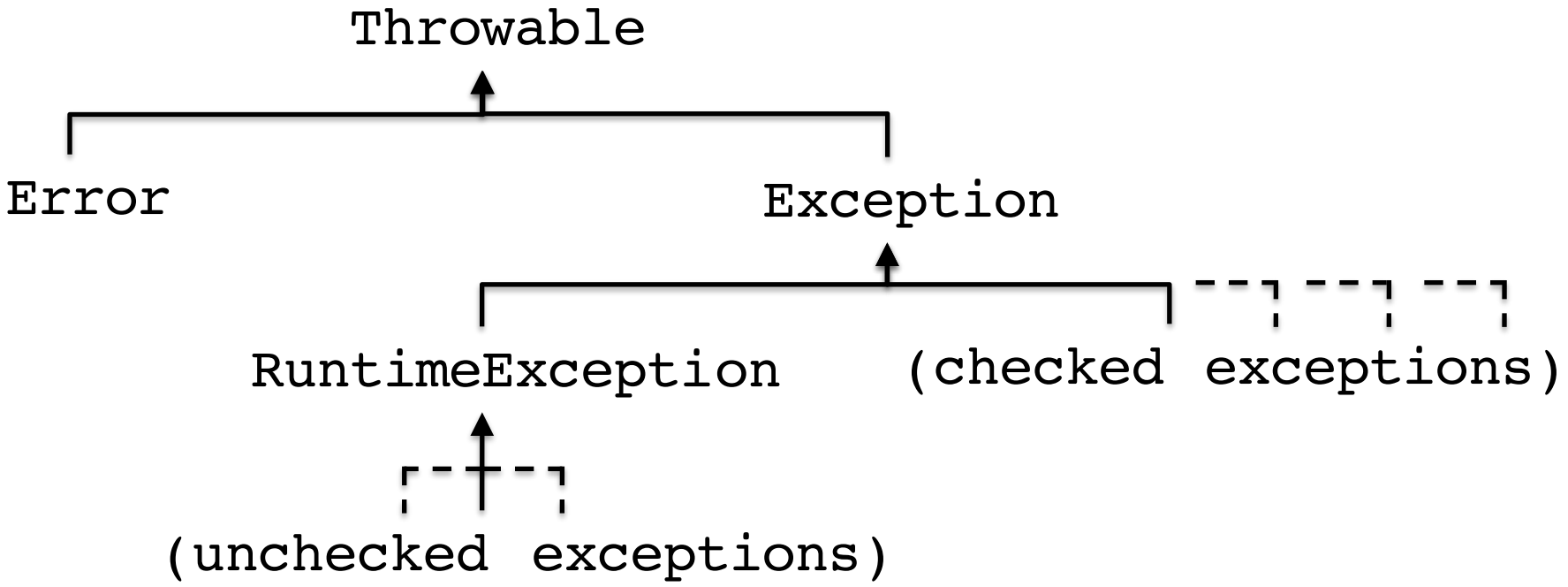
```
/**
* @requires v != null
* @modifies v
* @effects ajoute x dans v,
* @throws NullPointerException si x est null
* @throws NotSmallException si v contient un élément > x
*/
public static void addMax(List<Integer> v, Integer x)
    throws NullPointerException, NotSmallException {
    ...
}
```

- Une autre façon de spécifier est d'indiquer en premier lieu le comportement "normal" avant les exceptions. On peut donc également indiquer les clause **`@throws`** après les clauses **`@effects`** ou **`@return`**
- Il s'agit juste d'un point de vue différent, par contre nous demandons de garder une cohérence au sein du code quant à la façon de spécifier

Les exceptions en Java

Types d'exceptions

- Les types d'exceptions sont des sous-types d'**`Exception`** ou bien de **`RuntimeException`**



Les exceptions en Java

Types d'exceptions

- La plupart des types d'exceptions prédéfinis sont des exceptions non vérifiées/non contrôlées (unchecked exceptions)
  - Ex: `NullPointerException, IndexOutOfBoundsException`

- Il y a **deux différences** entre **les exceptions vérifiées** et **non vérifiées**
  - si une procédure renvoie une exception vérifiée, Java exige que celle-ci figure dans la signature de la procédure
    - dans le cas contraire, une erreur de compilation survient
    - cette contrainte ne s'applique pas aux exceptions non vérifiées
  - si une partie de programme appelle une procédure qui peut renvoyer une exception vérifiée, Java exige que cette partie de programme traite explicitement le cas d'exception (comment? --> voir plus loin)
    - dans le cas contraire, une erreur de compilation survient
    - cette contrainte ne s'applique pas aux exceptions non vérifiées

Les exceptions en Java

Types d'exceptions

- Nous allons cependant être **plus contraignants que Java** et exiger que **toutes les exceptions pouvant survenir sous hypothèse de la précondition, vérifiées ou non, soient listées dans la clause `@throws` et l'instruction `throws` des procédures**

- Pourquoi ?
  - information nécessaire pour l'utilisateur de la procédure
  - la signature expose l'information de manière plus immédiate que les clauses `@effects` et `@throws` (qui doivent de toute façon décrire les cas de survenance de toutes ces exceptions)

- Exemple

```
/**
* @requires a est trié
* @throws NullPointerException si a == null,
* @throws NotFoundException si x n'appartient pas à a,
* @return i t.q. a[i]=x sinon
*/
public static int search (int[] a, int x)
    throws NullPointerException, NotFoundException { ... }
```

Les exceptions en Java

Définition d'exceptions

```
public class NewKindOfException extends Exception {
    public NewKindOfException() {super();}
    public NewKindOfException(String s) {super(s);}
}
```

- On définit un nouveau type d'exception **vérifiée** en sous-typant `Exception`
  - `extends Exception`
- On définit un nouveau type d'exception **non vérifiée** en sous-typant `RuntimeException`
  - `extends RuntimeException`
- Définir de nouvelles classes d'exception est très simple
  - deux constructeurs dont le code est hérité

Les exceptions en Java

Création d'exceptions

- On peut créer une nouvelle instance de `NewKindOfException` (pour la renvoyer par la suite) d'une des deux manières suivantes
  - `Exception e1 = new NewKindOfException("origine de l'exception");`
  - ou `Exception e2 = new NewKindOfException();`
- On peut consulter le `String` contenu dans l'exception via la méthode `toString()`
  - Ex: `String s = e1.toString()` a pour effet de mettre dans la `String` référencée par `s` la valeur "`NewKindOfException: origine de l'exception`"

Les exceptions en Java

Création d'exceptions

- Les nouveaux types d'exceptions créés par le programmeur peuvent être placés
  - **dans le même package que la classe** qui les produit
    - pas très pratique car multiplication des types d'exceptions
  - **dans un package particulier où sont repris tous les types d'exceptions du programme**
    - permet de réutiliser un même type par plusieurs procédures (le nom de la procédure qui la renvoie peut de toute façon être fourni en paramètre à la création de l'exception)

- Les noms des types d'exceptions créés par le programmeur ne doivent pas nécessairement se terminer par **`Exception`** mais c'est une **bonne pratique** qu'il convient de conserver

Les exceptions en Java

Renvoi d'exceptions

- Renvoyer une exception se fait au moyen de l'instruction **`throw`**
- Exemple

```
if (n<=0) throw new NonPositiveException("Num.fact(int)");
```

- L'utilité première de la `String` passée en paramètre est d'indiquer l'origine de l'erreur : *"nomClasse.nomProcédure(type des paramètres formels)"*
- Ainsi, si la partie de programme qui reçoit l'exception ne peut la gérer, elle peut au moins afficher à l'écran l'origine du problème ou l'écrire dans un log (ou fichier de journalisation)
- Comme un même type d'exception peut être renvoyé par plusieurs procédures, **indiquer dans la `String` la procédure qui en est la source est une bonne pratique, ainsi que ses paramètres formels**

Les exceptions en Java

Gestion d'exceptions

- Lorsqu'une procédure appelée renvoie une exception plutôt que de se terminer normalement (avec une éventuelle valeur de retour), le contrôle ne revient pas juste après l'appel

- Il y a deux possibilités
  - Le code appelant utilise les instructions **`try... catch`** pour gérer l'exception
    - Exemple

```
try{
   x=Num.fact(y);
} catch (NonPositiveException e) {
   // ici, on peut utiliser e
}
```

  - Le code appelant ne gère pas l'exception et la propage à son tour au code qui l'appelle

Les exceptions en Java

`try... catch`

```
try {
    ...
} catch (TypeDException1 e1) {
    ...
} catch (TypeDException2 e2) {
    ...
} finally {
    ...
}
```

- Le bloc suivant le mot clé **`try`** contient les instructions qui seront exécutées jusqu'à ce que ce qu'une exception survienne ou que le bloc se termine

Les exceptions en Java
`try... catch`

- Si une exception survient et si cette exception est une instance d'un des types d'exception `TypeDException1, TypeDException2,...` alors le bloc suivant la première clause **`catch`** pouvant convenir est exécuté
  - Un **`catch`** convient si le type effectif de l'exception est un sous-type du type d'exception de la clause
  - L'argument de la clause appelée (`e1, e2,...`) contient alors une référence vers l'exception interceptée
- Le bloc suivant le mot-clé **`finally`** est toujours exécuté, et l'est toujours en dernier lieu (même si le bloc **`try`** contient une instruction de rupture de séquence, comme **`break`** ou **`return`**)
- Les clauses **`catch`** et **`finally`** sont optionnelles

Les exceptions en Java
`try... catch`

- Exemple

```
try {
    x = Arrays.search(v,y);
} catch (Exception e) {
    system.err.println(e);
    return;
}
```

- Dans ce cas, tant `NullPointerException` que `NotFoundException` sont gérées par la clause **`catch`**

Les exceptions en Java
try... catch

- Les **try...catch** peuvent être imbriqués
- Exemple

```
try {
    ...;
    try {
        x=Arrays.search(v,7);
    } catch (NullPointerException e) {
        throw new NotFoundException("…");
    }
} catch (NotFoundException b) {...}
```

Les exceptions en Java
`try... catch`

- Un même **`catch`** peut traiter différents types d'exceptions pour autant que le comportement en réaction puisse s'appliquer à ces différents types.
- Exemple

```
try {
    ...
} catch (TypeDException1 | TypeDException2 e){
    ...
}
```

Les exceptions en Java

Propagation

- Si l'exécution d'une instruction dans le corps d'une procédure P génère une exception qui n'est pas récupérée par un **`catch`** dans P, alors Java renvoie l'exception au code qui appelle P.
  Il faut toutefois qu'une des deux conditions ci-dessous soit vérifiée:
  - le type de l'exception ou un de ses super-types apparaît dans la signature (instruction **`throws`**) de P
  - ou l'exception est de type non vérifiée (unchecked exception)
- Dans le code qui appelle P, le même processus s'applique et ainsi de suite.
- La propagation d'une exception se poursuit donc de méthode appelée à méthode appelante, jusqu'à rencontrer une clause **`catch`** qui convienne.
- Si une telle clause ne peut être trouvée, l'exception est propagée jusqu'à l'environnement d'exécution Java qui signale alors une erreur.

Les exceptions en Java

Attention aux exceptions non vérifiées

- En Java, c'est au **programmeur** de vérifier que les exceptions **non vérifiées** renvoyées par une procédure (soit explicitement par une instruction `throw`, ou par propagation) apparaissent dans la signature (instruction `throws`)

- Ceci est nécessaire car les utilisateurs de la procédure doivent pouvoir connaître les différents comportements possibles de la procédure à partir de sa spécification

- Java n'oblige pas à respecter cette contrainte pour les exceptions non vérifiées

Les exceptions en Java

Attention aux exceptions non vérifiées

- N'importe quel appel peut potentiellement donner lieu à une exception non vérifiée
- Il est dès lors difficile d'identifier leur origine
- Exemple

```
try {
        int y = x + 42;
        if (v.contains(x)) { /* do something clever */ }
    } catch (NullPointerException | ClassCastException e) {
        /* gestion de l'exception issue de y = x +42 /*
    }
```

Les exceptions en Java

Attention aux exceptions non vérifiées

- Pour être sûr de l'origine de l'exception, il faut réduire la portée du **try**
- Exemple

```
int y;
try {
   y = x + 42;
} catch (NullPointerException e) { … }
try {
   if (v.contains(x)) { … }
} catch (NullPointerException | ClassCastException e)
{ … }
```

Programmer avec les exceptions

- Lorsque l'on écrit une procédure P qui est susceptible de recevoir une exception du code qu'elle appelle, il faut faire un choix entre:
  - transmettre une exception au code qui appelle P
  - masquer l'exception au code qui appelle P

Programmer avec les exceptions
Transmission d'une exception au code appelant

- Il y a deux manières de transmettre une exception au code appelant
  - propagation **implicite** de l'exception reçue
    - voir ci-avant, « propagation »
    - limité car c'est le même (type d')exception qui est renvoyé
  - renvoi **explicite** d'une exception via l'instruction **`throw`**
    - permet de renvoyer une exception d'un autre type que celle reçue et donc de fournir au code appelant un feedback du bon niveau d'abstraction
    - voir exemple à la page suivante

Programmer avec les exceptions

Transmission d'une exception au code appelant

- Exemple

```
public class Tableaux {
    /**
    * @throws NullPointerException si a == null,
    * @throws EmptyException si a.length == 0,
    * @return le minimum de a sinon
    */
    public static int min (int[] a)
                throws NullPointerException,EmptyException{
        int m;
        try { m=a[0];}
        catch (IndexOutOfBoundsException e) {
            throw new EmptyException("Tableaux.min(int[])");
        }
        for (int i=1; i < a.length; i++) { if (a[i] < m) m = a[i]; }
        return m;
    }
    ...
```

Programmer avec les exceptions

Masquage de l'exception

- Exemple (suite et fin)

```
...
 /**
 * @throws NullPointerException si a == null,
 * @return true si a est trié en ordre croissant; false sinon.
 */
 public static boolean sorted (int[] a) throws NullPointerException {
      int prev;
      try {prev=a[0];}
      catch (IndexOutOfBoundsException e) {return true;}

      for (int i=1; i < a.length; i++) {
           if (prev<=a[i]) prev = a[i];
           else return false;
      }
      return true;
 }
}
```

Programmer avec les exceptions

Remarques

- Exception ≠ erreur !
  - Exemples
    - demander de chercher dans un tableau un élément qui ne s'y trouve pas
    - demander de trier un tableau vide
    - parcourir les records d'un fichier et passer ceux qui sont incorrects (erreur mais pas exception signalée : OK)
    - …
- Souvent, une exception est simplement une situation particulière sur laquelle on veut attirer l'attention
  - Dire si cette situation est normale ou anormale relève souvent de l'arbitraire ou du niveau d'abstraction
    - appeler `get` sur une `List` dont le pointeur courant est `null` est une erreur du point de vue de la liste. Pour l'appelant de `get`, cela peut simplement indiquer la fin du parcours du liste.
  - On peut donc très bien diriger le flux normal d'un programme à l'aide d'exceptions
    - voir exemple précédent

Conception d'abstractions avec des exceptions

- Deux décisions de conception essentielles
  - Quand utiliser des exceptions ?
  - Utiliser des exceptions vérifiées ou non ?

Conception d'abstractions avec des exceptions

Quand utiliser des exceptions ?

- Pour permettre d'**éliminer** la clause **`@requires`**
  - On maintiendra cependant la clause **`@requires`** uniquement
    - si des **raisons d'efficacité prévalent**
      - ex : `search` est plus efficace si `a` est trié
    - ou si le **contexte d'utilisation** est tellement **limité** qu'on a la garantie que la contrainte **`@requires`** sera toujours satisfaite
      - ex : `partition` dans le `quicksort` peut exiger que `i<j` car elle est appelée uniquement par `sort`
- Pour **éviter d'encoder une information singulière dans un résultat ordinaire**
  - ex: `search` renvoie une exception plutôt que -1 si l'élément n'est pas dans le tableau
  - le contraire est risqué dans un contexte d'utilisation général (i.e. non restreint)

Conception d'abstractions avec des exceptions

Utiliser des exceptions vérifiées ou non ?

- Les exceptions vérifiées doivent être soit gérées explicitement par la procédure appelante, soit listées dans l'instruction **`throws`**. Le contraire cause une **erreur de compilation**.
- Java **n'effectue pas** ces vérifications pour les **exceptions non vérifiées** !
- Une procédure peut donc retourner une exception non vérifiée sans que cette éventualité apparaisse dans sa signature
- Exemple : une implémentation incorrecte de `search` pourrait renvoyer une `IndexOutOfBoundsException` sans que ce type d'exception apparaisse dans sa signature

Conception d'abstractions avec des exceptions

Utiliser des exceptions vérifiées ou non ?

- Danger : gestion de l'exception par erreur
- Exemple :

```
try {x=y[n];i=Tableaux.search(z,x);}
    catch {IndexOutOfBoundsException e} {
        // gestion de e relative à y
    }
    // code qui suppose que le problème relatif à y est réglé
```

- Deux sources fréquentes de propagation des exceptions non vérifiées :
  - erreurs de programmation
  - problèmes de ressources, ex: plus de place dans le heap, fichier manquant ou endommagé,...

Conception d'abstractions avec des exceptions

Utiliser des exceptions vérifiées ou non ?

- Pourquoi y a-t-il des exceptions non vérifiées alors ?
- Parce que les exceptions vérifiées sont aussi parfois ennuyeuses
  - On doit écrire le code qui les gère même quand on est sûr qu'elles ne surviendront jamais

    ⇒ Complication inutile du code du programme

Conception d'abstractions avec des exceptions

Utiliser des exceptions vérifiées ou non ?

- On utilisera (et éventuellement définira) donc des **exceptions non vérifiées** seulement quand on s'attend à ce que le code appelant ne fasse pas survenir l'exception, i.e.
  - s'il y a une manière pratique et peu coûteuse d'éviter sa survenance
    - Exemple : les tableaux sont essentiellement utilisés dans des boucles du genre `for (int i=0; i < a.length; i++)` où il est facile et peu coûteux de garantir qu'il n'y aura pas d'accès en dehors des limites
  - ou si le contexte d'utilisation est local

Conception d'abstractions avec des exceptions

Utiliser des exceptions vérifiées ou non ?

- Sinon, on utilisera (et éventuellement définira) des **exceptions vérifiées**
  - Exemple : `NotFoundException` dans `search`
    - le contexte d'utilisation n'est pas local
    - on ne peut pas supposer que le code appelant sache à l'avance si l'élément recherché appartient ou non au tableau !

Programmation défensive

- Dans une procédure correcte, une erreur peut survenir à cause :
  - d'une autre procédure
  - du hardware
  - de données erronées entrées par l'utilisateur
- On ne peut donc jamais garantir à 100% que tout fonctionnera bien
- Il y aura toujours des erreurs **hors du champs des hypothèses faites par le programme**
- Que faire alors ?
  - Arrêter le programme ? Pas très robuste...
  - ⇒ Utiliser les exceptions et laisser les méthodes de plus haut niveau gérer la situation (ex : faire un restart)

Programmation défensive

- Pour tous les cas où il apparaît qu'une hypothèse faite par le programme n'est pas vérifiée, on génèrera une **`FailureException`** (non vérifiée)
- Utilisations typiques
  - Violation de précondition (**`@requires`**)
    - Si pas trop coûteux à vérifier
  - Survenance d'une exception qui ne devait pas se produire
- Exemple

```
// contexte où on est sur que x est dans z
try {i=Arrays.search(z,x);}
catch (NotFoundException e){
    throw new FailureException("C.p" + e.toString())
}
```

- On ne renseignera pas **`FailureException`** ni dans la signature des procédures, ni dans la clause **`@throws`** (ni ailleurs dans la post-condition) car **son utilisation est réservée aux situations où les choses ne se passent pas comme spécifiées** !