Gestion des erreurs

Les **exceptions** permettent d'*anticiper les erreurs* qui pourront potentiellement se produire lors de l'utilisation d'une portion de code.

Exemple : on veut écrire une fonction qui calcule l'inverse d'un nombre réel quand c'est possible :

f
entrée : x sortie : $1/x$
Si $x = 0$ *erreur* ***Sinon*** *retourner 1/x*

☞ Mais que faire **concrètement** en cas d'erreur ?

Gestion des erreurs (2)

① retourner une valeur choisie à l'avance :

```
double f(double x) {
  if (x != 0.0) {
    return 1.0 / x;
  } else {
    return Double.MAX_VALUE;
  }
}
```

Mais cela

1. **n'indique pas** à l'utilisateur potentiel qu'il a fait une erreur
2. retourne de toutes façons un **résultat inexact** ...
3. suppose une **convention arbitraire** (la valeur à retourner en cas d'erreur)

Gestion des erreurs (3)

② afficher un message d'erreur :

```
double f(double x) {
  if (x != 0.0) {
      return 1.0 / x;
  } else {
    System.out.println("Erreur dans f : division par 0");
    return ????;
  }
}
```

mais que retourner effectivement en cas d'erreur ?...
...on retombe en partie sur le cas précédent

De plus, cela est **très mauvais** car produit des *effets de bord* : affichage dans le terminal alors que ce n'est pas du tout dans le rôle de `f` !

Gestion des erreurs (4)

③ retourner un code d'erreur :

```
boolean f(double x, Double resultat) {
  if (x != 0.0) {
      resultat = 1.0 / x;
      return true;
  } else {
      return false;
  }
}
```

Cette solution est déjà **meilleure** car elle laisse à la fonction qui appelle `f` le soin de décider quoi faire en cas d'erreur.
Cela présente néanmoins l'inconvénient d'être assez lourd à gérer pour finir :

- cas de l'appel d'appel d'appel.... ...d'appel de fonction,
- mais aussi écriture peu intuitive :
  `if (f(x,y))... // le résultat de la division est dans y`
  au lieu de
  `y=f(x);`

Exceptions

Il existe une solution permettant de *généraliser* et d'*assouplir* cette dernière solution : déclencher une **exception**

☞ mécanisme permettant de *prévoir une erreur* à un endroit
et de **la gérer à un autre endroit**

Exceptions (2)

Principe :

- lorsque qu'une erreur a été détectée à un endroit, on la signale en *« lançant » un objet* contenant toutes les informations que l'on souhaite donner sur l'erreur
  (« lancer » = créer un objet disponible pour le reste du programme)
- à l'endroit où l'on souhaite gérer l'erreur (au moins partiellement), on peut *« attraper » l'objet « lancé »* (« attraper » = utiliser)
- si un objet « lancé » n'est pas attrapé du tout, cela provoque l'arrêt du programme : *toute erreur non gérée provoque l'arrêt*.

Un tel mécanisme s'appelle « gestion des exceptions ».

Syntaxe de la gestion des exceptions

On cherche à remplir 4 tâches élémentaires :

1. signaler une erreur
2. marquer les endroits réceptifs aux erreurs
3. leur associer (à chaque endroit réceptif) un moyen de gérer les erreurs qui se présentent
4. éventuellement, « faire le ménage » après un bloc réceptif aux erreurs

On a donc 4 mots du langage Java dédiés à la gestion des exceptions :

`throw` : indique l'erreur (i.e. « lance » l'exception)
`try` : indique un bloc réceptif aux erreurs
`catch` : gère les erreurs associées (i.e. les « attrape » pour les traiter)
`finally` : (optionel) indique ce qu'il faut faire après un bloc réceptif

Syntaxe de la gestion des exceptions

Notez bien que :

- L'indication des erreurs (`throw`) et leur gestion (`try`/`catch`) sont le plus souvent *à des endroits bien séparés* dans le code
- Chaque bloc `try` possède son/ses `catch` associé(s)

Pour résumer

Une exception est un moyen de signaler un événement nécéssitant une attention spéciale au sein d'un programme, comme:

- une **erreur grave**
- une **situation inhabituelle** devant être traitées de façon particulière

☞ But : **améliorer la robustesse des programmes** en :

- séparant le code de traitement des erreurs du code « effectif »
- fournissant le moyen de forcer une réponse à des erreurs particulières

Exceptions : intérêt

Avantages de la gestion des exceptions par rapports aux codes d'erreurs retournés par des fonctions :

- écriture plus facile, plus intuitive et plus lisible
- propagation **automatique** de l'exception aux niveaux supérieurs d'appel (fonction appelant une fonction appelant ...)
  plus besoin de gérer obligatoirement l'erreur au niveau de la fonction appelante
- une erreur peut donc se produire à n'importe quel niveau d'appel, elle sera toujours reportée par le mécanisme de gestion des exceptions

Note : si une erreur peut être gérée localement, le faire et ne pas utiliser le mécanisme des exceptions.)

Syntaxe de la gestion des exceptions

On cherche à remplir 4 tâches élémentaires :

1. signaler une erreur
2. marquer les endroits réceptifs aux erreurs
3. leur associer (à chaque endroit réceptif) un moyen de gérer les erreurs qui se présentent
4. éventuellement, « faire le ménage » après un bloc réceptif aux erreurs

On a donc 4 mots du langage Java dédiés à la gestion des exceptions :

- `throw` : indique l'erreur (i.e. « lance » l'exception)
- `try` : indique un bloc réceptif aux erreurs
- `catch` : gère les erreurs associées (i.e. les « attrape » pour les traiter)
- `finally` : (optionel) indique ce qu'il faut faire après un bloc réceptif

`throw`

`throw` est l'instruction qui **signale l'erreur** au reste du programme.

Syntaxe : `throw` *`exception`*

`exception` est un objet de type `Exception` qui est « lancé » au reste du programme pour être « attrapé »

Exemple :

```
throw new Exception("Quelle erreur !");
```

`Exception` est une classe de `java.lang` qui possède de nombreuses sous-classes et qui hérite de `Throwable`.

`throw` (2)

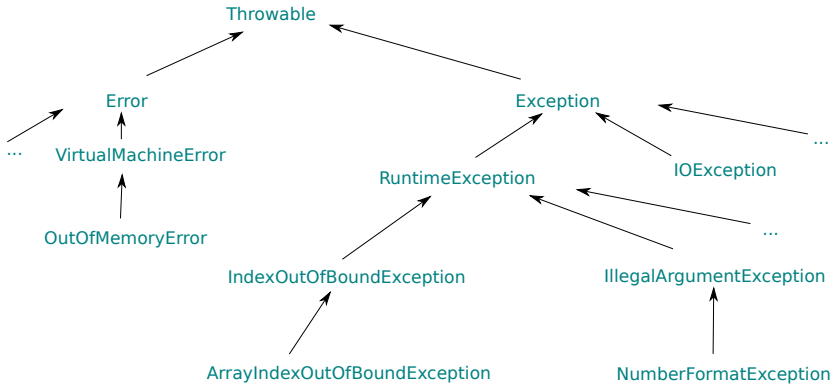
`throw`, en « lançant » une exception, interrompt le cours normal d'exécution et :

- saute au bloc `catch` du bloc `try` directement supérieur, si il existe ;
- quitte le programme si l'exécution courante n'était pas dans au moins un bloc `try`.

Exemple :

```
try {
  // ...
  if (...) {
    throw new Exception("Quelle erreur !");
  }
  // ...
}
catch (Exception e) {
  // ...
}
```

Hiérarchie partielle de `Throwable`



Les sous-classes de Throwable

Chaque sous-classe de `Throwable` décrit une erreur précise

La classe `Error` : 12 sous-classes directes

- Erreur fatale
- Pas censée être traitée par le programmeur

La classe `Exception` : 74 sous-classes directes (hormis les `RuntimeException`)

- Circonstance exceptionnelle
- Souvent une erreur (mais pas toujours)
- Doit être traitée par le programmeur (« *checked exceptions* »)

La classe `RuntimeException` : 49 sous-classes directes

- Exception dont le traitement n'est pas vérifié par le compilateur
- Peut être traitée par le programmeur (« *unchecked exceptions* »)

La classe java.lang.Throwable

```
public class Throwable extends Object
```

- Deux constructeurs:
  - Erreur avec ou sans message

```
public Throwable()
public Throwable(String message)
```

- deux méthodes (parmi d'autres) :
  - Accès au message d'erreur
  - Affichage du chemin vers l'erreur

```
public String getMessage()
public void printStackTrace()
```

try

`try` (*lit.* « essaye ») introduit un **bloc réceptif aux exceptions** lancées par des instructions, ou des méthodes appelées à l'intérieur de ce bloc (ou même des méthodes appelées par des méthodes appelées par des méthodes... ...
... à l'intérieur de ce bloc)

Exemple :

```
try {
  // ...
  y = f(x); // f pouvant lancer une exception
  // ...
}
```

catch

`catch` est le mot-clé introduisant un **bloc dédié à la gestion** d'une ou plusieurs **exceptions**.

Tout bloc `try` doit toujours être suivi d'au moins un bloc `catch` gérant les exceptions pouvant être lancées dans ce bloc `try`.

Si une exception est lancée mais n'est pas interceptée par le `catch` correspondant, le programme s'arrête (message « `Exception in thread ...` » et affichage de la trace d'exécution).

Syntaxe :

`catch (type nom) { ... }`

intercepte toutes les exceptions de type *`type`* lancées depuis le bloc `try` précédent
`type` peut-être une classe prédéfinie de la hiérarchie d'exceptions de Java ou une classe d'exception créée par le programmeur.

Exemple d'utilisation de `catch`

```
try {
 // ...
 if (age >= 150)
 { throw new Exception("age trop grand"); }
 // ...
 if (x == 0.0)
 { throw new ArithmeticException("Division par zero"); }
 // ...
}

catch (ArithmeticException e) {
   System.out.println(e.getMessage());
   e.printStackTrace();
}

catch (Exception e) {
   System.out.println("Qui peut vivre si vieux ?");
}
```

Flot d'exécution (1/3)

Un bloc `catch` n'est exécuté **que** si une exception de type correspondant a été lancée depuis le bloc `try` correspondant.

Sinon le bloc `catch` est simplement ignoré.

En l'absence du bloc `finally`, si un bloc `catch` est exécuté, le déroulement continue ensuite normalement **après** ce bloc `catch`
(ou après le dernier des blocs `catch` du même bloc `try`, lorsqu'il y en a plusieurs).

En aucun cas l'exécution ne reprend après le `throw` !

Flot d'exécution (2/3)

Exemple :
en cas d'erreur (lancement d'une exception) :

```
try {
  // ...
  if (...) {
    throw new Exception("Quelle erreur !");
  }
  // ...
}
catch (Exception e) {
  // ...
}
```

Flot d'exécution (3/3)

Exemple :
si il n'y a pas d'erreur (**pas** de lancement d'exception) :

```
try {
  // ...
  if (...) {
    throw new Exception("Quelle erreur !");
  }
  // ...
}
catch (Exception e) {
  // ...
}
```

try/throw/catch dans la même méthode

```
int lireEntier(int maxEssais) throws Exception
{
  int nbEssais = 1;
  do {
    System.out.println("Donnez un entier : ");
    try {
      int i = clavier.nextInt();
      return i;
    }
    catch (InputMismatchException e) {
      System.out.println("Il faut un nombre entier. Recommencez !");
      clavier.nextLine();
      ++nbEssais;
    }
   } while(nbEssais <= maxEssais);

  throw new Exception ("Saisie échouée");
}
```

`catch` : remarques

Notes :

- Java 7 a introduit le multi-catch : `catch(`*`Exception1 | Exception2 | ..`*`)`
- s'il y a plusieurs blocs catchs toujours les spécifier du plus spécifique au plus général
  (sinon, erreur signalée par le compilateur)

Le bloc `finally`

Le bloc `finally` est optionnel, il suit les blocs `catch`

Il contient du code destiné à être exécuté qu'une exception ait été lancée ou pas par le bloc `try`

☞ But : **faire le ménage** (fermer des fichiers, des connexions etc..)

Bloc `finally` : exemple (1)

```
class Inverse {
  public static void main (String[] args) {
    try {
      int b = Integer.parseInt(args[0]);
      int c = 100/b;
      System.out.println("Inverse * 100 = " + c);
    }
    catch (NumberFormatException e1) {
      System.out.println("Il faut un nombre entier !");
    }
    catch (ArithmeticException e2) {
      System.out.println ("Parti vers l'infini !");
    }
    finally {
      System.out.println("Passage obligé !");
    }
  }
}
```

Bloc `finally` : exemple (2)

Exemples d'exécution :

```
>java Inverse 4.1
Il faut un nombre entier!
Passage obligé !

>java Inverse 0
Parti vers l'infini!
Passage obligé !

>java Inverse 4
Inverse * 100 = 25
Passage obligé !

>java Inverse
Passage obligé !
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: 0
```

« Relancement »

Une exception peut être **partiellement traitée** par un bloc `catch` et *attendre* un *traitement* plus complet *ultérieur* (c'est-à-dire à un niveau supérieur).

Il suffit pour cela de **« *relancer* » l'exception** au niveau du bloc n'effectuant que le traitement partiel.
(Il faudra bien sûr pour cela que l'appel à ce bloc `catch` soit lui-même dans un autre bloc `try` à un niveau supérieur).

« Relancement » : exemple

```
catch (Exception erreur) {
    // traitement partiel :
   System.out.println("Hmm... pour l'instant  "
                    + "je ne sais pas quoi faire"
                    + "avec l'erreur :"
                    +  erreur.getMessage());
   throw erreur; // relance l'exception captée
   // throw new Exception("un autre message"); // alternative
}
```

Déclaration d'une exception

Une méthode lançant une exception sans la traiter localement doit généralement informer qu'elle le fait

Ceci se fait en ajoutant une clause `throws` à l'entête de la méthode
Syntaxe:

`Type method(...) throws` *`Exception1, Exception2, ...`*

Exemple:

```
int inverse(int x) throws Exception
{
 if (x == 0) {
   throw new Exception("Une division par zéro");
 }
 return 1/x;
}
```

Règle « déclarer ou traiter »

Toutes les exceptions en dehors des `RunTimeException` et des `Error` doivent :

- soit **être interceptées** dans la méthode où elles sont lancées;
- soit **être déclarées** par la méthode.

Si une exception de ce type est lancée sans être interceptée

☞ le compilateur émettra un message d'erreur

☞ « *Checked exceptions* »

Exceptions personnalisées

Il est possible de programmer **ses propres classes d'exception**

☞ sous-classe de `Exception` (ou d'une autre sous-classe d'exception existante)

Contenu minimal :

```
class MonException extends Exception
{
  public MonException() {
     super("mon message par défaut";);
  }
  public MonException(String message) {
     super(message);
  }
}
```

☞ permet de préserver le comportement attendu de `getMessage()`

Exceptions personnalisées

Il et possible de définir dans une sous-classe d'exception personnalisée tout membre jugé utile :

- code d'erreurs
- informations sur le contexte de détection de l'exception
- etc.

Exemple ...

Exceptions personnalisées : exemple

```
class TropChaudException extends Exception
{
 private double temperatureAnormale;
 private String consigne;

 public TropChaudException() { super("Température trop élevée");  }
 public TropChaudException(String message) { super(message); }

 public TropChaudException(double uneTemperature, String uneConsigne) {
  super("Température trop élevée");
  temperatureAnormale = uneTemperature;
  consigne = uneConsigne;
 }
 public double getTemperature() {
  return temperatureAnormale;
 }
 public String getConsigne() {
  return consigne;
 }
}
```

Exceptions personnalisées : exemple (2)

```
try {
  //...
    if (temperature > TEMP_MAX){
      throw new TropChaudException(temperature,
                "Vérification de l'appareil de mesure");
    }
  }

  catch(TropChaudException e){
    System.out.print(e.getMessage() + " : " );
    System.out.println(e.getTemperature());
    System.out.println("Consigne -> " + e.getConsigne());
  }
```

Exemple d'exécution du bloc `catch` :

```
Température trop élevée : 150.0
Consigne -> Vérification de l'appareil de mesure
```

Exemple complet (1)

```
public static void main(String[] args) {
  int nbEssais = 0;
  final int MAX_ESSAIS = 2;
  ArrayList<Double> mesures = new ArrayList<Double>();

  do {
    nbEssais++;

    acquerirTemp(mesures);     // remplit le tableau

    try {
      plotTempInverse(mesures);
     }
   // ...
```

Exemple complet (2)

```
    catch (ArithmeticException e) {
      if (nbEssais < MAX_ESSAIS) {
        System.out.println("Ressaisir les valeurs !");
      } else {
        System.out.println("Il y a déjà eu au moins "
           + MAX_ESSAIS + " essais.");
        System.out.println(" -> abandon");
      }
    }
  } while (nbEssais < MAX_ESSAIS);
}
```

Exemple complet (3)

```
private static void plotTempInverse(ArrayList<Double> t)
throws ArithmeticException
{
  for(int i = 0; i < t.size(); i++) {
    try {
      plot(inverse(t.get(i)));
    } catch (ArithmeticException e) {
      System.out.println("Problème à l'indice :" + i);
      // RELANCEMENT
      throw e;
    }
  }
}
```

Exemple complet (4)

```
private static void plot(double x) {
  // fait le dessin
}

private static double inverse(double x)
   throws ArithmeticException // PAS NECESSAIRE
                              //RunTimeException
{
  if (x == 0.0) {
    throw new ArithmeticException("Division par 0 !");
  }
  return 1.0/x;
}
```

Conseils/mise en garde

La gestion d'une exception coûte beaucoup plus en temps de calcul qu'un simple `if.. then.. else`

☞ **Si l'erreur peut-être traitée là où elle est découverte, il faut le faire sans passer par les exceptions**

Lancer des exceptions spécifiques est plus informatif et utile !