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)

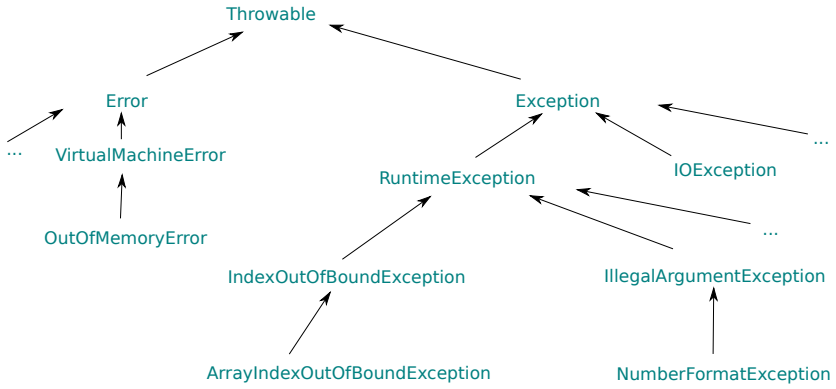
`throw`, en « lançant » une exception, interrompt le cours normal d'exécution et :

- 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
```