Gestion d'exceptions

Définition

La cause du mauvais fonctionnement d'un programme peut être la conséquence de 2 types d'erreurs : une erreur de programmation ou une erreur pendant l'exécution du programme. Si le programmeur commet une erreur de programmation, le programme ne se compilera généralement pas, mais il existe des erreurs de programmation qui surviennent après la compilation, à l'appel de la machine virtuelle par java.exe (pour je JDK 1.2 de Sun). Toutes ces erreurs peuvent et doivent être corrigées par le programmeur. Dans les deux cas précédents, le programme ne démarre pas.

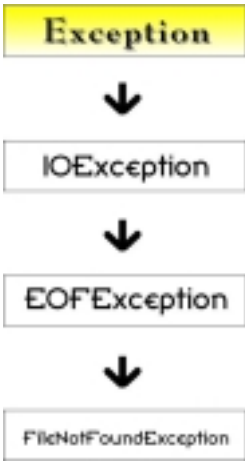
Le second type d'erreur survient pendant l'exécution du programme. Ce sont en quelque sorte des « bugs ». On distingue 2 catégories : les *erreurs* de machine virtuelle (vous ne pouvez quasiment rien y faire ☹), et les *exceptions* que vous devez gérer. Ces exceptions peuvent se produire dans beaucoup de cas, elles ont cependant plus de chance de survenir lors d'un transfert de données. Par exemple, si un programme essaie d'écrire sur un disque plein ou protégé en écriture, une exception de type IOException (*input/output* ou entrée/sortie de données) sera générée. La gestion de cette exception permettra au programme de ne pas « planter ». Vous pouvez spécifier ce que le programme doit faire lors de l'apparition d'exceptions.

try…catch

try…catch peut être expliqué comme : « essaye ce bout de code (try), si une exception survient, attrape-la (catch) et exécute le code de remplacement (s'il y en a un) »

```
try {
    //code susceptible de produire des exceptions
} catch (Exception e) {
    // code de remplacement, se limite souvent à un System.out.println("Une erreur : "+e);
    }
```

Exception est la super-classe de toutes les exceptions, elle capte toutes les exceptions. Elle fonctionne dans tous les cas, cependant il est préférable de bien canaliser l'exception en spécifiant le type d'exception le plus précis possible (voir le schéma ci contre). On doit spécifier un nom à cette exception créée, ici c'est *e*, on choisit généralement e ou ex. Vous pouvez en raison de la portée de variables spécifier le même nom pour toutes vos exceptions (ce nom n'est reconnu qu'à l'intérieur de catch).



Finally

La clause *finally* permet d'exécuter le code qu'elle renferme quoiqu'il arrive. Qu'il y ait une exception ou pas, les instructions de la clause finally sont exécutées. Elle fonctionne également avec try.

```
try {
    // instructions à essayer de réaliser
    return ;
} finally {
    // instructions à réaliser absolument
}
return ;
```

Déclarer des méthodes susceptibles de générer des Exceptions

On utilise la clause throws dans la déclaration de méthode. Exemples :

```
Public boolean myMethod (xxx) throws AnException {…}
Public boolean myMethod (xxx) throws AnException, ASecondException, AThirdException {…}
Public void myMethod() throws IOException {…}
```

Générer des exceptions

Ceci sert à faire croire au programme qu'une exception d'un certain type est apparue.

```
NotInServiceException() nis = new NotInServiceException("Exception : DataBase out of use");
throw nis;
```

On note qu'il s'agit de throw et pas de throws.

Créer des exceptions

Dans des programmes complexes, il se peut que les exceptions standard de Java ne soient pas suffisantes et que vous ayez par conséquent besoin de créer vos propres définitions. Dans ce cas, vos exceptions devront hériter d'une exception plus haute dans la hiérarchie.

```
public class SunSpotException extends Exception {
    public SunSpotException () {}
    public SunSpotException(String msg) {
        super(msg);
    }
}
```

En Pratique

Au début, vous ne saurez pas quand utiliser ou ne pas utiliser les exceptions. Les automatismes viennent avec la pratique… Il faut savoir que les exceptions ralentissent les programmes, n'en usez donc pas trop souvent… De toute façon, le compilateur générera un message d'erreur si à un endroit vous avec oublié d'intercepter une exception, et en plus, il vous indiquera quelle exception vous devez intercepter !