Séquence 13

Exceptions

Les exceptions sont un mécanisme de traitement des erreurs qui surviennent à l'exécution. Ce mécanisme peut être utilisé par un programme pour signaler une erreur ou pour apporter un traitement correctif qui résoud le problème détecté.

1 Une exception est un objet

Une exception est matérialisée par un objet instance de la classe `Exception` ou une classe descendante de la classe `Exception`. Le plus souvent, un objet est créé au moment où l'exception est lancée au moyen de l'instruction `throw`. La création de l'objet est tout à fait classique avec un `new`. La plupart des classes d'exceptions ont plusieurs constructeurs dont deux qu'on utilise fréquemment : un constructeur sans paramètre et un constructeur qui prend un paramètre de type `String` qui est un message expliquant le problème détecté.

L'objet lancé par une instruction `throw` peut être attrapé dans un `try...catch`. Le message d'erreur et d'autres information peuvent être retrouvés dans l'objet en question. Le mot-clé `catch` est suivi entre parenthèse d'un type d'exception et d'un identificateur. Il s'agit de quelque chose qui ressemble à une déclaration de variable ou de paramètre. L'identificateur sert de nom pour désigner l'objet rattrapé dans le bloc qui suit. L'objet instance d'une sous-classe d'exception peut être utilisé dans ce bloc. Si l'on veut retrouver le message, on peut utiliser la méthode `getMessage` héritée de `Exception`.

L'exemple suivant est celui d'un parking qui peut accueillir un nombre donné de voiture représentées dans le programme par leur numéro d'immatriculation (String). Toutes les erreurs détectées par le programme sont matérialisée dans ce programme par une exception de type `RuntimeException` en utilisant le message pour préciser le problème. Il y en a trois qui sont traités :

— le nombre de place du parking est inférieur ou égal à 0.
— la parking est plein, impossible de garer un nouveau vehicule
— on ne peut pas faire sortir un véhicule qui n'est pas garé dans le parking.

```
1  public class Parking {
2      private String[] park;
3
4      public Parking(int nbPlaces){
5          if (nbPlaces <= 0)
6              throw new RuntimeException(nbPlaces +
7                                          "␣n'est␣pas␣une␣nombre␣de␣places␣correct");
8          park = new String[nbPlaces];
```

```
9       }
10
11      public String[] getContent(){
12          return park;
13      }
14
15      public void arriver(String num){
16          int i=0;
17          while(i<park.length && park[i]!=null)
18              i++;
19          if (i==park.length)
20              throw new RuntimeException("Plus␣de␣place␣dans␣le␣parking");
21          else
22              park[i]=num;
23      }
24
25      public void partir(String num){
26          int i = 0;
27          while(i<park.length && (park[i] == null || ! park[i].equals(num)))
28              i++;
29          if (i==park.length)
30              throw new RuntimeException("Vehicule␣non␣trouve:␣" + num);
31          else
32              park[i]=null;
33      }
34
35      public String toString(){
36          String res = "";
37          int nbvides=0;
38          for (String str: park)
39              if (str==null)
40                  nbvides++;
41              else
42                  res = res + str + ",␣";
43          res = res + nbvides + "␣places␣vides";
44          return res;
45      }
46
47      public static void main(String[] args){
48          Parking park = null;
49          int nbp = 0;
50          for (int i=0; i<6; i++){
51              try{
52                  if (park == null){
53                      park = new Parking(nbp);
54                      System.out.println("Parking␣cree:␣" + park);
55                      System.out.println("-----------------");
56                  }else{
57                      park.arriver("BWV␣10" + i);
58                      System.out.println("Nouvelle␣arrivee");
59                      System.out.println(park);
60                      System.out.println("-----------------");
61                  }
```

```
62                      park.partir("BWV␣21");
63                      System.out.println(park);
64                      System.out.println("-----------------");
65                  }catch (RuntimeException exc){
66                      nbp = 3;
67                      System.out.println("Exception␣attrapée");
68                      System.out.println(exc);
69                      System.out.println("Message:␣" + exc.getMessage());
70                      System.out.println("-----------------");
71                  }
72              }
73          }
74      }
```

Ligne 65, dans la clause `catch`, il y a déclaration d'une exception de type `RuntimeException` et de nom `exc`. Ce nom donné à l'objet rattrapé est utilisé dans le bloc qui suit : ligne 68, l'objet est affiché avec `System.out.println` et ligne 69, la méthode `getMessage` est appelée sur l'objet en utilisant ce nom.

2 Fonctionnement des exceptions

Un premier principe est essentiel : les exceptions ne doivent être utilisées que si la détection du problème et sa résolution sont situées dans deux méthodes différentes. Si un problème est détecté et résolu dans la même méthode, il n'est pas utile d'utiliser des exceptions : les instructions ordinaires telles que les if et les boucles suffisent.

Ensuite, les deux méthodes qui traitent les deux temps du problème (détection et résolution) ne sont pas totalement indépendantes : celle qui détecte le problème doit être appelée directement ou indirectement par celle qui le résoud. Directement : il y a une invocation de la méthode qui détecte le problème dans le code de la méthode qui le résoud. Indirectement : il y une invocation d'une méthode qui invoque une méthode . . .qui invoque la méthode qui détecte le problème.

Dans l'exemple du Parking, le constructeur et les méthodes `arriver` et `partir` détectent chacun un problème. La méthode `main` rèsoud les problèmes. Les invocations du constructeur et des deux méthodes sont dans le `main` à l'intérieur du `try...catch`. Il s'agit donc d'une relation directe dans ce cas.

Une exception qui est levée à un moment donné ne peut être rattrapée que par une des méthodes en cours d'exécution, une des méthodes qui a une mémoire privée dans la pile à ce moment-là.

Un autre point important des exceptions est que lorsqu'une exception est levée, le cours normal du programme est interrompu. Des instructions qui étaient prévues ne sont pas exécutées. Cela concerne aussi bien du code de la méthode qui lève l'exception que du code de celle qui la rattrape et également des méthodes intermédiaires s'il y en a.

3 Illustration du code non exécuté

La programme suivant veut illustrer ce qui est exécuté ou non dans le cas d'un lancer d'exception. Il y a une méthode `methodeAttrappeuse` qui dispose d'un `try...catch` pour traiter les problèmes. Cette méthode appelle une méthode `methodeIntermediaire` qui ne lance pas d'exception mais qui appelle une troisième méthode. Celle-ci s'appelle `methodeLanceuse` et elle lance une exception si son paramètre vaut 0.

```
public class CodeNonExecute{
    public void methodeLanceuse(int x){
        System.out.println("␣␣␣␣␣␣Debut␣de␣methodeLanceuse");
        if (x == 0){
            System.out.println("␣␣␣␣␣␣Exception␣lancee␣par␣methodeLanceuse");
            throw new RuntimeException();
        }
        System.out.println("␣␣␣␣␣␣Fin␣de␣methodeLanceuse");
    }
    public void methodeIntermediaire(int n){
        System.out.println("␣␣␣␣Debut␣de␣methodeIntermediaire");
        methodeLanceuse(n);
        System.out.println("␣␣␣␣Fin␣de␣methodeIntermediaire");
    }
    public void methodeAttrapeuse(int nb){
        System.out.println("␣␣Debut␣de␣methodeAttrapeuse");
        try{
            methodeIntermediaire(nb);
            System.out.println("␣␣Suite␣de␣methodeAttrapeuse");
        }catch(RuntimeException exception){
            System.out.println("␣␣Résolution␣par␣methodeAttrapeuse");
        }
        System.out.println("␣␣Fin␣de␣methodeAttrapeuse");
    }
    public static void main(String[] args){
        CodeNonExecute cne = new CodeNonExecute();
        java.util.Scanner scan = new java.util.Scanner (System.in);
        int nombre;
        System.out.println("Debut␣de␣main");
        System.out.print("Entrez␣un␣nombre:␣");
        nombre = scan.nextInt();
        cne.methodeAttrapeuse(nombre);
        System.out.println("Fin␣de␣main");
    }
}
```

Dans ce programme, les affichages pemettent de comprendre les emboîtements dans les exécutions de méthodes. Ils représentent des portions de code qui dans un cas réel, comporteraient n'importe quelles instructions. Voyons d'abord le cheminement normal du programme, les instructions exécutées lorsque l'exception n'est pas lancée.

```
> java CodeNonExecute
Debut de main
Entrez un nombre: 10
  Debut de methodeAttrapeuse
    Debut de methodeIntermediaire
      Debut de methodeLanceuse
      Fin de methodeLanceuse
    Fin de methodeIntermediaire
  Suite de methodeAttrapeuse
  Fin de methodeAttrapeuse
```

```
Fin de main
```

On voit que chaque méthode commence à s'exécuter avant les méthodes qu'elle invoque et termine son exécution après la fin de l'exécution des méthodes invoquées.

Voyons maintenant le cheminement lorsque l'exception est lancée.

```
> java CodeNonExecute
Debut de main
Entrez un nombre: 0
  Debut de methodeAttrapeuse
    Debut de methodeIntermediaire
      Debut de methodeLanceuse
      Exception lancee par methodeLanceuse
  Résolution par methodeAttrapeuse
  Fin de methodeAttrapeuse
Fin de main
```

On voit qu'il y a du code exécuté dans la méthode qui lance l'exception, dans la méthode intermédiaire et dans la méthode qui attrappe l'exception. Il s'agit de la portion de code correspondant aux affichages :

```
      Fin de methodeLanceuse
    Fin de methodeIntermediaire
  Suite de methodeAttrapeuse
```

Ce code n'est pas exécuté et ne le sera pas, même une fois le problème résolu avec le `catch`.

En revanche, la portion de code correspondant au message

```
  Résolution par methodeAttrapeuse
```

n'est exécuté que dans le cas où l'exception lancée puis attrapée. Il faut dans ce morceau de code-là faire toutes les opérations nécessaires pour que le programme reprenne son cours normal à partir de ce point, c'est-à-dire après le `try...catch`.

4 Traiter plusieurs exceptions avec un seul try

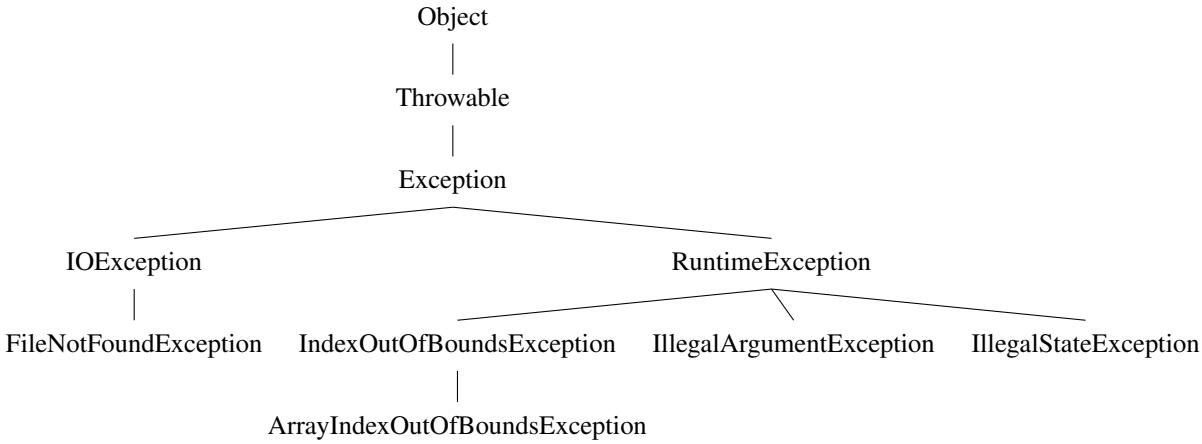
Il est possible de traiter plusieurs exceptions avec un seul `try` doté de plusieurs `catch` à condition qu'elles aient des types différents.

```
try{
   ...
   code qui peut lancer différentes exceptions
   ...
}catch(IllegalArgumentException exc){
   code qui traite les IllegalArgumentException
}catch(IllegalStateException exc){
   code qui traite les IllegalStateException
}catch(NullPointerException exc){
   code qui traite les NullPointerException
}
```

De cette façon, on peut avoir un code spécifique pour chaque type d'erreur. Pour être sûr de ne traiter que des erreurs levées par son propre code, un programmeur peut créer de nouvelles classes d'exceptions, une pour chaque type d'erreur qu'il veut traiter d'une certaine manière. Les nouvelles classes doivent hériter de la classe `Exception` ou d'une de ses sous-classes.

5 Hiérarchie des classes

Quelques classes parmi les plus intéressantes de la hiérarchie des exceptions sont notées dans l'arbre d'héritage suivant.



Parmi ces classes il y a une distinction importante entre `RuntimeException` et ses sous-classes d'un côté et les exceptions qui ne sont pas des descendantes de `RuntimeException`. Cette distinction, c'est qu'une méthode qui lance directement ou indirectement des exceptions autres que les `RuntimeException` doit déclarer ces exceptions dans son entête.

Prenons un exemple concret avec l'exception `FileNotFoundException` qui n'est pas une `RuntimeException`. Voyons un exemple de méthode qui lance cette exception.

```
public void methodeLanceuse(int x){
    System.out.println("␣␣␣␣␣␣Debut␣de␣methodeLanceuse");
    if (x == 0){
        System.out.println("␣␣␣␣␣␣Exception␣lancee␣par␣methodeLanceuse");
        throw new FileNotFoundException();
    }
    System.out.println("␣␣␣␣␣␣Fin␣de␣methodeLanceuse");
}
```

À la compilation, ce code provoque une erreur :

```
FileNotFoundThrown.java:7: error: unreported exception
    FileNotFoundException; must be caught or declared to be thrown
            throw new FileNotFoundException();
```

Ce message dit que l'exception doit être attrapée dans la méthode ou déclarée. Pour la déclarer, on ajoute une clause `throws FileNotFoundException` dans l'entête de la méthode, après la liste des paramètres. Notez que ce `throws` a un s à la fin et que c'est une déclaration à ne pas confondre avec l'instruction `throw` qui est du code à l'intérieur du corps de la méthode et qui sert à lancer effectivement l'exception. Le code devient :

```
public void methodeLanceuse(int x) throws FileNotFoundException{
    System.out.println("␣␣␣␣␣␣Debut␣de␣methodeLanceuse");
    if (x == 0){
        System.out.println("␣␣␣␣␣␣Exception␣lancee␣par␣methodeLanceuse");
        throw new FileNotFoundException();
    }
    System.out.println("␣␣␣␣␣␣Fin␣de␣methodeLanceuse");
}
```

Supposons ensuite qu'une méthode ne lance pas directement l'exception mais qu'elle invoque la méthode `methodeLanceuse` qui est susceptible de lancer l'exception. Dans ce cas, elle aussi doit déclarer qu'elle lance l'exception avec la même clause `throws` faute de quoi il y a une erreur à la compilation (must be caught or declared).

```
public void methodeIntermediaire(int n) throws FileNotFoundException{
    System.out.println("␣␣␣␣Debut␣de␣methodeIntermediaire");
    methodeLanceuse(n);
    System.out.println("␣␣␣␣Fin␣de␣methodeIntermediaire");
}
```

Cette obligation de déclarer les exceptions est supprimée pour la classe `RuntimeException` et ses descendante. C'est la raison pour laquelle nous avons privilégié ce type jusqu'ici. Le plus souvent, nous utiliserons soit `IllegalArgumentException`, soit `IllegalStateException`, qui sont des sous-classes de `RuntimeException`. Et si nous préférons créer nos propres classes, nous écrirons des sous-classes de `IllegalArgumentException` et `IllegalStateException`.

6 Importance de l'ordre des catch

Un objet a plusieurs types : il appartient au type de la classe instanciée, mais également au type de la super-classe de la classe instanciée, et au type de la super-classe de la super-classe et ainsi de suite jusqu'à `Object`.

Par exemple les types d'un objet instance de `IllegalArgumentException` sont `IllegalArgumentException`, `RuntimeException`, `Exception`, `Throwable` et `Object`. Chacun de ces type sauf `Object` peut être utilisé dans un `catch` pour définir quel type d'exception est attrapée par ce catch. Ainsi, une clause `catch` peut rattraper toutes les exceptions : `catch(Exception exc){...}`

Lorsqu'il y a plusieurs `catch` dans un try, ces catch peuvent attraper plusieurs des types d'un même objet. Les catch sont parcourus dans l'ordre d'écriture et le premier qui appartient aux types de l'objet lancé est exécuté. Une exception n'est rattrapée que par un seul `catch`, même s'il y en a plusieurs comportant des types de l'exception.

Comme conséquence, il faut prendre soin de toujours mettre les `catch` les plus précis (types les plus bas dans l'arbre) en premier, les plus généraux (types les plus hauts dans l'arbre) en dernier. Si l'on se trompe dans l'ordre, le compilateur le détecte et cela produit une erreur.

```
try{
    System.out.println("Coco␣l'est␣content");
}catch(RuntimeException exc){
    System.out.println("il␣y␣a␣eu␣une␣RuntimeException");
}catch(IllegalArgumentException exc){
```

```
            System.out.println("il␣y␣a␣eu␣une␣IllegalArgumentException");
        }
```

```
TryCatch.java:7: error: exception IllegalArgumentException has
                                                  already been caught
        }catch(IllegalArgumentException exc){
         ^
1 error
```

Il faut inverser l'ordre des catch.

```
        try{
            System.out.println("Coco␣l'est␣content");
        }catch(IllegalArgumentException exc){
            System.out.println("il␣y␣a␣eu␣une␣IllegalArgumentException");
        }catch(RuntimeException exc){
            System.out.println("il␣y␣a␣eu␣une␣RuntimeException");
        }
```

Ce code est correct et ne produit pas d'erreur de compilation.