Chapitre XI

Exceptions

1 Notion d'exception

Le mécanisme d'exceptions permet de gérer les situations exceptionnelles (comme des erreurs) qui peuvent survenir lors de l'exécution d'un programme. Sans ce mécanisme, on traite les cas exceptionnels par une série de conditionnelles, qui rendent le code peu lisible.

Définition

Une **exception** est un objet Java de la classe `Throwable`, ou de l'une de ses sous-classes.
Une **exception** est levée (ou lancée) lorsque quelque chose d'exceptionnel, comme une erreur, se produit.
Rattraper l'exception consiste à exécuter des actions pour traiter ce cas exceptionnel.
Si l'exception n'est pas rattrapée, le programme Java est interrompu.

2 Lever une exception

Considérons le programme Java suivant :

```
1 class TestExcept {
2     public static void main (String[] args) {
3         int x  = 0;
4         int y = 1/x; // Ici, l'exception java.lang.ArithmeticException est levé
              e
5         // Le programme s'interrompt et le code qui suit n'est pas exécuté
6         System.out.println("Fin du programme");
7     }
8 }
```

Ce code compile. A l'exécution, la division par 0 lève l'exception `java.lang.ArithmeticException`, et la pile des appels de méthode est affichée : c'est la ligne 4 du fichier source `TestExcept.java`, dans la méthode main, qui a levé l'exception.

```
1 Exception in thread "main" java.lang.ArithmeticException: / by zero
2 at TestExcept.main(TestExcept.java:4)
```

3 Propagation des exceptions

Dans le programme suivant, la méthode main appelle la méthode `meth1` (ligne 4), qui appelle la méthode `meth2` (ligne 9). La méthode `meth2` lève l'exception `java.lang.ArithmeticException` ligne 15. A cet

endroit, le flot d'exécution du programme est interrompu. L'exception n'est pas rattrapée au niveau de `meth2`, donc elle est propagée au niveau de `meth1`, où elle n'est pas non plus rattrapée. Elle est donc propagée au niveau de la méthode `main` et le programme s'arrête.

Les lignes 5, 10, 11, 16 et 17 ne sont pas exécutées.

```
1  class TestExcept2 {
2      public static void main(String[] args) {
3          TestExcept2 t = new TestExcept2();
4          int y = t.meth1(0);
5          System.out.println("Ligne non exécutée");
6      }
7
8      int meth1(int x) {
9          int y = meth2(x);
10         System.out.println("Ligne non exécutée");
11         return y;
12     }
13
14     int meth2(int x) {
15         int y = 1/x; // Exception java.lang.ArithmeticException levée
16         System.out.println("Ligne non exécutée");
17         return y;
18     }
19 }
```

Lorsque le programme s'arrête, un message indique que l'exception `java.lang.ArithmeticException` a été levée, ainsi que la pile des méthodes appelées, avec le numéro de ligne correspondant.

```
1  Exception in thread "main" java.lang.ArithmeticException: / by zero
2          at TestExcept2.meth2(TestExcept2.java:15)
3          at TestExcept2.meth1(TestExcept2.java:9)
4          at TestExcept2.main(TestExcept2.java:4)
```

4 Lever explicitement une exception: instruction `throw`

On peut lever explicitement une exception avec l'instruction throw. Dans le programme suivant, la méthode `f(double x)` lève l'exception `IllegalArgumentException` lorsque x vaut 1 ou -1.

```
1  class TestExcept3 {
2      public static void main(String[] args) {
3          TestExcept3 t = new TestExcept3();
4          System.out.println(t.f(1));
5      }
6
7      double f(double x) {
8          if (x == 1 || x == -1) {
9              throw new IllegalArgumentException("Argument incorrect : x = " + x)
                   ; // 9
10         }
11         return 1 / (1 - x * x);
12     }
13 }
```

L'exécution de ce programme affiche :

```
1  Exception in thread "main" java.lang.IllegalArgumentException: Argument
       incorrect : x = 1.0
2          at TestExcept3.f(TestExcept3.java:9)
3          at TestExcept3.main(TestExcept3.java:4)
```

5 Rattraper une exception: bloc `try / catch`

On peut rattraper une exception avec un bloc `try / catch`. Dans le programme ci-dessous, dans la méthode `meth1`, l'appel à la méthode `meth2` est réalisé dans un bloc try (cf. lignes 10 à 14).

```
1  class TestExcept4 {
2      public static void main(String[] args) {
3          TestExcept4 t = new TestExcept4();
4          int y = t.meth1(0);
5          System.out.println("Fin du programme");
6      }
7
8      int meth1(int x) {
9          int y = 0;
10         try {
11             y = meth2(x);
12         } catch (ArithmeticException e) {
13             System.out.println("Exception rattrapée");
14         }
15         return y;
16     }
17
18     int meth2(int x) {
19         int y = 1/x; // Exception java.lang.ArithmeticException levé
20         System.out.println("Ligne non exécutée");
21         return y;
22     }
23 }
```

L'exception `ArithmeticException` est levée dans la méthode `meth2`, ligne 19. Elle n'est pas récupérée au niveau de `meth2` et est donc propagée dans `meth1`. Dans `meth1`, elle est récupérée par le bloc catch, donc le message `"Exception rattrapée"` est affiché. Ensuite la ligne 15 est exécutée, et on sort de `meth1`. Enfin, la ligne 5 est exécutée, le message Fin du programme est affiché, et le programme termine normalement (sans erreur).

Si le bloc `try` peut lever plusieurs exceptions, on peut récupérer chacune d'entre elles avec plusieurs blocs catch à la suite.

```
try {
    // instructions
} catch (Except1 e) {
    // instructions
} catch (Except2 e) {
    // instructions
}
```

A partir de la Java 1.7, on peut également récupérer plusieurs exceptions dans un même bloc catch, ce qui peut être utile pour factoriser du code.

```
try {
    // instructions
} catch (Except1 | Except2 e) {
    // instructions
}
```

On peut ajouter, après un bloc `try` et une suite de blocs `catch`, un bloc `finally`.

```
try {
    // instructions
} catch (Except1 e) {
    // instructions
} catch (Except2 e) {
    // instructions
} finally {
    // instructions
}
```

Les instruction du bloc `finally` sont les dernières instructions exécutées, après l'exécution du bloc try et l'éventuelle exécution d'un bloc catch. Elle sont toujours exécutées, qu'une exception ait été levée ou non, récupérée ou non.

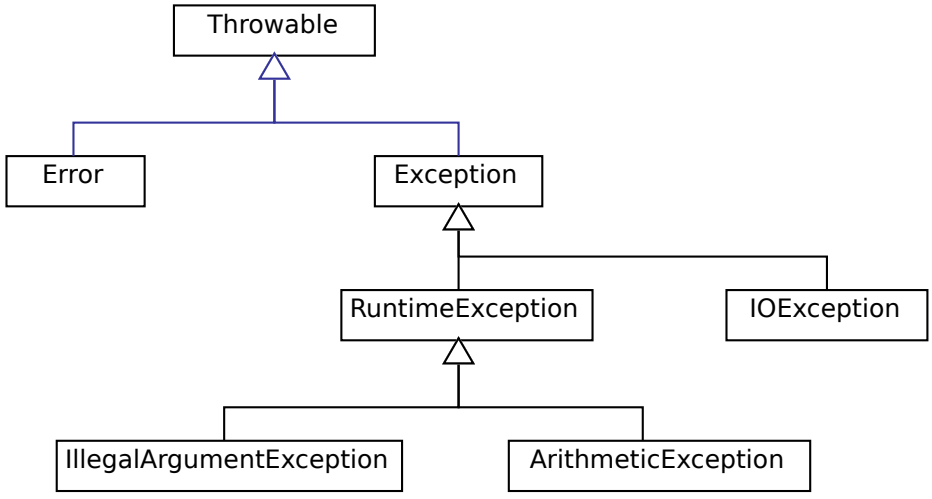
```
1  class TestExcept5 {
2      public static void main(String[] args) {
3          int x = 0;
4          int y = 0;
5          try {
6              y = 1/x; // Exception ArithmeticException levée
7          } catch (IllegalArgumentException e) {
8              System.out.println("On ne passe pas par ici");
9          } finally {
10             System.out.println("Ligne exécutée");
11         }
12     }
13 }
```

Dans ce programme, la ligne 6 lève l'exception `ArithmeticException`. Elle n'est pas récupérée par le bloc catch, qui attend l'exception `IllegalArgumentException`. La ligne 10 est ensuite exécutée, puis le programme s'interrompt. Il a affiché :

```
1  Ligne exécutée
2  Exception in thread "main" java.lang.ArithmeticException: / by zero
3          at TestExcept5.main(TestExcept5.java:6)
```

6 Hiérarchie de classes des exceptions

Une exception est un objet Java, d'une classe qui hérite de la classe `Throwable`. On a une hiérarchie de classes d'exceptions, qui a pour racine la classe `Throwable`. Cette classe `Throwable` a deux sous-classes : `Error` et `Exception`. La classe `Error` est la classe des exceptions "graves", qui ne devraient pas être récupérées (erreurs système, plus de mémoire ... etc.). La classe `Exception` correspond aux exceptions qu'on peut vouloir récupérer. Celle-ci a une sous-classe `RuntimeException`. Les exceptions vues précédemment (`ArithmeticException`, `IllegalArgumentException`) sont des sous-classes de `RuntimeException`.



6.a Exceptions contrôlées / non contrôlées

On distingue les exceptions **contrôlées** (*checked*) et les exceptions **non contrôlées** (*unchecked*). Les exceptions qui héritent de `Error` et `RuntimeException` sont non contrôlées, les exceptions qui héritent de `Exception` (mais pas de `RuntimeException`) sont contrôlées.

6.b Clause throws

Définition

Une méthode doit déclarer les exceptions contrôlée qu'elle peut lever dans la clause `throws`.

Par exemple, la méthode `read` de la classe `java.io.InputStream` peut lever l'exception contrôlée `java.io.IOException`.

```
1 public abstract int read() throws java.io.IOException {
2     // ...
3 }
```

Les exceptions contrôlées qui peuvent être levées lors de l'appel d'une méthode font partie de la spécification de la méthode. Par exemple, toute méthode qui utilise la méthode `read` devra :

- soit déclarer qu'elle peut lever l'exception,
- soit la récupérer.

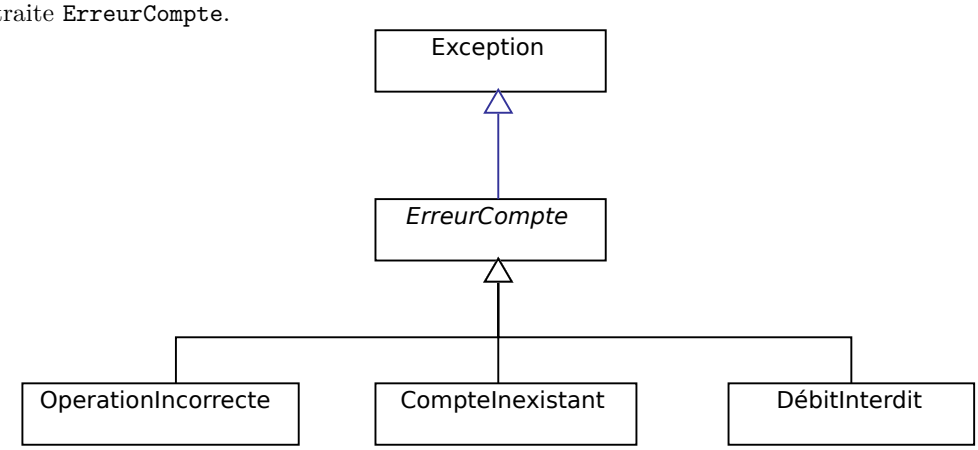
Il est conseillé d'utiliser essentiellement des exceptions contrôlées. De cette façon, on écrit des programmes plus sûrs car lorsqu'on utilise une méthode, on sait quelles exceptions celle-ci peut lever, et les traiter correctement. Dans certains cas, il peut être acceptable d'utiliser des exceptions non contrôlées, en particulier lorsque l'on fait de la programmation défensive : si on détecte une erreur interne, qui n'est pas sensée pouvoir arriver, on peut stopper le programme en levant une exception non contrôlée.

7 Déclaration d'exceptions

L'utilisateur peut déclarer sa propre hiérarchie d'exceptions. Considérons par exemple une application permettant de gérer des comptes bancaires. Plusieurs situations anormales peuvent se produire :

- Accès à un compte inexistant;
- Montant insuffisant pour effectuer un débit;
- Tentative de débiter une somme négative.

On peut donc définir une exception pour chacun de ces cas : classes `CompteInexistant`, `DebitInterdit`, et `OperationIncorrecte`. On définit également une super-classe pour ces trois classes : la classe abstraite `ErreurCompte`.



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

class CompteInexistant extends ErreurCompte {
    public CompteInexistant(String s) {
        super(s);
    }
}

class DebitInterdit extends ErreurCompte {
    public DebitInterdit(String s) {
       super(s);
    }
}

class OperationIncorrecte extends ErreurCompte {
    public OperationIncorrecte(String s) {
       super(s);
    }
}
```

Dans chacune de ces classes, on définit un constructeur prenant une String en paramètre, de façon à pouvoir associer à chaque exception un message d'erreur.

On définit une classe `Compte`, avec un numéro de compte, un solde et un découvert autorisé.

On définit une méthode débiter, qui peut lever les exceptions `OperationIncorrecte` et `DebitInterdit`.

```
class Compte {

    private int noCompte;
    private int solde;
    private int decouvertAutorise;

    Compte(int noCompte, int solde, int decouvertAutorise) {
        this.noCompte = noCompte;
        this.solde = solde;
        this.decouvertAutorise = decouvertAutorise;
    }

    public int getNoCompte() {
        return noCompte;
    }

    public void debiter(int somme) throws OperationIncorrecte, DebitInterdit {
        if (somme < 0) {
            throw new OperationIncorrecte("debit : somme négative");
        }
        if (solde - somme < decouvertAutorise) {
            throw new DebitInterdit("debit : somme insuffisnate sur le compte")
                ;
        }
        solde = solde - somme;
    }
}
```

On peut enfin définir une classe `Banque`, qui permet de gérer un ensemble de comptes bancaires.

```
class Banque {
    Map lesComptes = new HashMap();
    int compteur = 1;

```

```
    public void creerCompte(int solde, int decouvertAutorise) {
        Compte c = new Compte(compteur, solde, decouvertAutorise);
        compteur++;
        lesComptes.put(c.getNoCompte(), c);
    }

    public Compte getAccount(int noCompte) throws CompteInexistant {
        if (!lesComptes.containsKey(noCompte)) {
            throw new CompteInexistant("No de compte incorrect");
        } else {
            return lesComptes.get(noCompte);
        }
    }
```

La méthode `getAccount` permet de retrouver un compte ayant un certain numéro. Elle lève l'exception `CompteInexistant` en cas de numéro de compte incorrect.

On peut ensuite définir une méthode `transferer`, qui permet d'effectuer un transfert d'argent entre deux comptes.

```
public void transferer(int noCompteSource, int noCompteDest, int somme) {
    try {
        Compte source = getAccount(noCompteSource);
        Compte dest = getAccount(noCompteDest);
        source.debiter(somme);
        dest.crediter(somme);
    } catch (ErreurCompte e) {
        System.out.println("Transfert impossible");
    }
}
```

Ici, on a fait le choix que la méthode `transferer` ne lève pas d'exception : il faut donc récupérer les exceptions susceptibles d'être levées. On peut faire le choix de récupérer avec ErreurCompte les trois exceptions qui peuvent être levées.

On peut également vouloir être plus précis dans l'affichage du message et récupérer les trois exceptions indépendamment :

```
public void transferer(int noCompteSource, int noCompteDest, int somme) {
    try {
        Compte source = getAccount(noCompteSource);
        Compte dest = getAccount(noCompteDest);
        source.debiter(somme);
        dest.crediter(somme);
    } catch (CompteInexistant e) {
        // ...
    } catch (DebitInterdit e) {
        // ...
    } catch (OperationIncorrecte e) {
        // ...
    }
}
```

On aurait pu également récupérer uniquement l'exception OperationIncorrecte, puis toutes les autres en même temps, avec `ErreurCompte`.

```
public void transferer(int noCompteSource, int noCompteDest, int somme) {
    try {
        Compte source = getAccount(noCompteSource);
        Compte dest = getAccount(noCompteDest);
        source.debiter(somme);
        dest.crediter(somme);
    } catch (OperationIncorrecte e) {
        // ...
    } catch (ErreurCompte e) {
        // ...
```

```
    }
}
```

On peut remarquer qu'il n'aurait pas été possible d'échanger les deux exceptions, en écrivant :

```
public void transferer(int noCompteSource, int noCompteDest, int somme) {
    try {
        Compte source = getAccount(noCompteSource);
        Compte dest = getAccount(noCompteDest);
        source.debiter(somme);
        dest.crediter(somme);
    } catch (ErreurCompte e) {
        // ...
    } catch (OperationIncorrecte e) {
        // ...
    }
}
```

En effet, dans ce cas, il se produit une erreur à la compilation car la bloc catch correspondant à l'exception `OperationIncorrecte` ne pourrait jamais être atteint.