

Chapitre 8

Les exceptions

8.1 Introduction : qu'est-ce qu'une exception ?

De nombreux langages de programmation de haut niveau possèdent un mécanisme permettant de gérer les erreurs qui peuvent intervenir lors de l'exécution d'un programme. Le mécanisme de gestion d'erreur le plus répandu est celui des exceptions. Nous avons déjà abordé le concept d'exception dans le cours sur les fonctions : lorsqu'une fonction n'est pas définie pour certaines valeur de ses arguments on lève une exception en utilisant le mot clef `throw`. Par exemple, la fonction factorielle n'est pas définie pour les nombres négatifs, et pour ces cas, on lève une exception :

Listing 8.1 – (lien vers le code brut)

```
1 class Factorielle {
2     static int factorielle(int n){
3         int res = 1;
4         if (n<0){
5             throw new PasDefini();
6         }
7         for(int i = 1; i <= n; i++) {
8             res = res * i;
9         }
10        return res;
11    }
12 }
13
14 class PasDefini extends Error {}
```

Une exception signale une erreur comme lorsqu'un nombre négatif est passé en argument à la fonction factorielle. Jusqu'ici, lever une exception signifiait interrompre définitivement le programme avec l'affichage d'un message d'erreur décrivant l'exception à l'écran. Cependant, il est de nombreuses situations où le programmeur aimerait gérer les erreurs sans que le programme ne s'arrête définitivement. Il est alors important de pouvoir intervenir dans le cas où une exception a été levée. Les langages qui utilisent les exceptions possèdent toujours une construction syntaxique permettant de "rattraper" (ou "récupérer") une exception, et d'exécuter un morceau de code spécifique à ce traitement d'erreur.

Examinons maintenant comment définir, lever et récupérer une exception en Java.

8.2 Définir des exceptions

Afin de définir une nouvelle sorte d'exception, on crée une nouvelle classe en utilisant une déclaration de la forme suivante :

```
class NouvelleException extends ExceptionDejaDefinie {}
```

On peut remarquer ici la présence du mot clé **extends**, dont nous verrons la signification dans un chapitre ultérieur qui traitera de l'héritage entre classes. Dans cette construction, `NouvelleException` est le nom de la classe d'exception que l'on désire définir en "étendant" `ExceptionDejaDefinie` qui est une classe d'exception déjà définie. Sachant que `Error` est prédéfinie en Java, la déclaration suivante définit la nouvelle classe d'exception `PasDefini` :

```
class PasDefini extends Error {}
```

Il existe de nombreuses classes d'exceptions prédéfinies en Java, que l'on peut classer en trois catégories :

- Celles définies par extension de la classe `Error` : elles représentent des erreurs critiques qui ne sont pas censées être gérées en temps normal. Par exemple, une exception de type `OutOfMemoryError` est levée lorsqu'il n'y a plus de mémoire disponible dans le système. Comme elles correspondent à des erreurs critiques elles ne sont pas normalement censées être récupérées et nous verrons plus tard que cela permet certaines simplifications dans l'écriture de méthodes pouvant lever cette exception.
- Celles définies par extension de la classe `Exception` : elles représentent les erreurs qui doivent normalement être gérées par le programme. Par exemple, une exception de type `IOException` est levée en cas d'erreur lors d'une entrée sortie.
- Celles définies par extension de la classe `RuntimeException` : elles représentent des erreurs pouvant éventuellement être gérées par le programme. L'exemple typique de ce genre d'exception est `NullPointerException`, qui est levée si l'on tente d'accéder au contenu d'un tableau qui a été déclaré mais pas encore créé par un `new`.

Chaque nouvelle exception entre ainsi dans l'une de ces trois catégories. Si on suit rigoureusement ce classement (et que l'on fait fi des simplifications évoquées plus haut), l'exception `PasDefini` aurait dû être déclarée par :

```
class PasDefini extends Exception {}
```

ou bien par :

```
class PasDefini extends RuntimeException {}
```

car elle ne constitue pas une erreur critique.

8.3 Lever une exception

Lorsque l'on veut lever une exception, on utilise le mot clé `throw` suivi de l'exception à lever, qu'il faut avoir créée auparavant avec la construction `new NomException()`. Ainsi lancer une exception de la classe `PasDefini` s'écrit :

```
throw new PasDefini();
```

Lorsqu'une exception est levée, l'exécution normale du programme s'arrête et on saute toutes les instructions jusqu'à ce que l'exception soit rattrapée ou jusqu'à ce que l'on sorte du programme. Par exemple, si on considère le code suivant :

Listing 8.2 – (lien vers le code brut)

```

1 public class Arret {
2     public static void main(String [] args) {
3         int x = Terminal.lireInt();
4         Terminal.ecrireStringln("Coucou_1");           // 1
5         if (x > 0){
6             throw new Stop();
7         }
8         Terminal.ecrireStringln("Coucou_2");           // 2
9         Terminal.ecrireStringln("Coucou_3");           // 3
10        Terminal.ecrireStringln("Coucou_4");           // 4
11    }
12 }
13 class Stop extends RuntimeException {}

```

l'exécution de la commande `java Arret` puis la saisie de la valeur 5 (pour la variable `x`) produira l'affichage suivant :

```

Coucou 1
Exception in thread "main" Stop
    at Arret.main(Arret.java:7)

```

C'est-à-dire que les instructions 2, 3 et 4 n'ont pas été exécutées. Le programme se termine en indiquant que l'exception `Stop` lancée dans la méthode `main` à la ligne 7 du fichier `Arret.java` n'a pas été rattrapée.

8.4 Rattraper une exception

8.4.1 La construction `try catch`

Le rattrapage d'une exception en Java se fait en utilisant la construction :

Listing 8.3 – (pas de lien)

```

1 try {
2     ... // 1
3 } catch (UneException e) {
4     ... // 2
5 }
6 .. // 3

```

Le code 1 est normalement exécuté. Si une exception est levée lors de cette exécution, les instructions restantes dans le code 1 sont abandonnées. Si la classe de l'exception levée dans le bloc 1 est `UneException` alors le code 2 est exécuté (car l'exception est récupérée). Dans le code 2, on peut faire référence à l'exception en utilisant le nom donné à celle-ci lorsque l'on nomme sa classe. Ici le nom est `e`. Dans le cas où la classe de l'exception n'est pas `UneException`, le code 2 et le code 3 sont sautés. Ainsi, le programme suivant :

Listing 8.4 – (lien vers le code brut)

```

1 public class Arret2 {
2     public static void P () {
3         int x = Terminal.lireInt ();
4
5         if (x >0){
6             throw new Stop ();
7         }
8     }
9     public static void main(String [] args) {
10        Terminal.ecrireStringln("Coucou_1");           // 1
11
12        try {
13            P ();
14            Terminal.ecrireStringln("Coucou_2");        // 2
15        } catch (Stop e){
16            Terminal.ecrireStringln("Coucou_3");        // 3
17        }
18        Terminal.ecrireStringln("Coucou_4");           // 4
19    }
20 }
21 class Stop extends RuntimeException {}

```

produit l’affichage suivant lorsqu’il est exécuté et que l’on saisit une valeur positive :

```

Coucou 1
Coucou 3
Coucou 4

```

On remarquera que l’instruction 2 n’est pas exécuté (du fait de la levée de l’exception dans P.

Si on exécute ce même programme mais en saisissant une valeur négative on obtient :

```

Coucou 1
Coucou 2
Coucou 4

```

car l’exception n’est pas levée.

En revanche le programme suivant, dans lequel on lève une exception `Stop2`, qui n’est pas récupérée.

Listing 8.5 – (lien vers le code brut)

```

1 public class Arret3 {
2     public static void P () {
3         int x = Terminal.lireInt ();
4
5         if (x >0){
6             throw new Stop2 ();
7         }
8     }
9     public static void main(String [] args) {
10        Terminal.ecrireStringln("Coucou_1");           // 1
11        try {

```

```

12         P ();
13         Terminal. écrireStringln ("Coucou_2");    // 2
14     } catch (Stop e){
15         Terminal. écrireStringln ("Coucou_3");    // 3
16     }
17     Terminal. écrireStringln ("Coucou_4");        // 4
18 }
19 }
20 class Stop extends RuntimeException {}
21 class Stop2 extends RuntimeException {}

```

produit l’affichage suivant lorsqu’il est exécuté et que l’on saisit une valeur positive :

```

Coucou 1
Exception in thread "main" Stop2
    at Arret3.P(Arret3.java:7)
    at Arret3.main(Arret3.java:15)

```

8.4.2 Rattraper plusieurs exceptions

Il est possible de rattraper plusieurs types d’exceptions en enchaînant les constructions `catch` :

Listing 8.6 – (lien vers le code brut)

```

1 public class Arret3 {
2     public static void P () {
3         int x = Terminal. lireInt ();
4
5         if (x >0){
6             throw new Stop2 ();
7         }
8     }
9     public static void main(String [] args) {
10        Terminal. écrireStringln ("Coucou_1");    // 1
11        try {
12            P ();
13            Terminal. écrireStringln ("Coucou_2");    // 2
14        } catch (Stop e){
15            Terminal. écrireStringln ("Coucou_3");    // 3
16        }
17        } catch (Stop2 e){
18            Terminal. écrireStringln ("Coucou_3_bis"); // 3 bis
19        }
20        Terminal. écrireStringln ("Coucou_4");        // 4
21    }
22 }
23 class Stop extends RuntimeException {}
24 class Stop2 extends RuntimeException {}

```

A l’exécution, on obtient (en saisissant une valeur positive) :

```

Coucou 1
Coucou 3 bis
Coucou 4

```

8.5 Exceptions et méthodes

8.5.1 Exception non rattrapée dans le corps d’une méthode

Comme on l’a vu dans les exemples précédents, lorsqu’une exception est levée lors de l’exécution d’une méthode et qu’elle n’est pas rattrapée dans cette méthode, elle “continue son trajet” à partir de l’appel de la méthode. Même si la méthode est sensée renvoyer une valeur, elle ne le fait pas :

Listing 8.7 – (lien vers le code brut)

```

1 public class Arret {
2     static int lance(int x) {
3         if (x < 0) {
4             throw new Stop();
5         }
6         return x;
7     }
8     public static void main(String [] args) {
9         int y = 0;
10        try {
11            Terminal.afficheStringln("Coucou_1");
12            y = lance(-2);
13            Terminal.afficheStringln("Coucou_2");
14        } catch (Stop e) {
15            Terminal.afficheStringln("Coucou_3");
16        }
17        Terminal.afficheStringln("y_vaut_" + y);
18    }
19 }
20 class Stop extends RuntimeException {}

```

A l’exécution on obtient :

```

Coucou 1
Coucou 3
y vaut 0

```

8.5.2 Déclaration throws

Lorsqu’une méthode lève une exception définie par extension de la classe `Exception` il est nécessaire de préciser au niveau de la déclaration de la méthode qu’elle peut potentiellement lever une exception de cette classe. Cette déclaration prend la forme **throws** `Exception1`, `Exception2`, ... et se place entre les arguments de la méthode et l’accolade ouvrant marquant le début du corps de la méthode. On notera que cette déclaration n’est pas obligatoire pour les exceptions de la catégorie `Error` ni pour celles de la catégorie `RuntimeException`.

8.6 Exemple résumé

On reprend l’exemple de la fonction factorielle :

Listing 8.8 – (lien vers le code brut)

```

1  class Factorielle {
2      static int factorielle(int n) throws PasDefini { // (1)
3          int res = 1;
4          if (n<0){
5              throw new PasDefini(); // (2)
6          }
7          for(int i = 1; i <= n; i++) {
8              res = res * i;
9          }
10         return res;
11     }
12     public static void main (String [] args) {
13         int x;
14         Terminal.ecrireString("Entrez un nombre (petit):");
15         x = Terminal.lireInt();
16         try { // (3)
17             Terminal.ecrireIntln(factorielle(x));
18         } catch (PasDefini e) { // (3 bis)
19             Terminal.ecrireStringln("La factorielle de "
20                                     +x+" n'est pas définie!");
21         }
22     }
23 }
24 class PasDefini extends Exception {} // (4)

```

Dans ce programme, on définit une nouvelle classe d'exception `PasDefini` au point (4). Cette exception est levée par l'instruction `throw` au point (2) lorsque l'argument de la méthode est négatif. Dans ce cas l'exception n'est pas rattrapée dans le corps et comme elle n'est ni dans la catégorie `Error` ni dans la catégorie `RuntimeException`, on la déclare comme pouvant être levée par `factorielle`, en utilisant la déclaration `throws` au point (1). Si l'exception `PasDefini` est levée lors de l'appel à `factorielle`, elle est rattrapée au niveau de la construction `try catch` des points (3) (3 bis) et un message indiquant que la factorielle du nombre entré n'est pas définie est alors affiché. Voici deux exécutions du programme avec des valeurs différentes pour `x` (l'exception est levée puis rattrapée lors de la deuxième exécution) :

```

> java Factorielle
Entrez un nombre (petit):4
24
> java Factorielle
Entrez un nombre (petit):-3
La factorielle de -3 n'est pas définie !

```

Annexe : quelques exceptions prédéfinies

Voici quelques exceptions prédéfinies dans Java :

- `NullPointerException` : utilisation de `length` ou accès à un case d'un tableau valant `null` (c'est à dire non encore créé par un `new`).
- `ArrayIndexOutOfBoundsException` : accès à une case inexistante dans un tableau.
- `ArrayIndexOutOfBoundsException` : accès au i^{eme} caractère d'un chaîne de caractères de taille inférieure à i .

- `ArrayIndexOutOfBoundsException` : création d'un tableau de taille négative.
- `NumberFormatException` : erreur lors de la conversion d'un chaîne de caractères en nombre.

La classe `Terminal` utilise également l'exception `TerminalException` pour signaler des erreurs.