Chapitre 5

Exceptions

Thibaud Martinez

thibaud.martinez@dauphine.psl.eu

Dauphine | PSL
UNIVERSITÉ PARIS

Définition

Une exception est un événement qui se produit pendant l'exécution d'un programme et qui perturbe le déroulement normal des instructions du programme.

```
int x = 1 / 0;  // Exception java.lang.ArithmeticException: / by zero
```

Par exemple,

- une erreur logique;
- un problème matériel;
- une entrée utilisateur inattendue;
- un manque de place sur le disque.

Fonctionnement

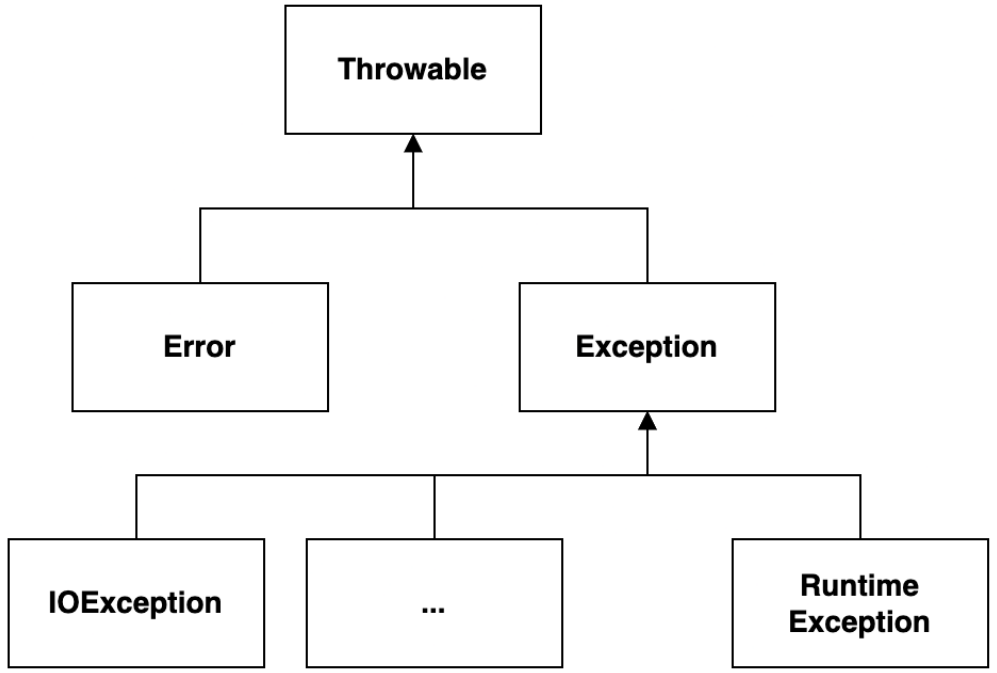
Chaque méthode à une voie de sortie alternative si elle n'est pas en mesure d'accomplir sa tâche de manière normale.

- La méthode **lève une exception** : elle un crée un **objet** exception qui contient les informations relatives à l'erreur et le transmet au système d'exécution.
- Elle **ne renvoie pas de valeur** et son exécution est **interrompue**.
- Le mécanisme de gestion des exceptions commence à rechercher un **gestionnaire d'exception** capable de traiter cette erreur.

Hiérarchie des exceptions

Un objet exception est une instance d'une classe dérivée de `Throwable`.

- `Error` : extérieures à l'application et que l'application ne peut pas anticiper ou gérer.
- `Exception` :
  - `RuntimeException` : généralement causée par une erreur de programmation.
  - Toute autre exception qu'une application bien écrite doit anticiper et gérer.



Checked et *unchecked exceptions*

- ***unchecked exceptions*** : toute exception qui dérive de la classe `Error` ou de la classe `RuntimeException` ;
- ***checked exceptions*** : toutes les autres exceptions.

Les *checked exceptions* doivent obligatoirement être **gérées** par le programmeur dans le code. Le compilateur Java vérifie que c'est bien le cas.

Déclarer des *checked exceptions*

Une méthode n'indique pas seulement au compilateur Java les valeurs qu'elle peut renvoyer, elle lui indique également les exceptions qu'elle peut générer, avec le mot-clé `throws`.

```
public void leveUneException() throws Exception {}
public void levePlusieursException() throws FileNotFoundException, UnknownHostException {}
```

⚠️ La méthode doit déclarer toutes les *checked exceptions* qu'elle pourrait lever.

Lever une exception

On crée un objet exception avec `new` et on lève l'exception avec `throw`.

```
public boolean verifier(String utilisateur) throws Exception {
    if (utilisateur.equals("Bob")) {
        throw new Exception(
            "Je ne peux pas vérifier " +
            "les autorisations de Bob"
        );
    }

    return true;
}
```

Classes d'exceptions existantes

Il existe un certain nombre de classes d'exceptions prédéfinies dans la bibliothèque standard du langage.

- `CloneNotSupportedException`
- `InterruptedException`
- `ReflectiveOperationException`
- `RuntimeException`
  - `ArithmeticException`
  - `IllegalArgumentException`
  - `IndexOutOfBoundsException`
  - `NullPointerException`
  - ...
- ...

Créer une classe d'exception

On peut rencontrer un problème qui n'est pas décrit de manière adéquate par l'une des classes d'exception standard. Dans ce cas, on peut créer sa propre classe d'exception. Il suffit d'hériter d'`Exception`, ou d'une sous-classe d'`Exception`, telle que `IOException`.

```
class FileFormatException extends IOException {
    public FileFormatException() {}
    public FileFormatException(String message) {
        super(message);
    }
}
```

Intercepter une exception

Intercepter une exception permet d'exécuter certaines instructions en réaction à une exception qui a été levée.

```
try {
    // code pouvant lever une exception
} catch (<ExceptionType> <nom de la variable exception>) {
    // gestionnaire d'exceptions
}
```

```
try {
    int y = 5 / x;
} catch (ArithmeticException e) {
    System.out.println("Vous ne pouvez pas diviser par zéro!");
}
```

Intercepter de multiples exceptions

```
try {
    // code qui peut lever des exceptions
} catch (FileNotFoundException e) {
    // action lorsque des fichiers sont manquants
} catch (UnknownHostException e) {
    // action lorsque des hôtes sont inconnus
} catch (IOException e) {
    // action pour les problèmes d'entrées/sorties
}
```

⚠️ Les clauses `catch` sont testées de haut en bas.

```
try {
    // code qui peut lever des exceptions
} catch (FileNotFoundException | UnknownHostException e) {
    // action lorsque des fichiers sont manquants ou lorsque des hôtes sont inconnus
}
```

finally

Le code de la clause `finally` s'exécute, qu'une exception ait été détectée ou non.

Dans l'exemple suivant, le programme fermera le flux d'entrée en toutes circonstances.

```
var in = new FileInputStream(. . .);

try {
    // code qui peut lever des exceptions
} catch (IOException e) {
    // gère les problème d'entrées/sorties
} finally {
    in.close();
}
```

Lorsque le code lève une exception, il arrête de traiter le code restant dans la méthode et quitte la méthode.

💥 Sans clause `finally`, si le code dans `try` lève une exception autre que de `IOException`, le flux d'entrée ne sera pas fermé.

```
var in = new FileInputStream(. . .);

try {
    // code qui lève NullPointerException
} catch (IOException e) {
    // ...
}

in.close(); // cette instruction n'est jamais exécutée
```

On peut utiliser `finally` sans clause `catch`.

```
InputStream in = . . .;

try {
    // code qui peut lever des exceptions
} finally {
    in.close();
}
```

Où traiter une exception ?

Supposons qu'on ait le code suivant.

```
class MonException extends Exception {}

public static void leveMonException() throws MonException {
    throw new MonException();
}
```

Lorsqu'on utilise du code qui est susceptible de lever une *checked exception*, on peut :

- gérer immédiatement l'exception dans un bloc `try` / `catch` ;

```
try {
    leveMonException();
} catch (MonException e) {
    System.out.println("Quelque chose s'est mal passé...");
}
```

- propager l'exception.

```
public static void propage() throws MonException {
    System.out.println("Propagation de l'exception");
    leveMonException();
}
```

✨ Pratiques recommandées

- Ne pas utiliser directement la classe `Exception`, utiliser des classes dérivées de celle-ci.
- Définir uniquement des *checked exceptions* en héritant de `Exception`, ne pas hériter de `Error` ou de `RuntimeException`.
- Dans la plupart des cas, il est superflu de déclarer les *unchecked exceptions* lors de la déclaration de la fonction.

```
// pas besoin de déclarer NullPointerException
public void method() throws NullPointerException {}
```