POGL 2020 – Exceptions

1 Flot de contrôle et pile d'appels

Flot de contrôle

Lors de l'exécution d'un programme, chaque instruction est visitée zéro, une ou plusieurs fois, selon un ordre qui dépend du *flot de contrôle* de ce programme. Quelques règles sont :

- Si deux instructions sont en séquence, séparées par `;`, on passe de l'une à l'autre.
- Lors d'un appel de méthode, on *saute* au début du code de cette méthode.
- À la fin de l'exécution d'une méthode, on revient au point où la méthode avait été appelée.
- En cas de branchement (`if`), on saute au début du code de la branche choisie.
- À la fin d'un tour de boucle, on revient au test du début de la boucle.

```
 1   public static void main(String[] args) {
 2     int k   = 4;
 3     int res = f(k);
 4     int k   = 2;
 5     int res = f(k);
 6     System.out.println(res);
 7   }
 8   public static int g(int a, int b) {
 9     int res = a / b;
10     return res;
11   }
12   public static int f(int n) {
13     int res = g(n, n-2);
14     return res;
15   }
```

Dans cet exemple, l'exécution passe par la séquence de lignes `2-3-13-9-10-14-4-5-13-9`. Le processus s'arrête ensuite à cet endroit car une erreur va survenir à la ligne `9` (division par zéro) et interrompre l'exécution du programme.

Pile d'appels

À chaque étape de la séquence précédente, nous sommes en train d'exécuter l'appel à une certaine méthode, qui peut-lui même avoir été déclenché par un autre appel de méthode. La *pile d'appels* enregistre ces informations sur le contexte de l'exécution : elle énumère l'ensemble des appels de méthode dont l'exécution est en cours, en plaçant le plus récent (ou le plus local) au sommet et le plus ancien (ou le plus global) au fond.

2	3	13	9	10	14	4	5	13	9
			g:9	g:10					g:9
		f:13	f:13	f:13	f:14			f:13	f:13
m:2	m:3	m:3	m:3	m:3	m:3	m:4	m:5	m:5	m:5

Au moment où l'erreur de division par zéro survient dans notre programme, la machine virtuelle Java nous indique donc une erreur (`java.lang.ArithmeticException: / by zero`), et précise que l'erreur est apparue lors de l'exécution de la ligne `9` pendant l'appel de la méthode `g`, appel déclenché à la ligne `13` lors de l'exécution d'un appel de la méthode `f`, lui-même déclenché à la ligne `5` lors de l'exécution de la méthode `main`.

```
Exception in thread "main" java.lang.ArithmeticException: / by zero
        at Exemple.g(Exemple.java:9)
        at Exemple.f(Exemple.java:13)
        at Exemple.main(Exemple.java:5)
```

Rattraper les interruptions

On peut insérer dans le code des mécanismes de rattrapage, qui indiquent le comprtement à adopter en cas d'erreur (au lieu de simplement arrêter l'exécution d'un programme). Ce mécanisme prend la forme d'une instruction `try`, couplée à une ou plusieurs instructions `catch`.

— `try { ... }` introduit un bloc de code qui peut produire des erreurs.

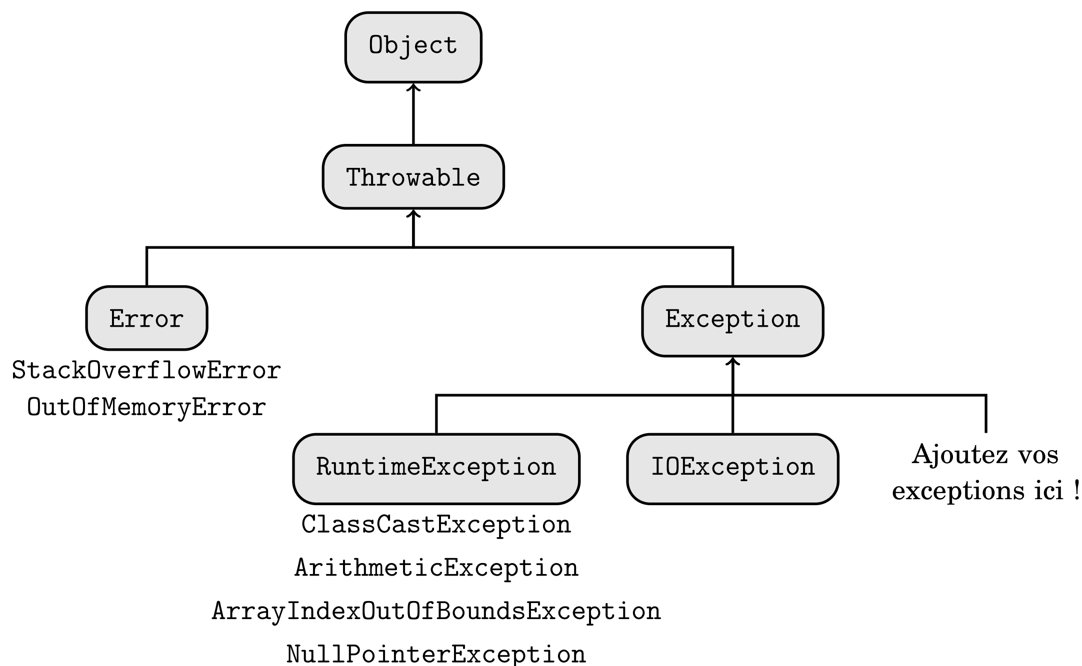
— `catch (E e) { ... }` introduit un bloc de code à exécuter en cas d'occurrence d'une erreur de type `E` dans le bloc `try`.

```
 1   public static void main(String[] args) {
 2     int res;
 3     try {
 4       res = g(2, 0);
 5       System.out.println("Pas d'erreur !");
 6     }
 7     catch (java.lang.ClassCastException e) {
 8       res = -1;
 9     }
10     catch (java.lang.ArithmeticException e) {
11       res = 0;
12     }
13     System.out.println(res);
14   }
15   public static int g(int a, int b) {
16     int res = a / b;
17     return res;
18   }
```

La séquence des lignes visitées est maintenant `2-4-16-11-13`. Comme précédemment, l'instruction `return` après l'erreur (ligne `17`) n'est pas exécutée, et la ligne suivant l'appel de méthode erroné (ligne `5`) non plus. En revanche, l'erreur est advenue à l'intérieur d'un bloc `try`, et peut donc être rattrapée dans l'un des deux blocs `catch` associés. Le premier est ignoré, car l'erreur n'est pas un problème de transtypage (`java.lang.ClassCastException`), et la ligne `8` n'est donc pas visité. Au contraire, la ligne `catch (java.lang.ArithmeticException e)` intercepte l'erreur de division, et l'exécution reprend donc au point `11`. Après l'exécution sans erreur du bloc `try` ou, en cas d'erreur rattrapée, après l'exécution sans erreur du bloc `catch` correspondant, l'exécution reprend normalement au niveau du code qui suit les blocs `catch` (ici à la ligne `13`) et se poursuit jusqu'à la fin du programme. Si l'erreur n'avait pas été rattrapée par l'un des blocs `catch`, alors comme dans la partie précédente l'exécution aurait été simplement interrompue (le cas échéant, après vérification des autres paires `try/catch` présentes dans la pile d'appels).

2 La hiérarchie des exceptions

En Java, tout est objet. En particulier, les erreurs observées dans la partie précédente, qu'on appelle en réalité des *exceptions*, sont des objets comme les autres, qui appartiennent à des classes. Ces classes sont inscrites dans une hiérarchie de sous-classes, et on peut en définir de nouvelles en utilisant l'héritage.



Tous les objets de la classe `Throwable`, et donc toutes les exceptions que nous pouvons définir, possèdent un certain nombre de champs et de méthodes, qui nous renseignent sur l'erreur qui a eu lieu. On peut par exemple récupérer un message associé à l'exception (`public String getMessage()`) ou obtenir des indications sur l'état de la pile d'appels. Comme `Throwable` hérite de `Object`, nous avons également une méthode `public String toString()`. Enfin, la classe `Throwable` possède deux constructeurs :

— `public Throwable()` crée une exception sans message associé.
— `public Throwable(String s)` crée une exception avec le message `s`.

Par les mécanismes d'héritage habituels, toutes les classes d'exceptions possèdent aux moins ces deux constructeurs.

Les exceptions appartenant à des sous-classes de `Error` ou de `RuntimeException` correspondent en général à des erreurs de programmation. Les `Error`s sont liées à l'environnement d'exécution (dont, par exemple, on aura saturé la mémoire avec une chaîne sans fin d'appels récursifs d'une méthode), tandis que les `RuntimeException`s sont déclenchées par des opérations illicites (mauvais transtypage, opérations sur un pointeur nul, accès en dehors d'un tableau, etc.). Ces exceptions peuvent survenir dans n'importe quel programme, et sont dites pour cela *hors contrôle* (*unchecked*).

Toutes les autres exceptions sont dites *sous contrôle* (*checked*). Les autres exceptions sont en général liées à une application ou une classe particulière, et correspondent à des appels de méthode avec de mauvais arguments ou dans un contexte inadapté. Par exemple : tentative d'ouverture d'un fichier inexistant, ou initialisation d'une classe avec des paramètres illégitimes. Java demande que toutes les exceptions sous contrôle pouvant être levées par une méthode donnée soient explicitement citées dans la signature de cette méthode avec le mot-clé `throws` (voir section suivante).

3 Définir, déclarer, et lever des exceptions

Définissons une classe `Nat` des entiers naturels, représentés par des entiers de type `Integer` supérieurs ou égaux à `0`.

```
1   public class Nat {
2     private Integer nombre;
3     public  Integer getNombre()          { return this.nombre; }
4     public  void    setNombre(Integer n) { this.nombre = n;    }
5     public          Nat(Integer n)       { this.setNombre(n);  }
6   }
```

Introduction d'une exception

Nous voulons interdire la création d'un objet de class `Nat` dont le `nombre` soit négatif. Nous allons donc créer une classe d'exceptions `NatException`, qui hérite de `Exception`, et faire qu'une telle exception soit levée lorsque l'on tente d'affecter un nombre négatif à un `Nat`.

```
1   class NatException extends Exception {}
2   public class Nat {
3     ...
4     public void setNombre(Integer n) throws NatException {
5       if (n<0) { throw new NatException(); }
6       else     { this.nombre = 0; }
7     }
8     public Nat(Integer n) throws NatException { this.setNombre(n); }
9   }
```

Quatre points à noter ici :

- Ligne `1` : la class `NatException` hérite de la class `Exception` sans rien y ajouter : nos exceptions ne porteront pas d'information particulière, mais pourront être levées et rattrapées comme toutes les autres.
- Ligne 5 : `NatException` est une classe. Une exception que l'on lève est un objet de cette classe. Ainsi, `new NatException()` crée un object exception, et l'exception est levée par `throw`. L'instruction `throw new NatException()` aurait pu être découpée en deux instructions, une pour créer une exception `e` et une pour la lever :
  ```
  NatException e = new NatException();
  throw e;
  ```
- Ligne 4 : la mention `throws NatException` indique que la méthode `setNombre(Integer)` est susceptible de lever des exceptions (sous contrôle) de type `NatException`. Cette mention est obligatoire.
- Ligne 8 : le constructeur `Nat` ne lève pas lui même d'exceptions, mais il fait appel à la méthode `setNombre(Integer)` qui peut le faire. Le constructeur `Nat(Integer)` est donc susceptible de lever indirectement des exceptions de type `NatException`, ce qui doit être signalé par `throws`. C'est encore obligatoire.

Déclarer ou rattraper

Quand une méthode peut lever une exception (directement ou indirectement), la règle est de déclarer ceci avec `throws`. Cependant, ceci ne concerne pas les exceptions qui sont déjà *rattrapées* par notre méthode avec une construction `try/catch`. Seules doivent être déclarées les exceptions qui peuvent "sortir" de notre méthode. La métaphore : si votre petit frère a cassé un vase sous votre surveillance, mais que vous avez pu réparer ou remplacer l'objet avant le retour des parents, pas besoin de leur raconter.

Par exemple, on peut imaginer que le constructeur `Nat(Integer)` affecte automatiquement le nombre `0` lors d'une tentative de création erronée, et empêche les exceptions de type `NatException` de se manifester.

```
 1   public Nat(Integer n) {
 2     try   { this.setNombre(n); }
 3     catch (NatException e) { this.nombre = 0; }
 4   }
```

Dans ce cas, une exception de type `NatException` déclenchée par l'appel à la méthode `setNombre(Integer)` ligne 2 sera rattrapée par l'instruction `catch` ligne 3, et ne pourra donc pas "s'échapper" : aucune exception `NatException` ne pourra être visible de l'extérieur. Ainsi, il n'est plus nécessaire d'ajouter `throws NatException` à la signature du constructeur `Nat(Integer)`.

En revanche, si nous avions remplacé à la ligne 3 l'instruction `this.nombre = 0` par `this.setNombre(0)`, alors il aurait fallu maintenir la mention `throws NatException`. En effet, même si *nous pouvons savoir* qu'une telle exception ne sera pas levée en pratique, car `0` est un argument autorisé, *le compilateur ne le peut pas*. La seule chose que retient le compilateur, c'est que la méthode `setNombre(Integer)` est susceptible de lever une exception `NatException`, car c'est ce que dit sa mention `throws`.

Exceptions et héritage

De même que `NatException` hérite de `Exception`, nous pouvons définir plusieurs classes d'exceptions organisées selon une hiérarchie d'héritage, et dont certaines possèdent des attributs ou méthodes supplémentaires.

```
 1   class NatException extends Exception {
 2     public NatException()         { super();  }
 3     public NatException(String s) { super(s); }
 4   }
 5   class NullNatException extends NatException {}
 6   class NegNatException extends NatException {
 7     private Integer n;
 8     public  Integer getN()                     { return this.n; }
 9     public          NegNatException(Integer n) { this.n = n;    }
10   }
```

Ici, la classe `NatException` reproduit les deux constructeurs habituels des `Throwable` et `Exception`, la classe `NullNatException` est une sous-classe de `NatException` qui n'apporte pas de champs supplémentaires, et la classe `NegNatException` est une sous-classe de `NatException` qui possède un nouvel attribut `n` et une méthode `getN()` y accédant. On pourra se servir de ce nouvel attribut `n` pour mémoriser la valeur (négative) à l'origine de l'exception.

On peut alors donner une définition plus fine de la méthode `setNombre(Integer)`, qui pourra lever une exception de type `NullNatException` si on l'appelle avec la valeur `null`, ou une exception de type `NegNatException` si on l'appelle avec une valeur négative. Dans ce dernier cas, la valeur négative problématique sera enregistrée dans l'attribut `n` de notre objet exception. À la ligne `11`, on indique ces deux possibilités en séparant les deux classes d'exceptions par une virgule.

```
11   public void setNombre(Integer n) throws NullNatException, NegNatException {
12     if      (n==null) { throw new NullNatException(); }
13     else if (n<0)     { throw new NegNatException(n); }
14     else              { this.nombre = n;              }
15   }
```

On peut adapter le constructeur, par exemple pour affecter le nombre `0` en cas d'argument négatif, mais ne pas rattraper une exception due à une tentative d'affectation de la valeur `null`.

```
16   public Nat(Integer n) throws NullNatException {
17     try { this.setNombre(n); }
18     catch (NegNatException e) { this.nombre = 0; }
19   }
```

Dans ce cas, il suffit à la ligne `16` d'indiquer `throws NullNatException`, car parmi les deux types d'exceptions que peut lever `setNombre`, celle-ci est la seule à ne pas être rattrapée.

Dans la définition de `setNombre(Integer)`, il aurait aussi été possible à la ligne `11` de se contenter de l'indication `throws NatException`, car `NullNatException` et `NegNatException` sont deux sous-classes de `NatException`, et donc toute exception levée par `setNombre` appartient bien, par héritage, à la classe `NatException`. Cependant, cela aurait été moins précis : le compilateur aurait à partir de là considéré qu'un appel à `setNombre(Integer)` pouvait lever n'importe quelle exception de type `NatException`. Dans ce cas, à la ligne `16`, lors de la définition du constructeur `Nat(Integer)`, il n'aurait plus été possible de se contenter de l'indication précise `throws NullNatException` : il aurait fallu préciser à la place `throws NatException` ou rattraper toutes les exceptions de type `NatException`.

Enfin, au moment de rattraper une exception, il est possible de lever à nouveau une exception, qui par exemple contiendra plus d'informations que la première.

```
16'   public Nat(int n) throws NatException {
17'     try { this.setNombre(n); }
18'     catch (NullNatException e) {
19'       throw new NatException("Initialisation avec null.");
20'     }
21'     catch (NegNatException e) {
22'       throw new NatException("Initialisation négative : " + e.getN());
23'     }
24'   }
```

Ici, les exceptions des deux classes `NullNatException` et `NegNatException` sont rattrapées, chacune dans sa clause `catch`. Dans chaque cas, on lève à nouveau une exception `NatException`, qu'on initialise avec un message précisant l'origine de l'exception : dans l'un on rapporte une tentative d'initialisation par la valeur `null`, dans l'autre on rapporte une tentative d'initialisation par avec une valeur négative, et on copie cette valeur dans le message avec `e.getN()`.

Clause finale

Les exceptions modifient la séquence normale des instructions d'un programme. Quand une exception est levée, l'exécution est interrompue et le contrôle est donné au gestionnaire d'exceptions le plus proche dans la pile d'appels, qui éventuellement fera repartir l'exécution à un point de programme différent. Problème potentiel : une partie du code non exécuté après la levée de l'exception pouvait être nécessaire au maintien de la cohérence de l'état de notre programme.

```
 1   public static void lecture(String fichier) {
 2     InputStream flux = new FileInputStream(fichier);
 3     try  { traiteFlux(flux);
 4            flux.close();     }
 5     catch (IOException e) { ... }
 6     ...
 7   }
```

Dans cet exemple, à la ligne 2 nous ouvrons un flux `flux` qui permet de lire le contenu du fichier dont le nom a été passé en argument. Après que le fichier a été traité, le flux est refermé à la ligne 4 avant que l'exécution ne continue.

Supposons que l'appel de méthode `traiteFlux(flux)` lève une exception de type `IOException`. Alors cette exception va être rattrapée, et l'exécution va continuer sans que le flux soit refermé. À ce stade nous avons deux solutions possibles : faire que le code du bloc `catch` contienne lui aussi un appel `flux.close()`, ou déplacer cet appel à la ligne 6.

Mais supposons maintenant que l'appel de méthode `traiteFlux(flux)` lève une exception différente, par exemple de type `IllegalArgumentException`, et que cette exception soit rattrapée par le contexte appelant :

```
 8   public static void main(String[] args) {
 9     try { lecture(args[0]); }
10     catch (IllegalArgumentException e) { ... }
11     ...
12   }
```

Dans ce cas, l'exécution va se poursuivre ligne 11 sans que le flux ait été fermé.

Pour éviter un tel problème, il est possible d'utiliser une clause `finally` à la fin du bloc `try/catch` :

```
 1   public static void lecture(String fichier) {
 2     InputStream flux = new FileInputStream(fichier);
 3'    try  { traiteFlux(flux); }
 4'    catch (IOException e) { ... }
 5'    finally { flux.close(); }
 6     ...
 7   }
```

Le code contenu dans le bloc `finally` est exécuté quoiqu'il arrive. On a en particulier trois issues possibles :

1. Si aucune exception n'est levée, alors le bloc `finally` est exécuté après le bloc `try`, puis on continue ligne 6.
2. Si l'appel `traiteFlux` lève une `IOException`, alors on saute au bloc `catch`, puis le bloc `finally` est exécuté à la suite du bloc `catch`, et enfin on continue ligne 6.
3. Si l'appel `traiteFlux` lève une `IllegalArgumentException`, alors le bloc `finally` est exécuté immédiatement, puis on passe au bloc `catch` de la ligne 10 et à la suite de la méthode `main`.

Il est également possible d'utiliser une paire `try/finally` sans intercaler de bloc `catch`. Découpler ainsi `try/catch` et `try/finally` permet d'ailleurs d'améliorer notre exemple :

```
 1   public static void lecture(String fichier) {
 2     InputStream flux = new FileInputStream(fichier);
 3'    try {
 4'      try     { traiteFlux(flux); }
 5'      finally { flux.close();     }
 6'    }
 7'    catch (IOException e) { ... }
 8'    ...
 9'  }
```

Dans cette version comme dans la précédente, le bloc `finally` est exécuté quelle que soit l'issue de l'appel `traiteFlux`. Cependant, nous pouvons maintenant en plus rattraper une `IOException` qui serait levée par l'appel `flux.close()`.