Cours Programmation Orientée Objet en Java

Cours 5

Anne Parrain

UFR des Sciences - Université d'Artois
Licence Informatique
Semestre 4

Année universitaire 2018–2019

Plan du cours

1. Exceptions

Section 1

Exceptions

Condition exceptionnelle

- Soit le code :

```
public Object depile() {
  Object obj;

  if(taille <= 0)
    ???
  ...
}
```

Que doit-il faire ?

- Une condition exceptionnelle est un problème qui empêche l'exécution normale de la méthode, et pour laquelle la méthode ne possède pas des informations nécessaires, dans son contexte, pour le résoudre.

Exceptions

- Java fournit une solution : le lancement d'une exception.

```
public Object depile() {
  Object obj;

  if(taille <= 0)
    throw new EmptyStackException();
  ...
}
```

- Dans ce code, un objet du type EmptyStackException est crée avec 'new' et ensuite, l'exception est lancée avec 'throw'.

Lancement d'une exception

- Quand une méthode lance une exception, le système essaye de trouver un bloc de code qui peut la traiter.
- Quand un tel bloc est trouvé, le système donne le contrôle d'exécution à ce bloc.
- Si le système ne trouve pas un bloc qui peut traiter l'exception, le programme termine.

Types d'exceptions

- Toute exception est dérivée de la classe Exception, qui est dérivée de la classe Throwable, qui est dérivée de la classe Object. (Elles se trouvent dans le paquetage java.lang.)
- Il y a deux types d'exceptions :
  - « Checked exceptions »
  - « Runtime exceptions »

Types d'exceptions (2)

- « Checked exceptions » : type de situation qu'un programme bien écrit doit anticiper.
- Par exemple, supposons qu'un programme demande un nom de fichier pour l'ouvrir, et que l'utilisateur fournit un nom de fichier erroné.
- « Runtime exception » : bogues dans le programme.
- Par exemple, le programme attribue 'null' à une variable du type Object et ensuite essaye d'accéder à une des ces méthodes.
- Ce type d'exception est dérivé de la classe RuntimeException qui est dérivé de la classe Exception.

Une classe pour les erreurs

- Java possède aussi la classe Error, qui est dérivé de la classe Throwable. Il s'agit des conditions spéciales et externes au programme.
- Par exemple, le fichier ne peut pas être ouvert due à un problème du matériel.
- En Java, il est possible d'attraper et de traiter des erreurs aussi.
- Mais ici nous allons supposer que dans ces cas, il fait plus de sens d'arrêter l'exécution du programme.

Traitement d'une exception

- Pour traiter une exception nous devons d'abord mettre le morceau du code qui peut lancer l'exception dans un bloc 'try'.
- Son traitement sera fait dans un des blocs 'catch' (ou le bloc 'finally') qui se suivent.

```
public Object depile() {
  Object obj;
  try {
    if(taille <= 0)
      throw new EmptyStackException();
  } catch(EmptyStackException e) {
    System.err.println("Probleme!!");
  }
  ...
}
```

Traitement d'une exception (2)

- Si plusieurs types d'exceptions peuvent être lancées, nous devons utiliser plus d'un 'catch' et/ou 'finally'.

```
try {

  \\ code avec deux types d'exceptions

} catch(Type1 id1) {
  ...
} catch(Type2 id2) {
  ...
} finally {
  ...
}
```

Traitement d'une exception (3)

- Chaque clause 'catch' se comporte comme une méthode qui reçoit toujours un paramètre. Le paramètre peut être utilisé dans le bloc 'catch'.
- Les blocs 'catch' doivent apparaître tout de suite après le bloc 'try'.
- Quand une exception est lancée, le mécanisme cherche le premier 'catch' dont le paramètre correspond à l'exception.
- Seulement une clause 'catch' est exécutée (il ne se comporte pas comme un 'switch').

Terminaison vs. reprise

- En principe, Java suppose que le programme termine après le traitement de l'exception.
- Mais il est possible d'implémenter une reprise après le traitement de l'exception.
- La solution est de placer les blocs 'try-catch' à l'intérieur d'un 'while'
- Note qu'en général il est difficile de reprendre l'exécution après une exception.

Nos propres exceptions

- Java possède une hiérarchie d'exceptions à notre disposition (regardez la API).
- Mais nous pouvons aussi créer les nôtres.
- Par exemple, supposons que nos implémentons une classe liste chaînée. Si nous planifions distribuer cela dans un paquetage, il est désirable que tout le code soit délivré ensemble, et que les éventuelles exceptions soient significatives. Donc, le paquetage doit contenir ses propres exceptions.
- Pour cela, il suffit de créer des classes qui dérivent de la classe Exception et les utiliser.

Nos propres exceptions (2)

```
class SimpleException extends Exception {}

public class DemoException {
  public void f() throws SimpleException {
    System.out.println("Appel de f()");
    throw new SimpleException();
}
  public static void main(String[] args) {
    DemoException d = new DemoException();
    try {
      d.f();
    } catch(SimpleException e) {
      System.out.println("Exception attrapee!");
    }
  }
}
```

Nos propres exceptions (2)

- Il est parfois utile de créer un constructeur avec un message d'erreur.

```
class SimpleException extends Exception {
  public SimpleException() {}
  public SimpleException(String msg) { super(msg); }
}
public class DemoException {
  public void f() throws SimpleException {
    throw new SimpleException("Exception dans f()");
}
  public static void main(String[] args) {
    DemoException d = new DemoException();
    try { d.f(); } catch(SimpleException e) {
      System.err.println(e.getMessage());
    }
  }
}
```

Relancer une exception sans traitement

- Dans certaines cas, nous ne pouvons pas traiter l'exception dans la méthode. Dans ce cas, nous devons relancer l'exception sans la traiter.
- Pour relancer une exception sans la traiter nous utilisons le mot-clé **throws**.

```
public void ecritListe() throws IOException {
  // code sans les blocs try-catch-finally
}
```

- Cela veut dire que le traitement de l'exception sera fait dans une méthode placé avant 'ecritListe' dans la pile d'appels.
- Le comportement de 'throw' s'avère "similaire" à celui de 'return'.

Relancement d'une exception (2)

- Attention : le traitement d'une checked exception est obligatoire en Java.
- Cela veut dire que si nous utilisons une méthode qui lance une checked exception donc, soit nous devons utiliser des blocs 'try-catch-finally', ou bien nous devons la relancer avec 'throws'.
- En revanche, il est possible de déclarer qu'une méthode lance une exception sans qu'il le fasse vraiment.
- Cela peut être intéressant pour les méthodes abstraites et les interfaces.

Un autre exemple

```
import java.io.*;
import java.util.Vector;

public class ListeNumeros {
  private Vector vecteur;
  private static final int TAILLE = 10;
  ...
  public void ecritListe() {
    PrintWriter sortie = new PrintWriter(
      new FileWriter("Sortie.txt"));
    for(int i = 0; i < TAILLE; i++) {
      sortie.println("Valeur a : + i + " = " +
        vecteur.elementAt(i));
    }
  }
}
```

Un autre exemple (2)

- Si le fichier ‘Sortie.txt’ ne peut pas être ouvert, le constructeur de FileWriter relance l’exception IOException.
- Par ailleurs, si la valeur de ‘i’ donnée comme paramètre à la méthode ‘elementAt’ est trop petite ou trop grande, cette méthode relancera l’exception ArrayIndexOutOfBoundsException.
- La première exception est une checked exception, alors que la deuxième ne l’est pas.
- Ce programme ne traite pas les exceptions mentionnées. Si elles sont lancées, le programme terminera.

```
PrintWriter sortie;
try {
  System.out.println("J'essaye d'ouvrir le fichier.");
  sortie = new PrintWriter(
    new FileWriter("Sortie.txt"));
  for(int i = 0; i < TAILLE; i++) {
    sortie.println(i + ": " + vecteur.elementAt(i));
  }
} catch (IOException e) {
  System.err.println("IOException");
} finally {
  if(sortie != null) {
    System.out.println("Je ferme PrintWriter.");
    sortie.close();
  }
  else
    System.out.println("PrintWriter pas ouvert.");
}
```

L'exécution

- Si aucune exception n'est produite :

```
1 J'essaye d'ouvrir le fichier.
2 Je ferme PrintWriter.
```

- Si le système ne peut pas écrire dans le fichier 'Sortie.txt' :

```
1 J'essaye d'ouvrir le fichier.
2 IOException
3 Je ferme PrintWriter.
```

La classe Exception

- Il est possible de créer un bloc ‘catch’ qui attrape toute exception :

```
catch(Exception e) {
  System.err.println("Exception attrapee.");
}
```

- Cela doit être mis le dernier bloc ‘catch’.
- La classe Exception possède plusieurs méthodes utiles :

```
String getMessage()
String toString()
void printStackTrace()
Thorwable fillInStackTrace()
```

La pile d'appels

```
class DemoException3 {
  public static void f() throws Exception {
    throw new Exception("Mon exception");
  }
  public static void main(String[] args) {
    try {
      f();
    } catch(Exception e) {
      e.printStackTrace();
    }
  }
}
```

```
1 java.lang.Exception: Mon exception
2   at DemoException3.f(DemoException3.java:3)
3   at DemoException3.main(DemoException3.java:7)
```

La pile d'appels (2)

```
public class DemoException4 {
  public static void f() throws Exception {
    throw new Exception("Mon exception");
  }
  public static void g() throws Exception {
    try { f(); } catch(Exception e) {
      e.printStackTrace();
      throw e;
    }
  }
  public static void main(String[] args) {
    try { g(); } catch(Exception e) {
      e.printStackTrace();
    }
  }
}
```

La pile d'appels (3)

```
1 java.lang.Exception: Mon exception
2   at DemoException4.f(DemoException4.java:4)
3   at DemoException4.g(DemoException4.java:8)
4   at DemoException4.main(DemoException4.java:17)
5 java.lang.Exception: Mon exception
6   at DemoException4.f(DemoException4.java:4)
7   at DemoException4.g(DemoException4.java:8)
8   at DemoException4.main(DemoException4.java:17)
```

La pile d'appels (4)

```
public class DemoException5 {
  public static void f() throws Exception {
    throw new Exception("Mon exception");
  }
  public static void h() throws Exception {
    try { f(); } catch(Exception e) {
      e.printStackTrace();
      throw (Exception)e.fillInStackTrace();
    }
  }
  public static void main(String[] args) {
    try { h(); } catch(Exception e) {
      e.printStackTrace();
    }
  }
}
```

La pile d'appels (5)

```
1 java.lang.Exception: Mon exception
2   at DemoException5.f(DemoException5.java:3)
3   at DemoException5.h(DemoException5.java:7)
4   at DemoException5.main(DemoException5.java:15)
5 java.lang.Exception: Mon exception
6   at DemoException5.h(DemoException5.java:10)
7   at DemoException5.main(DemoException5.java:15)
```

Exceptions enchaînées

- Il est parfois utile de traiter une exception et de lancer une autre exception.
- Par ailleurs, il est parfois utile de savoir quelle exception a causé le lancement de l'exception courante.
- Pour cela nous pouvons utiliser des méthodes de la classe Trowable :

```
Throwable getCause()
Throwable initCause()

// Constructeurs :
Throwable(String, Throwable)
Throwable(Throwable)
```

Exceptions enchaînées (2)

```
public class DemoException6 {
  public static void f() throws Exception {
    throw new Exception("1e exception");
  }
  public static void h() throws Exception {
    try { f(); } catch(Exception e) {
      Exception e2 = new Exception("2e exception");
      e2.initCause(e);
      throw e2;
    }
  }
  public static void main(String[] args) {
    try { h(); } catch(Exception e) {
      e.printStackTrace();
    }
  }
}
```

Exceptions enchaînées (3)

```
1 java.lang.Exception: 2e exception
2   at DemoException6.h(DemoException6.java:9)
3   at DemoException6.main(DemoException6.java:16)
4 Caused by: java.lang.Exception: 1e exception
5   at DemoException6.f(DemoException6.java:3)
6   at DemoException6.h(DemoException6.java:7)
7   ... 1 more
```

L'utilisation de 'finally'

- Il est parfois nécessaire d'exécuter un morceau de code après le traitement de l'exception. Cela est l'objectif du bloc 'finally'.
- Le bloc 'finally' sera **toujours** exécuté !

L'utilisation de 'finally' (2)

```
public class DemoException7 {
  public static void main(String[] args) {
    int i = 0;
    while(true) {
      try {
        if(i++ == 0)
          throw new Exception();
        System.out.println("Pas d'exception.");
      } catch(Exception e) {
        System.out.println("Exception.");
      } finally {
        System.out.println("Finally.");
        if(i == 2) break;
      }
    }
  }
}
```

L'utilisation de 'finally' (3)

```
1 Exception.
2 Finally.
3 Pas d'exception.
4 Finally.
```

‘finally’ et ‘return’

```
public class DemoException8 {
  public static void f(int i) {
    System.out.println("Debut.");
    try {
      System.out.println("Point 1.");
      if(i == 1) return;
      System.out.println("Point 2.");
      if(i == 2) return;
      System.out.println("Fin.");
      return;
    } finally { System.out.println("Finally."); }
  }
  public static void main(String[] args) {
    for(int i = 1; i <= 3; i++)
      f(i);
  }
}
```

‘finally’ et ‘return’

```
Debut.
Point 1.
Finally.
Debut.
Point 1.
Point 2.
Finally.
Debut.
Point 1.
Point 2.
Fin.
Finally.
```

L'exception perdue

```
public class MessagePerdu {
  public static void main(String[] args) {
    try {
      try {
        throw new Exception("Message perdu");
      } finally {
        throw new Exception("Message trouve");
      }
    } catch(Exception e) {
      System.out.println(e.getMessage());
    }
  }
}
```

```
1 Message trouve
```

L'exception perdu (2)

```
public class MessagePerdu2 {
  public static void main(String[] args) {
    try {
      throw new RuntimeException();
    } finally {
      return;
    }
  }
}
```

Exercice

Construisez un programme qui lit deux numéros de l'entrée standard et imprime le résultat de la division du premier par le second, en tenant compte de la division par zéro avec une exception. Créez votre propre exception avec un message d'erreur. Si l'exception se produit, le programme doit imprimer le message d'erreur et demander un nouveau numéro à l'utilisateur.