Remise à niveau Programmation

Programmation Objet avec Java

Gestion des exceptions

#4

Licence Pro
Métiers de l'Informatique : Conception, Développement,
Test de Logiciels, Parcours "Génie Logiciel, Système d'Information"
2020-2021

Erreurs et Exceptions

Robustesse d'un programme

Un programme est "robuste" s'il est capable de gérer des erreurs d'exécution sans s'interrompre

Exemple

Le programme suivant n'est pas robuste.

```
1  {
2    Scanner sc = new Scanner( System.in) ;
3    int numérateur = sc.nextInt() ;
4    int dénominateur = sc.nextInt() ;
5
6    int r = numérateur / dénominateur ;
7    System.out.println(numérateur +" / "+ dénominateur + " = "+r) ;
8
9    System.out.println("Fin de programme.") ;
10 }
```

Si l'utilisateur saisi l'entier **0** pour **dénominateur**, alors une division par **0** survient et provoque une erreur. Le programme interrompt son exécution et la dernière instruction (l. 9) n'est jamais évaluée alors qu'elle ne pose aucun problème. Cela ne doit pas arriver…

Principes de gestion d'erreurs

Une bonne gestion des erreurs doit permettre :

- d'un point de vue dynamique :
  - détecter, récupérer et traiter des erreurs
  - (si possible) réparer la situation erronée
- du point de vue de la lisibilité :
  - séparer les instructions gérant le comportement normal du programme des instructions gérant les erreurs

Les Exceptions

Java propose un mécanisme de gestion d'erreur, appelé *exception,* satisfaisant tous les principes précédents.
Une *exception* peut être vue comme un déroutement de la séquence normale d'instructions d'un programme.
Java utilise un principe explicite de surveillance des instructions et crée une exception quand une erreur est détectée lors de leur exécution.

La surveillance avec try/catch

Le système de surveillance explicite proposé par Java s'articule autour d'une structure de contrôle particulière composée de deux blocs :

- **`try{...}`** définit un bloc d'instructions à surveiller
- **`catch ... {...}`** définit un bloc d'instructions à exécuter quand une erreur est détectée

Exemple

La « mise sous surveillance » du code précédant donnerait le code suivant :

```
1  {
2    // Bloc sous surveillance
3    try
4    {
5         Scanner sc = new Scanner( System.in) ;
6         int numérateur = sc.nextInt() ;
7         int dénominateur = sc.nextInt() ;
8
9         int r = numérateur / dénominateur ;
10        System.out.println(numérateur +"/"+ dénominateur + " = "+ r) ;
11   }
12   // instruction a exécuter si une ArithmeticException survient
13   catch ( ArithmeticException e )
14   {
15        System.out.println("Une division par 0 est survenue.") ;
16   }
17
18   // reprise du déroulement normal du programme
19   System.out.println("Fin de programme.") ;
20 }
```

Principes de fonctionnement

Déroulement nominal (*ie* lorsqu'il n'y a pas d'erreur) :

- Le programme est exécuté à partir du début et jusqu'à la fin
  - Les instructions du bloc **`try{...}`** SONT exécutées séquentiellement
  - Les instructions du bloc **`catch ... {...}`** NE SONT PAS exécutées.

Levée d'exception :

Si lors de l'exécution des instructions du bloc une exception est levée (par exemple une **`ArithmeticException`** parce que l'on a essayé de faire une division par **`0`**), alors :

- Le programme est stoppé à l'instruction qui a émise l'exception (ici ligne 9)
- Les instructions du bloc **`try{...}`** qui suivent celle qui a émit l'exception ne seront pas évaluées et, à la place,
- Les instructions du bloc **`catch ... {...}`** le plus proche attrapant l'exception sera exécuté à la place (ici la ligne 13).
- Ensuite le programme reprend le cours de son exécution (ici ligne 19).

Principes de Lisibilité

Il n'est pas utile de surveiller des instructions ne provoquant pas d'erreurs (ici lignes 5 à 7), cependant l'usage est de laisser dans le bloc **try** toutes les instructions qui correspondent au comportement normal.

Exception implicite/explicite

Les exceptions implicites :

Elles sont automatiquement créées par la machine virtuelle lorsqu'elle détecte une erreur. Elles correspondent à des erreurs générales (division par zéro, invocation d'une méthode avec une référence **`null`**, ...)

Les exceptions explicites :

Elles sont explicitement créées par le programmeur lorsqu'il estime qu'une situation contrevient au déroulement normal du programme

Exception Implicite générée par le système

```
{
   try
   {
      Scanner sc = new Scanner(System.in);
      int numé = sc.readInt() ;
      int dénom = sc.readInt() ;

      double r = (double)(numé)/dénom ;
      System.out.println(r) ;



   }
   catch ( ArithmeticException e )
   {
      System.out.println("Div par 0 !");
   }

   System.out.println("Fin de prog.") ;
}
```

Exception Explicite levée par le programme

```
{

  try
  {
     Scanner sc = new Scanner(System.in);
     Personne p ;
     int age =sc.nextInt() ;

     if ( age < 0 )
        throw new Exception("Pas d'age<0!");

        p = new Personne(age) ;
  }
  catch (Exception e)
  {
     System.out.println(e);
  }

  System.out.println("Fin de prog.") ;
}
```

Lors de la conception d'un programme il faut anticiper les dysfonctionnements possibles. Dans le code de droite, le programmeur a anticipé la possibilité qu'un utilisateur saisisse une valeur négative pour l'âge d'une **`Personne`**.

Il a donc mis en place une procédure pour palier à cette erreur. Dans Java, pour faire cela, il faut utiliser les **`Exception`**s.

Instance d'Exception

Les exceptions sont des instances de la classe **`Exception`**. A ce titre elles sont créées au moyen de l'opérateur **`new`**

```
new Exception("Pas d'age<0!");
```

Le constructeur de l'exception prend en général au moins un paramètre qui décrit le motif de la levée d'exception.

Remarque

Nous verrons plus tard qu'il est possible de définir de nouvelles classes d'exception (par spécialisation de la classe Exception), et qu'il sera possible de transmettre plus qu'un message. Par exemple la classe d'exception pourra avoir des variables d'instances spécifiques permettant de stocker un état complexe (plusieurs valeurs) liées à la levée de l'exception.

Les exceptions sont émises par la commande **`throw`** et elles sont propagées jusqu'à ce qu'elles soient attrapées (**`catch`** en anglais) par une structure de type **`try-catch`**.

```
throw new Exception("Pas d'age<0!");
```

Lorsque la clause **`catch`** est invoquée, la référence de l'exception capturée est stockée dans une variable référence (ici la variable référence est nommée **`e`**) :

```
...
catch (Exception e)
{
     ...
}
```

Grâce à cette variable référence **`e`**, il est possible de manipuler le contenu de l'exception capturée dans le corps d'instruction du **`catch`**. Par exemple pour afficher le message.

```
...
catch (Exception e)
{
     System.out.println( e) ;
}
```

Cela deviendra intéressant lorsque nous définirons nos propres classes d'exception et que celles-ci transporteront plus qu'un message (les valeurs qui ont provoqué la création de l'exception).

Pluralité des Exceptions

Une même portion de code peut potentiellement lever plusieurs exceptions différentes (mais au cours d'une exécution du programme elle n'en lève qu'une seule à la fois).

Dans ce cas il est possible de mettre en place plusieurs clauses **`catch (Exception e){...}`** afin de différencier les comportements en fonction des **`Exception`**s trouvées :

```
{
 try {
   Scanner sc = new Scanner(System.in);
   Personne p ;
   int age =sc.nextInt()

   if ( age < 0 )
   {
     throw new Exception("Pas d'age<0!");
   }
   p = new Personne(age) ;
   System.out.println( 1 / p.getAge() ) ;
 }
 catch (ArithmeticException e)                    // <-------
 {
   System.out.println(e);
 }
 catch (Exception e)                              // <-------
 {
   System.out.println(e);
 }

 System.out.println("Fin de prog.") ;
}
```

Remarque

Ici le bloc soumit au contrôle peut lever :

- **`ArithmeticException`** par la division de l'instruction :

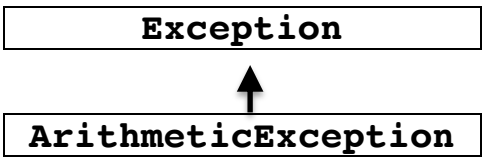
```
System.out.println( 1 / p.getAge() )
```

- **`Exception`** par l'évaluation de l'instruction dans le **`if`**

```
   if ( age < 0 )
   {
     throw new Exception("Pas d'age<0!");
   }
```

Ordre sur les clauses `catch`

De la même façon qu'une seule exception est émise au final, un seul **`catch`** est invoqué.



Comme les instances d'**`Exception`**s émises dérivent toutes de la classe parente **`Exception`**, il faut faire attention dans notre façon d'ordonner les clauses **`catch`**
Il faut toujours ordonner les **`catch`**s pour attraper les **`Exception`**s en commençant par les plus spécifiques pour finir par les plus générales.

Exemple

Soit le code suivant :

```
            {
              try {
                Scanner sc = new Scanner(System.in);
                Personne p ;
                int age =sc.nextInt()

                if ( age < 0 )
                {
                   throw new Exception("Pas d'age<0!");
                }
                p = new Personne(age) ;
                System.out.println( 1 / p.getAge() ) ;
              }
// Solution 1                          // Solution 2
catch (Exception e)                    catch (ArithmeticException e)
{                                      {
  System.out.println("age<0");           System.out.println("Div/0");
}                                      }
catch (ArithmeticException e)          catch (Exception e)
{                                      {
  System.out.println("Div/0");           System.out.println("age<0");
}                                      }

              System.out.println("Fin de prog.") ;
            }
```

Dans le cas où une division par 0 surviendrait, une **`ArithmeticException`** serait levée. À partir de là, les 2 solutions proposées (qui divergent sur l'ordre des clauses) ne se comportent pas de la même façon :

Attention

Solution 1
L'instance d'**`ArithmeticException`** est aussi une instance d'**`Exception`**. Elle est captée par le premier **`catch`** et le programme affiche **`age<0`**

Solution 2 :
L'instance d'**`ArithmeticException`** est captée par le premier **`catch`** et le programme affiche **`Div/0`**

Et … `finally`

Il est possible de spécifier un bloc d'instruction qui doit être évalué dans tous les cas : qu'une **`Exception`** soit levée ou pas !
Ceci est réalisé au moyen de la clause **`finally`**.

Exemple

Soit le code suivant :

```
public static int inverse( int x)
{
     int inv ;
     try
     {
          inv = 1 / x ;
          return inv ;
     }
     catch (ArithmeticException e)
     {
          System.out.println(e);
          return 0 ;
     }
     finally
     {
          System.out.println(" On continue ");
     }
}
```

Avec ce code, DANS TOUS LES CAS, que l'exception soit levée ou non, l'appel à cette méthode provoquera l'affichage `"On continue."`.

MÊME L'INSTRUCTION **`return`** est évaluée après le bloc **`finally`**!

Délégation de la capture

Une méthode (ou un constructeur) peut lever une exception et déléguer son traitement à la méthode appelante.
Toute méthode qui délègue la capture d'une ou plusieurs exceptions doit le signaler dans sa signature avec le mot clé **throws** .

```
public double inverse(double x) throws ArithmeticException
{
     ...
}
```

Le mot clef **throws** indique au compilateur que la méthode est susceptible de lever une exception. Il indique aussi que toute autre méthode appelant cette première méthode (ou ce constructeur) doit les capturer ou à nouveau déléguer à son tour leur capture …

Exemple

Soit une classe **Animal** dotée d'un constructeur pouvant lever une exception :

```
public class Animal
{
     public Animal(double p) throws Exception
     {
          if (p < 0)
               throw new Exception("Poids négatif") ;
          this.poids = p ;
     }
}
```

Toute méthode ***f1*** appelant ce constructeur DOIT mettre en place un mécanisme de contrôle de l'**Exception** ou elle-même déléguer le contrôle de cette exception à la méthode ***f2*** appelant elle même la méthode ***f1***. Dans le code suivant le rôle de la méthode ***f1*** est jouée par la méthode **main** (et donc aucune méthode ne peut jouer le rôle de la méthode ***f2***).

```
public class TestAnimal
{
     public static void main(String [] args)
     {
          try
          {
               Animal a = new Animal( -15) ;
          }
          catch (Exception e)
          {
               System.out.println( e) ;
          }
     }
}
```

Délégation de la capture

Dans le code précédent, le constructeur **`Animal(double)`** est susceptible de lever une exception.
Le constructeur ne gère pas la capture de l'**`Exception`**, il délègue sa capture au programme appelant.

Ici un programme appelant est la méthode **`copie(Animal)`**. Cette dernière ne gère pas non plus la capture de l'**`Exception`**. Elle doit donc également déléguer sa capture à son propre programme appelant, ici la méthode **`main()`** de la classe de test **`TestAnimal`**.

```
public class Animal
{
     private double poids ;

     public double getPoids() { return this.poids ; }
     public Animal() {this.poids = 0. ;}

     // Lève l'Exception et délègue sa capture              <-------
     public Animal(double p) throws Exception               <-------
     {
          if (p < 0)
               throw new Exception("Poids négatif") ;
          this.poids = p ;
     }

     // Ne lève pas en propre une Exception mais            <-------
     // délègue à son tour la capture de l'Exception       <-------
     // levée par le constructeur                          <-------
     public Animal copie( Animal a) throws Exception
     {
          return new Animal( a.getPoids()) ;
     }
}
```

```
public class TestAnimal
{
     public static void main(String [] args)
     {
          // gestion de la capture de l'Exception          <-------
          try
          {
               Animal a1  = new Animal();
               Animal a2 = a1.copie(a1) ;
          }
          catch (Exception e)
          {
               System.out.println( e) ;
          }
     }
}
```

Exception non contrôlées

Les exceptions non contrôlées (**`Error`** et **`RunTimeException`** et leurs sous-classes) sont le plus souvent des exceptions créées automatiquement par la machine virtuelle.

- Si elles ne sont pas capturées par un **catch** dans la méthode courante leur capture est automatiquement déléguée à la méthode appelante.
- Si la méthode appelante (*X*) ne la capture pas leur capture est de nouveau automatiquement déléguée à la méthode appelante de *X*.

Le processus se poursuit jusqu'à ce qu'une méthode capture l'exception. Si aucune ne le fait le ***programme est interrompu***, et la machine virtuelle affiche l'exception.

Exemple

`ArithmeticException` est une spécialisation de **`RunTimeException`**. C'est une **`Exception`** non-contrôlée.

Exemple

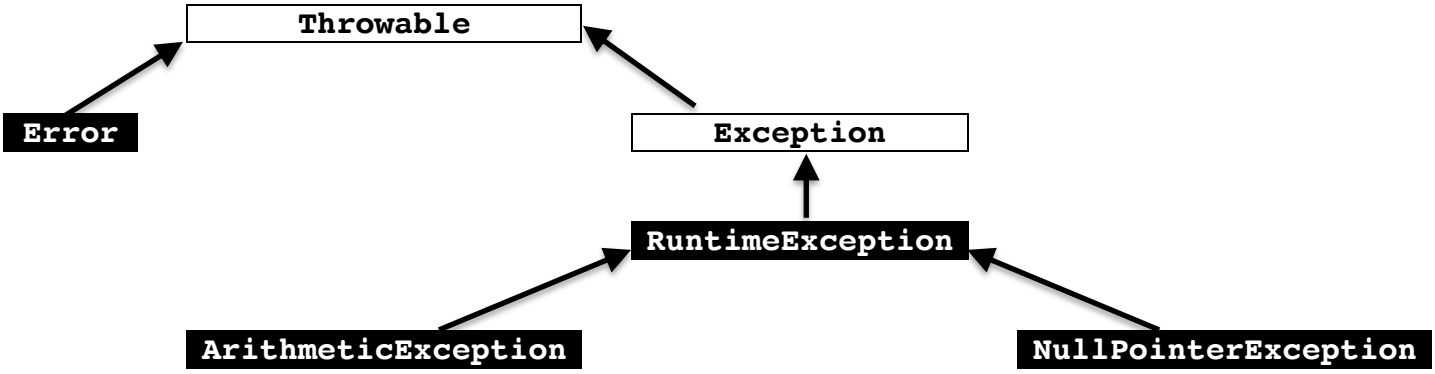
Dans le programme suivante :
- si le dénominateur vaut 0 une exception non contrôlée (**`ArithmeticException`**) est automatiquement créée (levée). N'étant pas capturée dans la méthode elle est automatiquement déléguée à la méthode appelante (**`main()`**)

```
public class Fraction
{
     private in numérateur ;
     private int dénominateur ;
     ...
     public int valeurNumérique(){
          return (this.getNumérateur()/this.getDénominateur()) ;
     }
}
```

La méthode appelante (**`main()`**) capture l'exception. Si elle ne le faisait pas, le programme serait interrompu et l'exception affichée (le **`main()`**) n'ayant pas d'appelant on ne peut continuer la délégation)

```
public class TestFraction
{
     public static void main(String [] args)
     {
          Fraction f = new Fraction( 12, 0) ;
          try
          {
               double r = f.valeurNémrique() ;
          }
          catch (ArithmeticException e)
          {
               System.out.prinln( e) ;
          }
     }
}
```

Étendre la classe Exception



Exceptions contrôlées : elles sont explicitement créées (**new**) et levées (**throw**) dans les programmes.
Le compilateur exigera qu'elles soient surveillées et capturées par un **try-catch**

Exceptions non contrôlées : elles sont automatiquement créées et levées par la machine virtuelle.
Le compilateur n'exigera pas qu'elles soient surveillées et capturées par un **try-catch** (mais on peut le faire)

Définition d'une classe d'Exception adaptée

La classe **Exception** est généralement adaptée (spécialisée) pour permettre le transport d'information sur l'incident qui a provoqué la levée d'**Exception**.
On leur adjoint en général des variables supplémentaires (en plus de la variable transportant un message hérité de la classe mère).
Par exemple, ici, la levée d'exception est produite par une incohérence sur le poids fournit au constructeur (poids négatif).
La classe **AnimalException**, introduit donc une variable **poidsFournis**, qui permettra de transporter le poids incohérent avec l'Exception.

```
public class AnimalException extends Exception
{
     private double poidsFournis ;

     public AnimalException( String message, double poids)
     {
          super( message) ;
          this.poidsFournis = poids ;
     }

     public getPoidsFournis() { return this.poidsFournis ;)
     public String toString(){
          return super.toString()+this.poidsFournis;
     }
}
```

Étendre la classe Exception

La nouvelle classe **`AnimalException`** se manipule comme sa classe mère : la classe **`Exception`** :

- on lève l'**`AnimalException`** avec **`throw`**
- on délègue la capture avec **`throws`**,

```
public class Animal
{
     private double poids ;

     public Animal(double poids) throws AnimalException
     {
          if( poids < 0.)
               throw new AnimalException("poids négatif", poids);
          this.poids = poids ;
     }
}
```

- on capture l'**`AnimalException`** avec **`try-catch`**

```
{
     try
     {
          double p = s.nextDouble() ;
          Animal a = new Animal(s) ;
     }
     catch (AnimalException e)
     {
          double lePoids = e.getPoidsFournis() ;
     }
}
```

La seule différence tient à l'adaptation du constructeur et à l'usage que l'on peut faire de l'instance d'**`AnimalException`** (variables d'instances et services proposés par les méthodes de la classe **`AnimalException`**.

Bon usage des Exception

- Une exception doit être utilisée dans les situations exceptionnelles. Elle ne doit pas se substituer aux contrôles usuels (filtres, etc...)
- En général une méthode (autre que **`main()`**) qui crée une exception la délègue vers la méthode appelante
  (ainsi le traitement de l'exception est de la responsabilité de l'appelant qui peut choisir la façon de sortir de la situation erronée)

Exercices #4

Ces exercices s'appuient sur les classes **`Vaisseau`** et **`VolEnFormation`**. Vous pouvez télécharger ces classes disponibles dans l'archive seance4_debut.tgz à l'adresse suivante :

https://www.lipn.univ-paris13.fr/~santini/RN3/

Question 0 :
Afin de vous aider dans votre développement, il vous est conseillé de compiler la JavaDoc. Ainsi, vous noterez entre autres choses l'apparition d'une méthode **`initRandom()`** dans la classe **`Vaisseau`**.

Question 1 :
Dans une classe de test, créez un vol en formation de 10 Vaisseaux aléatoires, puis affichez la formation.

Question 2 :
Toujours dans la même classe, créez un autre vaisseau aléatoire et insérez le dans la formation à une position déterminée par un indice saisi au clavier par un utilisateur. Que se passe-t-il si vous saisissez 10 au clavier. Expliquez pourquoi ?

Plutôt que de faire appel à des messages d'erreur, la bonne façon de gérer ce type de situation en Java est d'utiliser le système de levée d''exception.

Question 3 :
Définir une classe **`FormationInsertionException`** héritant de la classe **`Exception`** prédéfinie en Java (voir la documentation de l'API Java). En plus de la variable d'instance de type `String` héritée d'**`Exception`**, ajouter deux attributs **`rangSouhait`** et **`rangMax`** de type **`int`**:

- créer un constructeur membre à membre en conséquence
- redéfinir la méthode **`toString()`** pour qu'elle retourne la chaîne de caractères prévue par **`Exception`**, à laquelle on ajoute par exemple **`" : 10>5"`**. Dans cet exemple, on a souhaité ajouter un **`Vaisseau`** directement au **`rangSouhait`** 10 alors qu'il n'y a que 5 **`Vaisseau`** pour l'instant et qu'en conséquence, on ne peut ajouter directement de **`Vaisseau`** au delà du **`rangMax`** 4.

Question 4 :
Modifiez la méthode **`setVaisseau(...)`** de la classe **`VolEnFormation`** de manière à ce qu'elle lève une exception **`FormationInsertionException`** en cas d'erreur, au lieu de simplement procéder à un **`println().`**

Question 5 :

Ajoutez **`throws Exception`** à la signature de la méthode **`main()`** du programme de Test. De cette manière, on peut se passer de gérer n'importe quelle exception dans la méthode **`main()`**, en déléguant leur gestion à la machine virtuelle java elle-même. Lancez le programme de test pour constater la manière dont la machine virtuelle gère une exception explicite jamais traitée. *ATTENTION*, procéder ainsi, c'est *MAL* : cette question est uniquement posée pour montrer le fonctionnement des exceptions.

Question 6 :
Reproduisez exactement le même affichage en supprimant le **`throws Exception`** de la classe de test mais en ajoutant un **`try-catch`** autour de l'insertion et de l'affichage immédiatement après. Pour que la pile d'exécution soit affichée, utilisez dans le bloc **`catch`** la méthode **`printStackTrace()`** définie pour toutes les exceptions.

En plus de permettre de constater des erreurs, les exceptions offrent un mécanisme pour pouvoir les récupérer et les corriger sans empêcher le programme de continuer à fonctionner.

Question 7 :
Modifiez votre programme pour que si le rang d'insertion saisi est trop grand ce dernier soit alors redemandé à l'utilisateur jusqu'à ce que l'indice soit valide (inférieur à la taille du tableau). Ainsi, l'erreur est récupérée et le **`Vaisseau`** finir toujours par être ajouté à une position valide. Vous pouvez d'ailleurs sortir l'affichage du **`VolEnFormation`** du **`try-catch`** ou bien le mettre dans un bloc **`finally`**.