LU2IN002 - Introduction à la programmation orientée-objet

Responsable de l'UE : Christophe Marsala
(email: Christophe.Marsala@lip6.fr)

Cours du lundi : Sabrina Tollari
(email: Sabrina.Tollari@lip6.fr)

(support réalisé à partir de ceux de Christophe Marsala et de Vincent Guigue)

SCIENCES SORBONNE UNIVERSITÉ

Cours 9 – lundi 21 novembre 2022

Programme du jour

1. Exceptions
   - Comment gérer les erreurs à l'exécution ?
   - Qu'est-ce qu'une exception ?
   - `throw` : levée d'une exception
   - Définir ses exceptions personnalisées
   - `try ... catch` : capture d'exceptions
   - `finally`
   - `throws`

Rappels : erreurs usuelles à l'exécution

★ Certaines erreurs à l'exécution (JVM) sont dûes à des erreurs de programmation

Solution : le programmeur lit les messages d'erreurs et corrige

- NullPointerException

```
1 Point p = null;
2 p.move(1, 0);
3 // Exception in thread "main" java.lang.NullPointerException
4 //     at cours1.TestPoint.main(TestPoint.java:2)
```

  - ♦ erreur qui arrive souvent dans des cas plus complexes de composition d'objet

- IndexOutOfBoundException

```
1 int[] tab = new int[3];
2 tab[5] = 10;
3 // Exception in thread "main"
4 //        java.lang.ArrayIndexOutOfBoundsException: 5
5 //       at cours1.TestPoint.main(TestPoint.java:2)
```

  - ♦ vérifier la ligne et l'index !
  - ♦ arrive souvent dans les boucles for

Réaction en cas d'échec du test

★ Certaines erreurs à l'exécution (JVM) ne sont pas dûes à des erreurs de programmation, mais aux conditions d'exécution
Exemple : l'utilisateur saisie une mauvaise valeur
Exemple : l'utilisateur demande de dépiler une pile vide

Que faire ?

- utiliser une valeur spéciale : `null`, `NaN`

```
1 double valUtilisateur=2.0;
2 double r = Math.asin(valUtilisateur);
3 // NB: asin n'accepte que des valeurs entre -1 et 1
4 System.out.println(r); // NaN
```

- effectuer une rupture de calcul

```
1  ArrayList<Double> arr = new ArrayList<Double>();
2  arr.add(1.5);
3  arr.add(2.5);
4  int valUtilisateur=5;
5  Double x=arr.get(valUtilisateur);
6  // Exception in thread "main"
7  //   java.lang.IndexOutOfBoundsException: Index: 5, Size: 2
8  //   at java.util.ArrayList.RangeCheck(ArrayList.java:547)
9  //   at java.util.ArrayList.get(ArrayList.java:322)
10 //   at Test.main(Test.java:3)
```

Exceptions

Du point de vue programmeur :

- **Une exception est une rupture de calcul**

- ... qui arrête le programme quand il y a des erreurs
  - division par zéro
  - accés à la référence `null`
  - ouverture d'un fichier inexistant
  - ...
- sauf si l'exception est gérée

En Java :

- une exception est un objet
- des mécanismes permettent de **gérer l'exception**
  - `throw` : levée de l'exception
  - `throws` : information de propagation de l'exception
  - `try ... catch ... finally` : capture de l'exception

Exceptions : création et déclenchement

Une exception est un objet d'une classe...

- ... qui hérite de la classe `Exception`,
- ... qui elle-même hérite de la classe `Throwable`

Hiérarchie de classes pour les exceptions :

```
java.lang.Object
  extended by java.lang.Throwable
      extended by java.lang.Exception
          extended by java.lang.RuntimeException
```

- Création d'une exception : création d'un objet

```
RuntimeException e = new RuntimeException("Division par zéro");
```

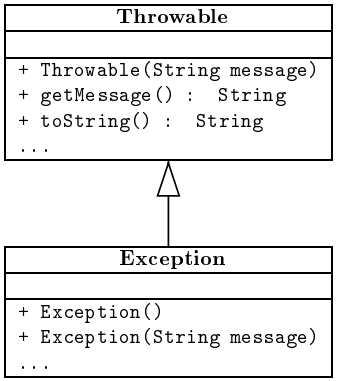
- Déclenchement : levée d'une exception (throw)

```
throw e;
```

★ Note : création et déclenchement sont souvent combinés

```
throw new RuntimeException("Division par zéro");
```


Les classes Exception et Throwable



- La classe `Exception` contient un constructeur prenant un paramètre : le message de l'exception
- Sa classe mère `Throwable` contient :
  - une méthode `getMessage()` qui permet de récupérer le message de l'exception
  - une redéfinition de `toString()` qui retourne : `"NomException: message"`

```
RuntimeException e = new RuntimeException("Division par zéro");

System.out.println(e.getMessage());
// Affiche : "Division par zéro"

System.out.println(e.toString());
// Affiche : "RuntimeException: Division par zéro"
```


Définir ses exceptions personnalisées

On peut définir ses propres exceptions en écrivant une classe qui hérite de la classe standard `Exception`

Exemple : définir des exceptions pour gérer les problèmes de pile

```
// Exception de base pour tous les problèmes de pile
public class PileException extends Exception {
  public PileException(String message) {
    super("Problème de pile : " + message);
  }
}
```

C'est une classe normale, on peut définir des attributs et méthodes pour transporter des informations sur l'exception, redéfinir `toString` et `getMessage`, et définir une hiérarchie d'exceptions :

```
// Exception spécifique pour la pile pleine
public class PilePleineException extends PileException {
  public PilePleineException() {
    super("Pile pleine");
  }
}
PilePleineException ppe=new PilePleineException();
System.out.println(ppe.getMessage()); //"Problème de pile : Pile pleine"
```

Déclenchement d'exceptions

Les instructions qui déclenchent des exceptions :

- sont souvent de la forme :

```
if (probleme) {
    throw new UneException("il␣y␣a␣un␣problème␣!!!");
}
```

Exemple :

```
if (pilePleine()) {
   throw new PilePleineException();
}
```

- cela peut être aussi un appel de méthode

```
ArrayList<Double> arr = new ArrayList<Double>();
arr.add(1.5); arr.add(2.5);
Double x=arr.get(5); // déclenche IndexOutOfBoundsException
```

△ le mot clef `throw` n'est pas toujours écrit

- ou un autre type d'instruction, par exemple, un accès à une case d'un tableau

```
int[] tab = new int[3];
tab[5] = 10; // déclenche ArrayIndexOutOfBoundsException
```

★ Comment faire pour éviter que le programme s'arrête ?

Capture d'exceptions : `try ... catch`

- Mise sous surveillance d'un ensemble d'instructions qui peuvent déclencher une ou plusieurs exceptions
  - ♦ bloc `try ... catch`
- Idée : si l'exécution de ces instructions produit une exception connue alors :
  - ♦ capturer l'exception
  - ♦ proposer un traitement approprié

Syntaxe de base

```
try {

   // Instructions à exécuter sous surveillance :
   // est-ce qu'une exception de type NomException
   // est déclenchée ?

} catch (NomException e) { // Si oui, capture de l'exception

   // Traitement à effectuer si l'exception a été capturée

}
// Instructions à exécuter une fois le bloc try
// ou le bloc catch terminé
```


Capture d'exceptions : exemple (1/2)

Exemple :

```
ArrayList<Double> arr = new ArrayList<Double>();
arr.add(1.5);
arr.add(2.5);
int valUtilisateur=5;

Double x=arr.get(valUtilisateur);
// Exception in thread "main"
//    java.lang.IndexOutOfBoundsException: Index: 5, Size: 2
```

★ Comment faire pour qu'au lieu que le programme s'arrête sur une exception, `x` prenne la dernière valeur de l'`ArrayList` ?

```
Double x=null;
try {
   x=arr.get(valUtilisateur);
} catch(IndexOutOfBoundsException e) {
   x=arr.get(arr.size()-1);
}
// Qu'il y ait une exception ou pas, le programme continue
```


Capture d'exceptions : exemple (2/2)

```
Double x=null;                                    // arr={1.5, 2.5}
try {
   System.out.println("avant␣get");
   x=arr.get(valUtilisateur);
   System.out.println("après␣get");
} catch(IndexOutOfBoundsException e) {
   System.out.println("dans␣le␣catch");
   x=arr.get(arr.size()-1);
}
System.out.println("Le␣programme␣continue␣avec␣x="+x);
```

★ Quel est l'affichage obtenu en fonction de `valUtilisateur` ?

- si `valUtilisateur=0` : l'exception n'est pas levée

```
avant get
après get
Le programme continue avec x=1.5
```

△ le bloc `catch` n'est pas effectué

- si `valUtilisateur=5` : l'exception est levée

```
avant get
dans le catch
Le programme continue avec x=2.5
```

△ toutes les instructions dans le bloc `try` après l'instruction qui lève l'exception ne sont pas exécutée

Attention aux portées des variables

Les blocs limitent la portée des variables

△ Dans le catch : pas d'accès aux variables déclarées dans le `try`

Problème de compilation :

```
try {
    int i = 7;
    ...
} catch (Exception e) {
    System.out.println(i); // ne compile pas: i n'existe pas!
}
```

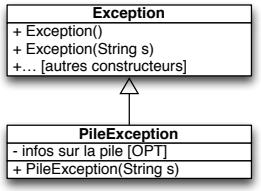
Bonne solution :

```
int i = 0; // déclaration avant
try {
    i = 7; // initialisation ici
    ...
} catch(Exception e) {
    System.out.println(i); // OK
}
```


Capture et hiérarchie d'exceptions 1/3

```
try {
    // dans le cas où la Pile p est vide...
    p.depiler(); // lève une PileException
} catch (PileException e) {
    System.out.println(e.getMessage()); // Méthode de l'exception
}
```

△ La variable e référence l'objet correspondant à l'exception



Une `PileException` EST UNE `Exception`...
Le principe de subsomption s'applique

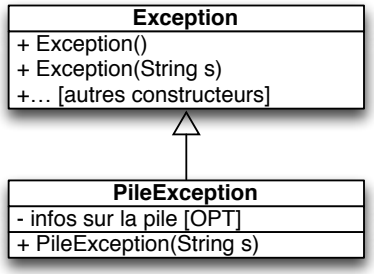
```
Exception e=new PileException("pile vide");
```

```
try {
    // dans le cas où la Pile p est vide...
    p.depiler(); // lève une PileException
} catch (Exception e) { // PileException est attrapée
    System.out.println(e.getMessage());
}
```

- `PileException` est attrapée, car c'EST UNE `Exception`

△ `catch(Exception e)` capture toutes les exceptions

Capture et hiérarchie d'exceptions 2/3



- Possibilité de faire plusieurs `catch` : traitement séquentiel, le premier qui correspond est utilisé (et les autres non)
- Possibilité de raffiner le traitement en fonction du type de l'exception

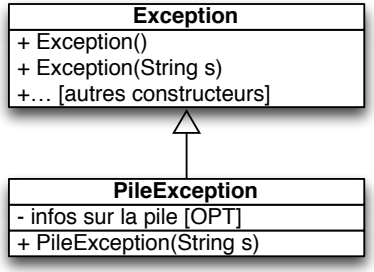
```
try {
    // dans le cas où la Pile p est vide...
    p.depiler(); // lève une PileException

} catch (PileException e) { // PileException est attrapée ici
      System.out.println("Traitement adaptée à PileException");

} catch (Exception e) { // On ne passe pas ici si PileException
      System.out.println(e.getMessage());    //  est attrapée

}
```


Capture et hiérarchie d'exceptions 3/3



- On définit obligatoirement les exceptions les plus spécialisées en premier
- Sinon erreur de compilation car code non accessible

Le programme suivant ne compile pas...

```
try {
    // dans le cas où la Pile p est vide...
    p.depiler(); // lève une PileException

} catch(Exception e) {  // Capture TOUTES les exceptions
    ...
} catch(PileException e) { // Code non accessible
    ...
}
```

Exemple : gestion d'une pile d'entiers (1/5)

Rappel

Une pile est une structure LIFO : Last In First Out.
Le dernier mis dans la pile sort en premier.

1 / Première solution sans gestion des erreurs

```
1  public class Pile {
2          private int[] items;
3          private int nbItems; // indice de la 1ère case libre
4
5          public Pile() {
6                  items = new int[10];
7                  nbItems = 0;
8          }
9
10         public void empiler(int item) {
11           items[nbItems++] = item; // syntaxe compacte
12         }
13
14         public int depiler() {
15           return items[--nbItems];
16         }
17 }
```


Exemple : gestion d'une pile d'entiers (2/5)

Erreur d'utilisation : trop dépiler provoque une erreur

```
31 Pile p = new Pile();
32 for(int i=1; i <= 3; i++)
33     p.empiler(i*10); // 10 20 30
34 int valUtilisateur = 5;
35 for(int i=1; i <= valUtilisateur ; i++)
36     System.out.println("i="+i+"␣"+p.depiler()); // rupture pour i=4
```

Affichage :

```
i=1 30
i=2 20
i=3 10
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException : -1
```

La méthode dépiler a provoqué une rupture de calcul...

- Le programme s'est arrêté
  - Bonne chose ! ! Si le programme continue avec une exécution faussée, l'erreur est plus dure à déceler
- Le message n'est pas clair

⇒ il faut détecter l'erreur et envoyer notre message :
message clair + interruption garantie

Exemple : gestion d'une pile d'entiers (3/5)

2 / Deuxième solution en utilisant des `RuntimeException`

```
1  public void empiler(int item) {
2    if (nbItems >= items.length)
3      throw new RuntimeException("Pile pleine: ajout impossible");
4    items[nbItems++] = item;
5  }
6
7  public int depiler() {
8    if (nbItems <= 0)
9      throw new RuntimeException("Pile vide: extraction impossible"
10   return items[--nbItems];
11 }
```

Affichage (même code que précédemment) :

```
i=1 30
i=2 20
i=3 10
Exception in thread "main" java.lang.RuntimeException:
    Pile vide: extraction impossible
```

- il y a toujours la rupture de calcul qui provoque l'arrêt du programme
- mais le message est plus clair

Exemple : gestion d'une pile d'entiers (4/5)

★ Comment faire pour que la rupture n'arrête pas le programme ?

- Solution : utiliser un `try...catch`
  - ★ Où placer le `try...catch` ?

Solution A : on place le `try...catch` dans le `for`

```
34 int valUtilisateur = 5;
35 for(int i=1; i<= valUtilisateur ; i++) {
36     try {
37         System.out.println("i="+i+" "+p.depiler());
38     } catch (Exception e) {
39         System.out.println("catch i="+i+" msg="+e.getMessage());
40     }
41 }
42 System.out.println("Fin boucle");
```

Quel est l'affichage ?

```
i=1 30
i=2 20
i=3 10
catch i=4 msg=Pile vide : extraction impossible
catch i=5 msg=Pile vide : extraction impossible
Fin boucle
```

★ Pourquoi 2 affichages pour le catch ?

Il y a eu 2 ruptures pour i=4 et i=5, mais la boucle for a continué

Exemple : gestion d'une pile d'entiers (5/5)

Solution B : on place le `try...catch` à l'extérieur du `for`

```
34 int valUtilisateur = 5;
35 try {
36     for(int i=1; i<= valUtilisateur ; i++)
37         System.out.println("i="+i+" "+p.depiler());
38 } catch (Exception e) {
39     System.out.println("catch msg="+e.getMessage());
40 }
41 System.out.println("Fin");
```

Quel est l'affichage ?

```
i=1 30
i=2 20
i=3 10
catch msg=Pile vide : extraction impossible
Fin
```

Le `try...catch` étant à l'extérieur du for, la rupture arrête la boucle for, mais le programme continue

★ Quelle solution choisir ? à l'intérieur ou à l'extérieur ?

Cela dépend de l'application et du résultat que l'on veut obtenir

Gestion d'exceptions : version intégrale

- Surveiller, gérer, et continuer
  - bloc : `try` ... `catch` . .. `finally`

Définition

Le bloc `finally` contient du code qui sera exécuté **dans tous les cas** (qu'il y ait une exception ou non)

Syntaxe

```
1  try {
2      ...
3
4  } catch (UneException e) {
5      ...
6
7  } catch (UneAutreException e) {
8      ...
9
10 } finally {
11     ...
12
13 }
```

- On peut mettre autant de block `catch` que nécessaire

Instruction finally : exemple (1/3)

- Quand il n'y a pas de levée d'exception

```
34 int valUtilisateur = 2 ; // Dans la pile, toujours 10, 20, 30
35 try {
36     for(int i=1 ; i <= valUtilisateur ; i++) {
37         System.out.println("i="+i+"␣"+p.depiler());
38     }
39 } catch (Exception e) {
40     System.out.println("catch␣msg="+e.getMessage());
41 } finally {
42     System.out.println("bloc␣finally␣toujours␣exécuté");
43 }
44 System.out.println("Fin");
```

Quel est l'affichage ?

```
i=1 30
i=2 20
bloc finally toujours exécuté
Fin
```


Instruction finally : exemple (2/3)

- Quand il y a levée d'une exception et un catch pour la capturer

```
34 int valUtilisateur = 5 ; // Dans la pile, toujours 10, 20, 30
35 try {
36     for(int i=1 ; i <= valUtilisateur ; i++) {
37         System.out.println("i="+i+"␣"+p.depiler());
38     }
39 } catch (Exception e) {
40     System.out.println("catch␣msg="+e.getMessage());
41 } finally {
42     System.out.println("bloc␣finally␣toujours␣exécuté");
43 }
44 System.out.println("Fin");
```

L'exception est levée pour i=4. Quel est l'affichage ?

```
i=1 30
i=2 20
i=3 10
catch msg=Pile vide : extraction impossible
bloc finally toujours exécuté
Fin
```

Instruction finally : exemple (3/3)

- Quand il y a levée d'une exception, mais PAS de catch pour la capturer

```
34 int valUtilisateur = 5 ; // Dans la pile, toujours 10, 20, 30
35 try {
36     for(int i=1 ; i <= valUtilisateur ; i++) {
37         System.out.println("i="+i+" "+p.depiler());
38     }
39 } finally {
40     System.out.println("bloc finally toujours exécuté");
41 }
42 System.out.println("Fin");
```

L'exception est levée pour i=4. Quel est l'affichage ?

```
i=1 30
i=2 20
i=3 10
bloc finally toujours exécuté
Exception in thread "main" java.lang.RuntimeException:
    Pile vide: extraction impossible
```

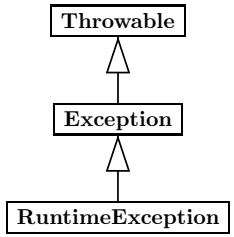
△ Le programme s'est arrêté sur une exception, mais le bloc finally a été exécuté avant

- Usage du `finally` : principalement pour la fermeture de fichier/socket

Mot clef throws : exceptions et méthode

Les Exception susceptibles d'être levées **dans une méthode** doivent être déclarées dans la signature de la méthode avec throws

```
1 // Si MonException hérite de Exception
2 public void maFonction() throws MonException {
3     ...
4     if(test)
5         throw new MonException("MonMessage");
6     ...
7 }
```



Les RuntimeException sont des exceptions particulières qui ne requièrent pas cette déclaration

```
1 public void maFonction() { // pas de throws
2     ...
3     if(test)
4         throw new RuntimeException("MonMessage");
5     ...
6 }
```


Mot clef throws : exemple (1/2)

Dans le cas d'un exception qui hérite de Exception, la déclaration est obligatoire sinon le code ne compile pas

```
1 public int depiler() { // on a oublié le throws
2   if (nbItems <= 0)
3     throw new PileException("Pile vide : extraction impossible");
4   return items[--nbItems];
5 }
```

A la compilation :

```
error : unreported exception PileException ; must be caught or declared to be thrown
      throw new PileException("Pile vide : extraction impossible")
```

Le compilateur nous donne les 2 solutions possibles :

1. `"must be caught"`, c-à-d mettre un `try...catch` pour capturer l'exception dans la méthode
   - ♦ Pour notre exemple, ce n'est pas la solution adaptée
2. `"or declared to be thrown"`, c-à-d mettre `throws PileException` dans la signature de la méthode, ce sera à la méthode main de gérer l'exception

Mot clef throws : exemple (2/2)

Solution corrigée :

```
1 public int depiler() throws PileException { // On a bien mis le throws
2   if (nbItems <= 0)
3     throw new PileException("Pile vide : extraction impossible");
4   return items[--nbItems];
5 }
```

Maintenant, si on utilise `depiler()` dans une autre méthode (ou dans le main), la gestion de l'exception est **obligatoire** :

```
6 public void maMethode() {
7     Pile p=new Pile();
8     p.depiler();
9 }
```

Erreur à la compilation :

```
error : unreported exception PileException ;
must be caught or declared to be thrown
      p.depiler();
```

Toujours les 2 mêmes solutions :

1/ `"must be caught"`

```
31 public void maMethode() {
32     Pile p=new Pile();
33     try {
34         p.depiler();
35     } catch(PileException e) {
36         System.out.println(e);
37     }
38 }
```

2/ `"declared to be thrown"`

```
41 public void maMethode()
42                 throws PileException {
43     Pile p=new Pile();
44     p.depiler();
45 }
```

Cas particulier

Attention

Si l'exception est traitée localement, la fonction n'est pas susceptible de la lever : pas de déclaration dans ce cas

```
1  // avec MonException extends Exception
2  public void maFonction() { // pas de throws
3      ...
4      try {
5          if(test)
6              throw new MonException("MonMessage");
7          ...
8      } catch (MonException e) {
9          ...
10     }
11 }
```

Dans ce cas, il est impossible que maFonction soit interrompue par une MonException non traitée

Cascade d'Exception

Une Exception peut déclencher d'autres Exception

Mécanisme : Exception e → catch → Exception e2

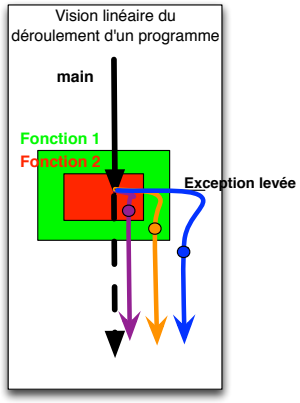
- Pour certains cas particuliers
- Traduction d'exceptions générales vers les exceptions d'un projet particulier

```
1  public void maFonction throws DepassementCapaciteException{
2      try {
3          ...
4
5      } catch(IndexOutOfBoundException e) {
6          ...
7          throw new DepassementCapaciteException();
8      }
9  }
```


Usage : méthode causant une rupture

Idée (logique de délégation)

Si une méthode Fonction1
- utilise une méthode Fonction2 susceptible de lever MonException
- alors Fonction1 est susceptible de lever MonException également



Résumé des différents cas :

1. MonException capturée dans Fonction2 : aucune déclaration nulle part
2. MonException capturée dans Fonction1
   - Fonction2 doit déclarer `throws MonException`
   - Fonction1 ne déclare rien
3. MonException capturée dans main
   - Fonction1 et Fonctions2 déclarent `throws MonException`

NB : suivant les cas, on ne reprend pas l'exécution au programme au même endroit...