

LU2IN002 - Introduction à la programmation orientée-objet

Christophe Marsala



Cours 9 – 25 novembre 2022

PLAN DU COURS

- 1 Exceptions
- 2 Fiabilité du code

RÉACTION EN CAS D'ÉCHEC DU TEST

Que faire ?

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

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

- effectuer une **rupture de calcul**

```
1 ArrayList<Double> arr = new ArrayList<Double>();
2 arr.add(1.); arr.add(2.); arr.add(4.);
3 System.out.println(arr.get(5));
```

```
1 Exception in thread "main"
2 java.lang.IndexOutOfBoundsException: Index: 5, Size: 3
3 at java.util.ArrayList.RangeCheck(ArrayList.java:547)
4 at java.util.ArrayList.get(ArrayList.java:322)
5 at Test.main(Test.java:3)
```

PROGRAMME DU JOUR

- 1 Exceptions
- 2 Fiabilité du code

RAPPELS : ERREURS USUELLES

Execution (JVM) : Toujours vérifier la ligne indiquée

- **NullPointerException**

```
1 Point p = null;
2 p.move(1, 0);
```

```
1 Exception in thread "main" java.lang.NullPointerException
2 at cours1.TestPoint.main(TestPoint.java:2)
```

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

- **IndexOutOfBoundsException**

```
1 int[] tab = new int[3];
2 tab[3] = 2;
```

```
1 Exception in thread "main"
2 java.lang.ArrayIndexOutOfBoundsException: 3
3 at cours1.TestPoint.main(TestPoint.java:2)
```

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



©2022-2023 C. Marsala / S. Tollari

LU2IN002 - POO en Java

4/40

EXCEPTIONS : POINT DE VUE DU PROGRAMMEUR

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

- elle interrompt l'exécution du programme

Elle est utilisée :

- pour éviter les erreurs de calcul
 - division par zéro
 - accès à la référence `null`
 - ouverture d'un fichier inexistant
 - ...
- mais aussi comme style de programmation

En Java une exception est un objet : instance d'une classe

- mécanismes de gestion spécifique
 - `throw` : déclenchement d'une exception
 - `throws` : indication de propagation d'une exception
 - `try ... catch ... finally` : capture de l'exception et traitement



©2022-2023 C. Marsala / S. Tollari

LU2IN002 - POO en Java

5/40



©2022-2023 C. Marsala / S. Tollari

LU2IN002 - POO en Java

6/40

EXCEPTIONS : CRÉATION ET DÉCLENCHEMENT

En Java une exception est un objet

- une exception est une **instance** de la classe **Exception** ...
- ... ou d'une descendante de cette classe

Hérarchie de classes pour les exceptions :

```
1 java.lang.Object
2   extended by java.lang.Throwable
3     extended by java.lang.Exception
4       extended by java.lang.RuntimeException
```

Création d'une exception : création d'une instance

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

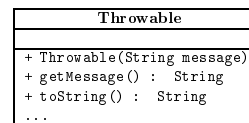
Déclenchement (**throw**) :

```
1 throw e;
```

Rem : création et déclenchement sont souvent combinés

```
1 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"

```
1 RuntimeException e = new RuntimeException("Division par zéro");
2
3 System.out.println(e.getMessage());
4 // Affiche : "Division par zéro"
5
6 System.out.println(e.toString());
7 // Affiche : "RuntimeException: Division par zéro"
```

RÉCUPÉRATION DE L'INTERRUPTION PROVOQUÉE PAR UNE EXCEPTION

- Mise **sous surveillance** d'un ensemble d'instructions
 - bloc **try ... catch**
- Idée** : si l'exécution d'un ensemble d'instructions produit une exception connue alors proposer un **traitement approprié**

Syntaxe de base

```
1 try{
2
3   // Instructions à exécuter sous surveillance
4
5 } catch (NomException e){
6
7   // Instructions pour gérer la levée
8   // d'une exception instance de NomException
9
10 }
11 // à partir d'ici : instructions à traiter une fois le bloc
12 // try terminé ou le bloc catch terminé
```

DÉFINIR SES EXCEPTIONS PERSONNALISÉES

Idée

Une exception est un **objet** : on peut définir ses propres exceptions
⇒ extension de la classe standard **Exception**

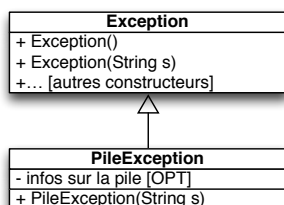
```
1 // Exception de base pour tous les problèmes de pile
2 public class PileException extends Exception {
3   public PileException(String message) {
4     super("Problème de pile : " + message);
5   }
6 }
```

On peut même définir une hiérarchie d'exceptions :

```
1 // Exception spécifique pour la pile pleine
2 public class PilePleineException extends PileException {
3   public PilePleineException() {
4     super("Pile pleine");
5   }
6 }
```

⇒ approche utile comme transport d'informations par les attributs de l'exception personnalisée

RÉCUPÉRATION DES INTERRUPTIONS AVEC DES HIÉRARCHIES D'EXCEPTIONS 1/3

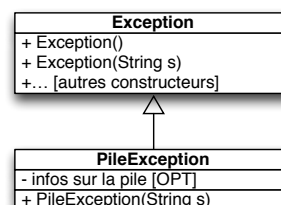


Une **PileException** EST UNE **Exception**...
Le principe de subsumption s'applique.

```
1 try{
2   // dans le cas où la Pile p est vide...
3   System.out.println(p.depiler()); // lève une PileException
4
5 } catch (Exception e){ // PileException est attrapée
6   // car c'est UNE Exception
7   ...
8 }
```

RÉCUPÉRATION DES INTERRUPTIONS AVEC DES HIÉRARCHIES D'EXCEPTIONS 2/3

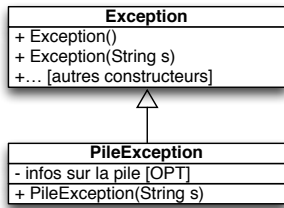
- Le traitement peut dépendre du type de l'exception



Plusieurs **plusieurs catch** sont possibles : traitement séquentiel, **seul le premier** bloc **catch** qui correspond est utilisé

```
1 try{
2   // dans le cas où la Pile p est vide...
3   System.out.println(p.depiler()); // lève une PileException
4
5 } catch (PileException e){ // PileException est attrapée ici
6   ...
7 } catch (Exception e){ // On ne passe pas ici
8   ...
9 }
```

RÉCUPÉRATION DES INTERRUPTIONS AVEC DES HIÉRARCHIES D'EXCEPTIONS 3/3



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

Le programme suivant ne compile pas...

```
1 try {
2     // dans le cas où la Pile p est vide...
3     System.out.println(p.depiler()); // lève une PileException
4 } catch (Exception e) {
5     ...
6 } catch (PileException e) { // Code non accessible
7     ...
8 }
9 }
```

EXEMPLE : GESTION D'UNE PILE D'ENTIERS

Rappel

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

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

PILE : FONCTIONNEMENT (1)

Usage normal :

```
1 public static void main(String[] args) {
2     Pile p = new Pile();
3
4     // Empiler les entiers 0 à 5:
5     for(int i=0; i<6; i++)
6         p.empiler(i);
7
8     // Dépilement:
9     for(int i=0; i<6; i++)
10         System.out.println(p.depiler());
11     // affichage de: 5 4 3 ... 0
12 }
```

PILE : FONCTIONNEMENT (2)

Erreur d'utilisation : **trop dépiler provoque une erreur**

```
1 for(int i=0; i < 6; i++)
2     p.empiler(i);
3
4 for(int i=0; i < 7; i++)
5     System.out.println(p.depiler());
```

```
1 Exception in thread "main"
2 java.lang.ArrayIndexOutOfBoundsException: -1
3   at Pile.depiler(Pile.java:13)
4   at TestPile.main(TestPile.java:14)
```

La méthode **depiler** a provoqué une rupture de calcul...

- arrêt du programme : les instructions suivantes ne sont pas exécutées
 - bonne chose!! : si le programme continuait, l'erreur serait plus dure à déceler
- mais : le message n'est pas clair

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

PILE SÉCURISÉE

```
1 public void empiler(int item){
2     if (prochaine >= items.length)
3         throw new RuntimeException("Pile_pleine: ajout_impossible");
4     items[prochaine++] = item;
5 }
6
7 public int depiler(){
8     if (prochaine <= 0)
9         throw new RuntimeException("Pile_vide: extraction_impossible");
10    return items[--prochaine];
11 }
```

En cas de problème à l'exécution :

```
1 Exception in thread "main" java.lang.RuntimeException:
2   Pile_vide: extraction impossible
3   at Pile.depiler(Pile.java:21)
4   at TestPile.main(TestPile.java:14)
```

EXEMPLE DE PILE AVEC GESTION DES EXCEPTIONS

```
1 Pile p = new Pile();
2 for (int i=0; i<6; i++)
3     p.empiler(i);
4
5 try {
6     for(int i=0; i<8; i++){
7         System.out.println(p.depiler()); // rupture pour i=7
8     }
9
10    // cette instruction ne sera pas exécutée:
11    System.out.println("Est-ce que je passe ici?");
12 }
13 catch (Exception e) { // récupération de la rupture
14     System.out.println("Impossible de dépiler => Exception");
15     System.out.println("Interception de l'erreur et poursuite du pgm");
16 }
17
18 // cette instruction sera exécutée
19 System.out.println("Je suis passé ici");
```

GESTION D'EXCEPTIONS : VERSION INTÉGRALE

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

Syntaxe

```

1 try{
2     // Instructions à exécuter sous surveillance
3 } catch (NomException e){
4     // Instructions pour gérer la levée
5     // d'une exception instance de NomException
6 }
7 // ... // autant de catch que l'on veut
8 finally {
9     // Instructions à toujours réaliser :
10    // - si aucune exception ne se produit
11    // - si une exception traitée se produit, et après son traitement
12    // - après le traitement de toutes les exceptions attrapées
13 }

```

INSTRUCTION finally

Le bloc finally

Le code de ce bloc est exécuté **dans tous les cas**

```

1 Pile p = new Pile();
2 for(int i=0; i<6; i++){
3     p.empiler(i);
4 }
5 try {
6     // on essaye limite = 4 et limite = 8
7     for(int i=0; i<limite; i++){
8         System.out.println(p.depiler());
9     }
10 }
11 catch (Exception e) {
12     System.out.println("Impossible de dépiler => Exception");
13     System.out.println("interception et on continue");
14 }
15 finally {
16     System.out.println("Fin du try ... catch"); // toujours exécuté
17 }

```

LE BLOC finally

```

1 try {
2     // bloc try : instructions surveillées
3 }
4 catch (MonException e) {
5     // bloc catch : gestion de l'exception MonException
6 }
7 finally {
8     // bloc finally : traitement dans tous les cas
9 }
10 // bloc suite : 1ère instruction suivante

```

Différentes possibilités d'exécutions des blocs :

	bloc try	bloc catch	bloc finally	bloc suite
cas 1	normal	×	✓	✓
cas 2	MonException	✓	✓	✓
cas 3	MonException	exception levée	✓	×
cas 4	exception ≠	×	✓	×

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 :

```

1 try {
2     int i = 7;
3     ...
4 }
5 catch (Exception e) {
6     System.out.println(i); // ne compile pas: i n'existe pas!
7 }

```

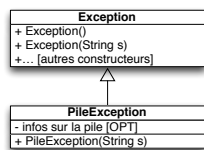
Bonne solution :

```

1 int i = 0; // déclaration avant
2 try {
3     i = 7; // initialisation ici
4     ...
5 }
6 catch (Exception e) {
7     System.out.println(i); // OK
8 }

```

DÉCLARATION D'UNE EXCEPTION



La levée d'une **Exception** dans une méthode doit être déclarée sinon le code ne compile pas :
 ⇒ mot clé : **throws**

Signature d'une méthode qui lève une exception :

public <Retour> fonction(<args>) throws <typeException>

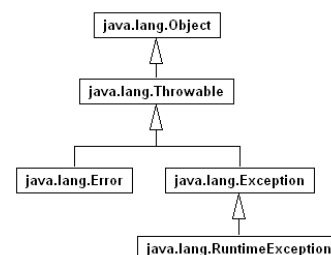
Sur un exemple :

```

1 // La méthode est susceptible de laisser passer une exception PileException
2 // cela se signale dans la signature par throws
3
4 public void empiler(int item) throws PileException {
5     // Si la pile est pleine alors on lève une exception :
6     if (this.prochaine >= this.items.length)
7         throw new PileException("Pile pleine : ajout impossible");
8
9     items[niveau++] = item;
10 }

```

EXCEPTION VS RUNTIMEEXCEPTION



Les exceptions susceptibles d'être levées dans une méthode **doivent être déclarée** dans la signature de la méthode via le mot clé **throws**.

Les **RuntimeException** (et descendantes) sont des **Exception** particulières qui **ne requièrent pas cette déclaration**.

Par exemple :

NullPointerException,
IndexOutOfBoundsException,
ArithmeticException,...

EXEMPLE AVEC LA MÉTHODE MAIN

```
1 // RuntimeException : pas besoin de déclarer un throws
2 public class Essai{
3     public static void main(String[] args) {
4         System.out.println("HelloWorld!");
5         // Ici on lève une exception 'pour le plaisir':
6         throw new RuntimeException("Erreur");
7     }
8 }
```

Compilation OK.

Résultat de l'exécution :

```
1 Hello World!
2 Exception in thread "main" java.lang.RuntimeException: Erreur
3     at Essai.main(Main.java:6)
```

EXEMPLE AVEC LA MÉTHODE MAIN

Mais : pour un autre type d'exceptions :

```
1 // Exception : déclaration throws obligatoire
2 public class Essai{
3     public static void main(String[] args) {
4         System.out.println("Hello");
5         // Ici on lève une exception 'pour le plaisir':
6         throw new Exception("Erreur");
7     }
8 }
```

Echecs de la compilation :

```
1 > javac Essai.java
2 Essai.java:4: error: unreported exception Exception; must be caught
3     throw new Exception("Erreur");
4 1 error
```

EXEMPLE AVEC LA MÉTHODE MAIN

Code Correct pour le compilateur :

```
1 // Exception : déclaration throws obligatoire
2 public class Essai {
3     public static void main(String[] args) throws Exception{
4         System.out.println("HelloWorld!");
5         // Ici on lève une exception 'pour le plaisir':
6         throw new Exception("Erreur");
7     }
8 }
```

Compilation OK.

Résultat de l'exécution :

```
1 Hello World!
2 Exception in thread "main" java.lang.Exception: Erreur
3     at Essai.main(Main.java:6)
```

CAS PARTICULIER

Attention

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

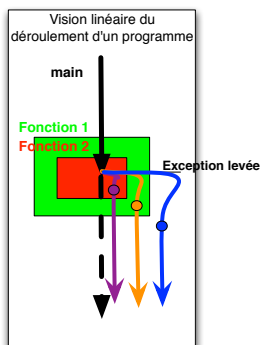
```
1 // avec MonException extends Exception
2
3 public void maFonction(Type args) { // pas de déclaration
4     ...
5     try{
6         ... // traitements
7         if (test)
8             throw new MonException("MonMessage");
9         ... // reste des traitements
10    } catch (MonException e){
11        ... // instructions pour traiter l'exception
12    }
13 }
```

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

USAGE : MÉTHODE CAUSANT UNE INTERRUPTION

Idée (logique de délégation)

Si une méthode **Fonction1** utilise une méthode **Fonction2** susceptible de lever l'Exception **MonException**, alors **Fonction1** est susceptible de lever **MonException** également.



- 1 **Exception traitée dans Fonction2 :**
aucune déclaration nulle part
- 2 **Exception traitée dans Fonction1**
- Fonction2 doit déclarer **throws MonException**
Délégation du traitement
- Fonction1 ne déclare rien
- 3 **Exception traitée dans main**
- Les fonctions déclarent **throws**

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

CASCADE D'EXCEPTION

Une exception peut déclencher d'autres exceptions

Mécanisme : **Exception1 e** → **catch** → **Exception2 e2**

- o pour certains cas particuliers
- o traduction d'exceptions générales vers les exceptions d'un projet particulier

```
1 public void maFonction throws DepassementCapaciteException{
2     try{
3         // gestion d'une structure tabulaire de données
4         // risque de dépassement d'indice
5         ...
6     } catch (IndexOutOfBoundsException e){
7         // traduction d'une Exception Java standard
8         // -> Exception perso
9         throw new DepassementCapaciteException();
10    }
11 }
```

ALTERNATIVE AUX EXCEPTIONS : LES ASSERTIONS

Idée

Simplifier la syntaxe des exceptions

Instruction `assert`

- Syntaxe : `assert expr1 [: expr2]`
 - `expr1` est une expression booléenne
 - `expr2` de type quelconque
 - Exécution :
 - `expr1` est évaluée
 - si `false` ⇒ une `AssertionError` est lancée
 - `expr2`, convertie en chaîne, est utilisée comme message
- `java.lang.Object`
extended by `java.lang.Throwable`
extended by `java.lang.Error`
extended by `java.lang.AssertionError`

Alternative avec des exceptions :

```
1 if (!expr1) {throw new MyException();}
```

ENJEUX DU COURS

Améliorer la fiabilité des programmes développés

Besoin d'outils pour donner confiance dans le code développé...

- 1 Objectif 1 : code compilé = code fonctionnel
 - les **erreurs de compilation** sont les plus faciles à corriger...
 - respectons les règles de développement
 - utilisons des outils pour **provoquer des erreurs de compilation** si le code prend mauvaise tournure...
 - annotations (aide au compilateur)
- 2 Objectif 2 : en cours d'exécution, détectons les erreurs et informons l'utilisateur
 - fonctionnement des ruptures (**exceptions**)
 - déclenchement, traitement...
- 3 Vers la programmation par contrat

EXEMPLE SUR LA VOITURE (1)

Implémenter une voiture = 2 visions

- Vision **client** : accès aux contrôles comme un pilote (pédales, volant)
- Vision **fournisseur** : physique complexe à gérer pour obtenir un comportement réaliste

- 1 Limiter la vision public :
 - commandes pédales/volant ⇒ **public**
 - calcul de la mise à jour de la position/direction, dérapage éventuel ⇒ **private**
- 2 Limiter la taille des méthodes/classes
 - gestion de la physique = moteur physique ou géométrie dans l'espace... ⇒ **classes outils, vecteur, fichier** gérées à part

PLAN DU COURS

- 1 Exceptions
- 2 Fiabilité du code

FIABILITÉ = RESPECT DES RÈGLES DE DÉVELOPPEMENT

Idée

Pour éviter les erreurs, respecter les règles :

- **Choix des noms** pour comprendre qui fait quoi
 - Classes et méthodes de **taille raisonnable**, limiter les accès public
 - les opérations complexes sont déléguées à d'autres classes
 - le client voit peu de choses : facile à comprendre, évite les failles
 - taille limitée = on peut envisager de relire le code de la classe si nécessaire
 - Evolutivité/architecture réfléchie (pour éviter les modifications ultérieures...), usage de **final** (cf. prochain cours)...
- Plus le code est clair, plus les erreurs sont faciles à voir

EXEMPLE SUR LA VOITURE (2)

Idée

- 1 Code morcelé = code plus lisible
- 2 Code morcelé = code testable

Développer plusieurs fonctions **main** pour tester le bon fonctionnement des différentes fonctions basiques

En séparant la gestion de la physique de la voiture, il devient possible de tester chaque fonction du moteur physique...

- plus facile de tester/corriger des méthodes basiques plutôt que l'ensemble d'une méthode très complexe

NB : prémices de la programmation par contrat (cf cours L3 "POO avancée")

VÉRIFICATIONS STATIQUES

Idée

Faire en sorte de vérifier un maximum de chose au niveau de la compilation...

⇒ plus facile à corriger

- Par défaut le **compilateur** vérifie
 - **syntaxe** (les `;`, parenthèses, accolades...)
 - **type** des variables et compatibilité avec les instances et méthodes appelées
 - **niveau d'accès** (aux méthodes, variables...)
- D'autres propriétés sont plus difficiles à montrer et nécessitent plus d'informations transmises au compilateur
 - **langage d'annotations**

ANNOTATIONS STANDARDS

Certaines erreurs ne posent pas de problème de compilation mais provoquent des comportements étranges lors de l'exécution... Ce sont les plus chères à corriger!

- Exemple : on veut redéfinir `toString` dans une classe :

```
1 public class Point {
2     ...
3     public String toString() {
4         return "Point{x=" + x + ",y=" + y + "}";
5     }
6 }
```

- et on écrit ailleurs :

```
1 public class TestPoint {
2     public static void main(String[] args) {
3         Point a = new Point();
4         System.out.println("Mon point: " + a.toString());
5     }
6 }
```

- Quel affichage?
- Pas d'erreur MAIS **problème** lors de l'exécution!

ANNOTATIONS STANDARDS

- **@Override** : pour indiquer une redéfinition de méthode
- ...

Certaines erreurs ne posent pas de problème de compilation mais provoquent des comportements étranges lors de l'exécution... Ce sont les plus chères à corriger!

```
1 @Override
2 public String toString() { // => erreur de compilation
3     return "Point{x=" + x + ",y=" + y + "}";
4 }
```

Provoque une erreur de compilation :

```
1 Point.java:23: method does not override or implement a method
2           from a supertype
3 @Override
4 ^
5 1 error
```

VÉRIFICATIONS DYNAMIQUES

Idées

Une fois la compilation passée, il reste de nombreuses erreurs possibles...

Les tests ne sont pas finis!

- ⇒ faire des tests sur les **arguments** des méthodes pour vérifier la faisabilité
- ⇒ faire des tests sur les **sorties** des méthodes pour vérifier la crédibilité des résultats obtenus

Exemples :

- Tester si la case demandée dans un tableau existe avant de retourner le résultat
- Tester si la valeur de retour de la fonction `carre` est bien positive
- Tester si l'argument de la division est bien différent de 0
- ...