Les exceptions

Maria Virginia Aponte

CNAM-Paris

24 mars 2023

Échecs, erreurs et exceptions.

Erreurs en cours d'exécution

Erreurs à l'exécution

L'exécution d'un programme peut aboutir sur une action « impossible à réaliser » qui se traduit par :

1. l'interruption de cette action
2. pontentiellement l'interruption de tout le programme.

Exemples :

- modifier la case 5 d'un tableau qui n'en contient que 3,
- ouvrir en lecture un fichier qui n'existe pas,
- lire un entier au clavier alors que l'on a tapé `a5b`.

En Java, on utilise les exceptions pour déclencher et gérer certaines erreus survenues pendant l'exécution.

Exemple 1 (TerminalException)

```
1 public static void main (String [] args){
2     Terminal.ecrireString("Un nombre: ");
3     int x = Terminal.lireInt();
4     Terminal.ecrireStringln("Double du nombre: "+ (x*2));
5 }
```

Quelle exécution si l'on saisit « 5f6 » ?

```
> java Arret1
Entrez un nombre: 5f6
Exception in thread "main" TerminalException
        at Arret.main(Arret.java:4)
```

L'exécution de ce programme est interropue par le déclenchement de l'exception `TerminalException`

Exemple 1 : explications

```
1 public static void main (String [] args){
2   Terminal.ecrireString("Un nombre? "); // on tape 5f6
3   int x = Terminal.lireInt();
4   Terminal.ecrireStringln("Double du nombre: "+ (x*2));
5 }
```

1. La méthode main invoque `Terminal.lireInt()` :
   - `Terminal.lireInt()` lit `5f6` (qui n'est pas un entier) ;
   - ne pouvant pas renvoyer un résultat, elle interrompt l'exécution en déclenchant l'exception `TerminalException`,
   - ce qui provoque l'échec (interruption et sortie) de `lireInt()`
2. l'exécution retourne au main (ligne 3) en mode échec ;
3. les instructions lui succédant sont « abandonnées »
4. le programme entier échoue.

Les exceptions

Mécanisme qui permet d'interrompre le cours normal de l'exécution du programme lorsqu'une erreur est détectée et gérer ces interruptions *sur des moments différents*.

1. si on sait résoudre le problème au point où il survient, on n'utilise pas les exceptions. Un *if* suivi du correctif suffissent.
2. les exceptions sont utiles si la méthode qui les détecte est différente de celle qui résoud le problème.

Avec une exception, on peut :

- Déclencher l'exception à l'endroit où l'erreur est detectée $\Rightarrow$ l'exécution du programme est interrompue
- Désactiver (ou *attrapper*) l'exception à l'endroit où l'erreur peut être résolue (traitée) $\Rightarrow$ l'exécution du programme reprend

Lancer, attrapper

En Java on parle de :

- Lancement (*throw*) pour le déclenchement d'une exception
- Rattrapage (*catch*) pour la désactivation d'une exception.

Quelques exceptions prédéfinies

- `NullPointerException` : accès à un champ ou appel de méthode non statique sur un objet valant `null`. Utilisation de `length` ou accès à un case d'un tableau valant `null`.
- `ArrayIndexOutOfBoundsException` : accès à une case inexistante dans un tableau, création d'un tableau de taille négative.
- `StringIndexOutOfBoundsException` : accès au i^{eme} caractère d'un chaîne de caractères de taille inférieure à i.
- `NumberFormatException` : erreur lors de la conversion d'un chaîne de caractères en nombre.

La classe `Terminal` utilise également l'exception `TerminalException` pour signaler des erreurs.

Les méthodes fournissent des services

Les méthodes qui déclenchement et qui traitent une exception doivent être différentes. Une analogie, la boulangerie, pour comprendre. Deux méthodes A (un client), B (le boulanger) :

1. A demande à B de lui fournir une baguette.
2. B constate qu'il n'y en a plus. Il reste peut-être des alternatives mais ce n'est pas à B de décider pour A. B se contente de signaler à A qu'il ne peut pas lui fournir de baguette. B ne peut pas rendre le service attendu.
3. A peut propager cet *échec* vers celui qui lui avait demandé d'acheter une baguette, ou peut décider *traiter et de reprendre* en demandant un autre service à A (reste-t-il du pain ?), etc.

Exemple 1 revisité

```
1 public static void main (String [] args){
2   Terminal.ecrireString("Un nombre? "); // on tape 5f6
3   int x = Terminal.lireInt();
4   Terminal.ecrireStringln("Double du nombre: "+ (x*2));
5 }
```

1. La méthode main invoque `Terminal.lireInt()` :
   - `Terminal.lireInt()` lit `5f6` (qui n'est pas un entier) ;
   - `Terminal.lireInt()` ne peut pas fournir le service attendu (un entier lu en résultat)
   - elle déclenche `TerminalException` et interrompt son exécution ;
2. l'exécution retourne au main (ligne 3) en mode échec ; on propage l'exception vers le main.
3. les instructions lui succédant sont « abandonnées »
4. le programme entier échoue.

Traiter les exceptions

Traiter une exception

Bloc try-catch

Idée : *entourer* le code pouvant lancer une exception par une construction de *rattrapage* d'exceptions, qui prévoit les instructions à exécuter pour traiter ce type d'erreur. Après ce bloc, le cours de l'exécution reprend normalement.

```
try {
        <code-entoure-pouvant-echouer-avec-NomExec>
} catch (NomExc e) {
        <code-de-traitement>
}
<code-apres-bloc-try-catch-suite-normale>
```

Ici, `NomExc` est le nom de l'exception susceptible d'être declenchée par cette exécution.

Cas 1 : le code du bloc-try ne produit pas d'échec

```
try {
        (1) <code-pouvant-echouer>
} catch (NomExc e) {
        (2) <code-de-traitement>
}
(3) <code-apres-try-suite-normale>
```

On exécute le code (1) du bloc-try, ET IL NE lève aucune exception :

1. pas d'erreur à rattrapper-traiter ; on termine le bloc try-catch
2. puis, on continue l'exécution par le code après le bloc try-catch : (3)

L'exécution du code se passe normalement, sans aucune interruption.

Exemple du cas 1 (pas d'échec)

```
boolean calcul = false;
try {
    System.out.print("Un nombre entier: ");
    int x = Terminal.lireInt();
    System.out.println("Son double est: "+ (x*2));
    calcul=true;
} catch (TerminalException e){
    System.out.println("Ce n'est pas un nombre entier!");
}
if (calcul){ System.out.println("On a pu realiser le calcul
else {
   System.out.println("On n'a pas realiser le calcul");  }
```

```
Un nombre entier : 7
Son double est : 14
```

Quelles sont les lignes exécutées ?

Cas 2 : la partie `try` produit un échec

```
try {
        (1) <code-pouvant-echouer>
} catch (NomExc e) {
        (2) <code-de-traitement>
}
(3) <code-hors-try>
```

Cas 2 : on exécute le code (1) du bloc-try, et il LÈVE l'exception E :

1. Le code (1) est interrompu par l'exception E (des instructions sont possiblement abandonnées) ;
2. On compare E avec l'exception du catch (ici, `NomExc`) ;
3. S'il y a correspondance :
   1. On exécute le bloc-catch *(2)*,
   2. puis, on termine NORMALEMENT le bloc try-catch
4. l'exécution se poursuit normalement par le code (3)

Exemple cas 2 : échec+traitement

```
boolean calcul =false;
try {
    System.out.print("Un␣nombre␣entier:␣");
    int x= Terminal.lireInt();
    System.out.println("Son␣double␣est:␣"+ (x*2));
    calcul =true;
} catch (TerminalException e){
    System.out.println("Ce␣n'est␣pas␣un␣nombre␣entier!");
}
if (calcul){ System.out.println("On␣a␣pu␣realiser␣le␣calcu
else {
  System.out.println("On␣n'a␣pas␣pu␣realiser␣le␣calcul");
}
```

```
Un nombre entier : 7.2
Ce n'est␣pas␣un␣nombre␣entier!
On␣n'a pas pu realiser le calcul
```

Cas 3 : un échec se produit et le bloc try-catch échoue

On exécute le code (1) du bloc-try, et il LÈVE l'exception E :

1. Le code (1) est interrompu par l'exception E (il y a possiblement des instructions abandonnées) ;
2. E est rattrapée ;
3. On compare E avec ce qui est attendu par le catch (ici, `NomExc`) ;
4. S'il N'Y A PAS de correspondance :
    1. l'exception E ne peut pas être traitée par le catch ;
    2. le bloc try-catch ÉCHOUE !
    3. E est propagé en DEHORS du bloc try-catch ;
5. le code de la méthode courante échoue et on abandonne la suite (3)

Exemple : un échec se produit, mais pas de traitement

```
boolean calcul = false;
try {
    System.out.print("Un␣nombre␣entier:␣");
    int x= Terminal.lireInt();
    System.out.println("Son␣double␣est:␣"+ (x*2));
    calcul=true;
} catch (IndexOutOfBoundsException e){
    System.out.println("Ce␣n'est␣pas␣un␣nombre␣entier!");
}
if (calcul){ System.out.println("On␣a␣pu␣realiser␣le␣calcu
else {
  System.out.println("On␣n'a␣pas␣realiser␣le␣calcul");  }
```

```
Un nombre entier : 7.2
TerminalException
        at Terminal.exceptionHandler(Terminal.java:115)
```

Expliquez cette exécution

Exemple : récuperer les erreurs de saisie

Lire un entier de manière sécurisée :

- on réalise la lecture dans un try-catch (TerminalException) ;
- ce bloc try-catch est mis dans une boucle ;
- si la lecture réussit, on sort de la boucle, sinon, on y reste

---

```
int x; boolean ok=false;
while (!ok) {
   System.out.println ("Entrez un entier");
   try {
       x = Terminal.lireInt();
       ok=true;
   } catch (TerminalException e){
       System.out.println("Erreur de saisie. Recomencez");
   }
}
System.out.println ("Valeur de x= "+x);
```

Meilleure solution : avec une méthode

Une méthode de lecture validée : elle renvoie toujours un entier. On pourra l'employer à chaque fois que l'on veut lire un entier sans riquer de planter !

```
static int saisieInt(String message){
 int res; boolean ok=false;
 while (!ok) {
   System.out.println(message);
   try { res = Terminal.lireInt();
         ok=true;
   } catch (TerminalException e){
       System.out.println
           ("Erreur de saisie. Recomencez");
   }
 }
 return res;
}
```

Exemple d'appel à la méthode de lecture validée

```
public static void main(String [] args) {
   int taille = saisieInt("Taille du tableau?");
   int [] t = new int [taille];
   for (int i=0; i<t.length; i++){
     t[i] = saisieInt("Element "+ (i+1)+ "? ");
   }
}
```

On peut employer la méthode plusieurs fois, par exemple, pour saisir de manière sécurisée, la taille d'un tableau et ensuite, les éléments du tableau.

Comment s'exécute ce programme ?

Rattraper plusieurs exceptions

```
public static void P () {
 int x = Terminal.lireInt();
 if (x >0){ throw new Stop2();}
}
public static void main(String [] args) {
 Terminal.ecrireStringln("Coucou␣1");        // 1
 try { P ();
       Terminal.ecrireStringln("Coucou␣2"); // 2
 }catch (Stop e){Terminal.ecrireStringln("Coucou␣3");//3
 }catch (Stop2 e){Terminal.ecrireStringln("Coucou␣3bis");//
 }
 Terminal.ecrireStringln("Coucou␣4");    // 4
}

class Stop extends RuntimeException {}
class Stop2 extends RuntimeException {}
```

Déclencher (throw)

Comment déclencher une exception ?

1. *Créer avec* `new` *un objet* du type d'exception pertinent pour le cas d'erreur. Ici, `RuntimeException` est un type d'exception prédéfinie.
2. *Lever*, (lancer, déclencher) cet objet (via `throw`), si les conditions de l'erreur sont réunies (suite à un test, donc)

```
if (...) throw new RuntimeException();
```

Conséquences d'un déclenchement

Le déclenchement d'une exception interrompt l'exécution du programme :

- Une partie du code qui devait suivre est abandonnée ;
- L'exécution peut reprendre si un traitement de l'exception est trouvé ;
- L'exécution est définitivement interrompue sinon.

Quand déclencher une exception ?

Si une fonction ne peut pas retourner un résultat correct, on peut à la place, lever une exception pour signaler l'erreur.

- `pgTabInt(t)` doit retourner le plus grand du tableau t ;
- si t est égal à null ou si t n'a aucune case ⇒ impossible

```
public static int pgTabInt(int [] t) {
    if (t==null || t.length==0)
        throw new IllegalArgumentException();
    int max=t[0];
    for (int i=0; i<t.length; i++) {
        if (t[i]>max)
            max=t[i];
    }
    return max;
}
```

Suite de l'exemple

```
public static void main(String[] args) {
    int [] t1 = {1, 7, 4, 2};
    System.out.print("Le plus grand : ");
    System.out.println(pgTabInt(t1)); // affiche 7
    t1 = new int[0];
    System.out.print("Le plus grand : ");
    System.out.println(pgTabInt(t1)); // echoue
    System.out.println("Jamais execute");
  }
```

```
Le plus grand : 7
Le plus grand :
Exception in thread "main"
  java.lang.IllegalArgumentException
  at demoChapExceptions.ExempleLevee1.pgTabInt(ExempleLevee
  at demoChapExceptions.ExempleLevee1.main(ExempleLevee1.ja
```

Échec d'un sous-programme

Si, pendant l'exécution d'une méthode P, une exception E survient :

1. interruption de l'exécution du code de P,
2. si aucun rattrapage-traitement de E n'est trouvé dans P :
   - abandon des instructions de P restant à exécuter $\Rightarrow$
     1. P se termine anormalement, on dit que P échoue ;
     2. on retourne vers la méthode qui a appelé P en propageant E ;
     3. Dans la méthode appelante M, on recommence le processus : (1) on cherche un traitement dans le code de M ; (2) si non trouvé, échec de M et propagation vers la méthode G appelant M ;

Un déclenchement exception peut se propager dans la chaîne d'appels et aboutir (transitivement) à l'échec au programme entier.

Définir ses exceptions

Définir ses propres exceptions

```
class NouvelleException extends ExceptionDejaDefinie {}
```

- Une exception est *un objet* : crée via une classe.
- mot clé **extends** $\Rightarrow$ notion d'*héritage* (vue plus tard).
- `NouvelleException` : nom de l'exception que lon défini.
- `ExceptionDejaDefinie` est une exception définie auparavant.

Exemple

L'exception `Error` est prédéfinie en Java

```
class PasDefini extends Error {}
```

Exceptions prédéfinies en Java

Trois catégories :

- Dérivées de `Error` : erreurs critiques.
- Dérivées de `Exception` : erreurs à gérer “obligatoirement”.
- Dérivées de la classe `RuntimeException` : erreurs pouvant ou non être gérées.

Dérivées de `Error`

- Représentent des erreurs critiques : lorsque cela arrive, le programme ne peut plus continuer son exécution.
- Ne sont pas censées être gérées dans le programme.

Exemple : `OutOfMemoryError` est levée lorsqu'il n'y a plus de mémoire disponible dans le système.

Erreur ciritique : plus de mémoire, plus d'exécution possible.

Dérivées de `Exception`

- représentent les erreurs non critiques,
- qui doivent normalement être gérées par le programme.

Exemple : une exception de type `IOException` est levée en cas d'erreur lors d'une entrée sortie.

Le programme doit normalement prévoir un mécanisme de gestion : message d'erreur, nouvelle saisie, etc., mais **pas s'interrompre** définitivement.

Dérivées de `RuntimeException`

- elles représentent des erreurs pouvant ou non être gérées par le programme.
- Exemple typique : `NullPointerException`, est levée si l'on tente d'accéder au contenu d'un tableau ou d'un objet qui vaut `null`.

Exceptions : règles importantes

Le mécanisme d'exécution et le raisonnement sur les programmes avec exceptions est compliqué :

- N'utiliser les exceptions qu'en des cirsconstances exceptionelles et seulement si les solutions avec tests ne sont pas viables ;
- Ne jamais lever et ratrapper une exception dans la même méthode : on aurait pu faire autrement avec un test ! inexistante dans un tableau, création d'un tableau de taille négative.
- Définir ses propres exceptions lorsque cela facilite la lecture du programme ;
- Favoriser les solutions simples et homogènes.