# Exceptions

**Université Française d'Egypte**
Version 5.18.1 – 22/11/20
Richard Grin

1

## Plan du support


- Présentation
- Mécanisme de traitement des exceptions
- Clause `finally`
- `try` avec ressources
- Types d'erreurs et exceptions
- Lancer une exception avec `throw`
- Créer une nouvelle classe d'exception
- Compléments sur les exceptions
- Choix des exceptions à lancer




2

## Fiabilité d'un logiciel

- Robustesse : fonctionne, même en présence d'événements exceptionnels
- Correction : donne les résultats corrects lorsqu'il fonctionne normalement



3

## Fiabilité en Java

- Pour avoir une bonne fiabilité :
  - exceptions pour la robustesse
  - assertions pour la correction
  - journalisation (*logging)* `java.util.logging`
  - débuggeurs
  - outils de tests d'unité (comme JUnit) ou de tests fonctionnels



4

## Erreurs/exceptions

- En Java, une erreur à l'exécution déclenche la création d'un objet, sans arrêt brutal du programme
- Cet objet est une instance d'une classe spéciale, une classe d'erreur ou d'exception



5

## Localisation du traitement des erreurs/exceptions

- Le traitement des exceptions s'effectue à part dans une zone spéciale (bloc `catch`)
- Le traitement « normal » apparaît ainsi plus simple et plus lisible



6

## Traitement normal

```
// lire la valeur de n au clavier;
int n = lireEntierAuClavier();
etagere.add(livre, n);
```



7

## Traitement avec exception

```
try {
  // lire la valeur de n au clavier;
  int n = lireEntierAuClavier();
  etagere.add(livre, n);
}
catch(NumberFormatException e) {
  // Traiter le cas « n n'est pas un nombre »
  ...
}
catch {EtagerePleineException e) {
  // Traiter le cas d'une étagère pleine
  ...
}
```



8

## Vocabulaire

- Une instruction, une méthode peut lever ou lancer une exception : une anomalie de fonctionnement provoque la création d'une exception
- Une méthode peut attraper, traiter une exception par une clause `catch`
- Une méthode peut laisser se propager / remonter une exception :
  - elle ne l'attrape pas
  - l'erreur est transmise à la méthode appelante (qui peut la traiter ou la laisser se propager)



9

## Mécanisme de traitement des exceptions



10

## Exception levée en dehors d'un bloc `try`

1. La méthode retourne immédiatement (les instructions de la méthode qui suivent l'instruction qui a lancé l'exception ne sont pas exécutées) ; l'exception est transmise à la méthode appelante
2. L'exception peut ensuite être attrapée par une des méthodes de la pile d'exécution



11

## Exception levée dans un bloc `try`

- Quitte le bloc `try` (les instructions du bloc `try` qui suivent l'instruction qui a lancé l'exception ne sont pas exécutées)
  - si une clause `catch` correspond à l'exception,
    1. la **première** clause `catch` appropriée est exécutée
    2. l'exécution se poursuit juste après le bloc `try-catch`
  - sinon,
    1. la méthode se termine
    2. l'exception est transmise à la méthode appelante



12

## Exécution sans exception d'un bloc `try`

- L'exécution du bloc `try` se déroule comme s'il n'y avait pas de bloc `try-catch`
- Le programme se poursuit après le bloc `try-catch`



13

## Exemples de traitements dans un bloc `catch`

- Fixer le problème et réessayer le traitement qui a provoqué le passage au bloc `catch`
- Faire un traitement alternatif
- Réparer partiellement le problème et relancer (`throw`) une exception (la même ou une autre)
- Retourner (`return`) une valeur particulière



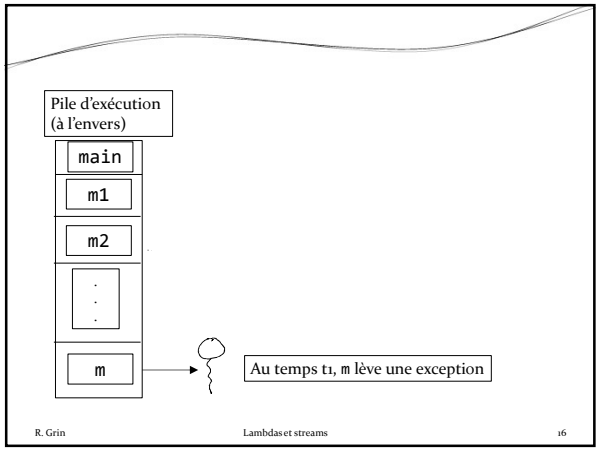
14

## Laisser se propager une exception

- Plus une méthode est éloignée de la méthode `main` dans la pile d'exécution, moins elle a une vision globale de l'application
- Une méthode peut laisser se propager une exception si elle ne sait pas comment la traiter, en espérant qu'une méthode appelante en saura assez pour la traiter

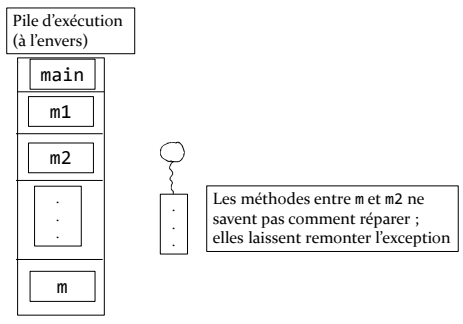


15



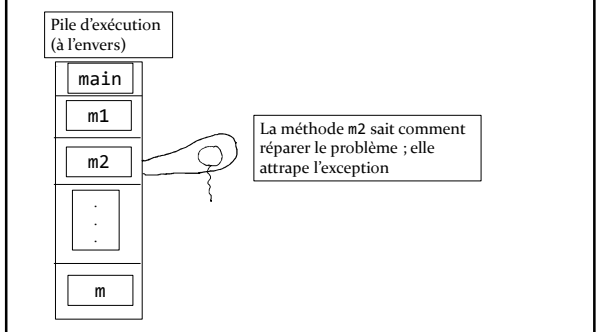



16






17






18

## Exception non traitée

- Si une exception n'est attrapée par aucune méthode,
  - le programme est stoppé
  - le message associé à l'exception est affiché
- Une exception doit donc être attrapée avant qu'elle ne sorte de la méthode `main` pour que le programme ne soit pas stoppé brutalement



19

## Savoir lire un message d'erreur

- Lire attentivement et comprendre un message d'erreur fait gagner beaucoup de temps pour corriger du code
- Un message d'erreur
  - Commence par le message d'erreur associé à l'erreur
  - Ensuite, la pile des méthodes traversées (avec les numéros de ligne) depuis la méthode `main` pour arriver à l'erreur
  - La méthode `main` en dernier



20

## Exemple

```
Exception in thread "main"
java.lang.ArrayIndexOutOfBoundsException: Index 3 out of
bounds for length 3
at eg.ufe.codecorrecteur.Bloc.getValeurDecodée(Bloc.java:25)
at eg.ufe.codecorrecteur.Bloc.decoder(Bloc.java:43)
at eg.ufe.codecorrecteur.TestBloc.main(TestBloc.java:45)
```

Quel est le type de l'erreur ?
Quelle ligne du code a lancé l'exception ?
Dans quelle méthode ?



21

## Déclaration incorrecte

```
try {
  int n;
  n = Integer.parseInt(args[0]);
  etagere.add(livres[n]);
}
catch(NumberFormatException e) {
  System.err.println("Mauvais emplacement " + n);
  return;
}
catch {EtagerePleineException e) {
  System.err.println("Etagere pleine");
  return;
}
n++;
```

Quel est le problème ?

Comment corriger ?



22

## Déclaration correcte

```
int n = 0;
try {
  n = Integer.parseInt(args[0]);
  etagere.add(livres[n]);
}
catch(NumberFormatException e) {
  System.err.println("Mauvais emplacement " + n);
  return;
}
catch {EtagerePleineException e) {
  System.err.println("Etagere pleine");
  return;
}
n++;
```



23

## Erreur fréquente à éviter

- **A NE PAS FAIRE !** Pourquoi ?
  - attraper une exception
  - afficher un message du type « Erreur ! » dans le bloc `catch`
- Le minimum à faire, pendant le développement uniquement : appeler la méthode `printStackTrace` qui affiche la pile d'exécution au moment du lancement de l'exception



24

## Clause `finally`



25

## Clause `finally`

- Exécutée dans tous les cas, qu'une exception ait été levée ou non, que l'exception ait été saisie ou non

```
try {
  . . .
}
catch (…) {
  . . .
}
finally {
  . . .
}
```

On peut, par exemple, fermer un fichier



26

## Syntaxe pour les exceptions

```
try {
  . . .
}
catch (ClasseException1 e) {
  . . .
}
catch (ClasseException2 e) {
  . . .
}
finally {
  . . .
}
```

Possible d'avoir `try` - `finally` sans `catch`

À quoi ça peut servir ?



27

## `try` avec ressources



28

## `try` avec ressources

- Une ressource : base de données, fichier, connexion réseau,…
- Souvent une ressource doit être ouverte avant d'être utilisée, et doit être fermée quand on a fini de les utiliser
- Les ressources qui sont déclarées dans les parenthèses qui suivent un `try` (`try(…)` ) sont automatiquement fermées à la fin du `try`
- Facilite la fermeture automatique des fichiers, connexion à une base de données,…



29

## Sans `try` avec ressources

```
InputStream in = null;
OutputStream out = null;
try {
  in = new FileInputStream("f1");
  out = new FileOutputStream("f2");
  byte[] buffer = new byte[8192];
  int n; // nombre d'octets lus
  while ((n = in.read(buffer)) >= 0)
    out.write(buffer, 0, n);
} finally {
  if (in != null) in.close();
  if (out != null) out.close();
}
```

Ce code copie le contenu du fichier f1 dans f2

Pourquoi `in` et `out` sont déclarés en dehors du `try` ?

Pourrait-on fermer `in` et `out` à la fin du `try` ?



30

## try avec ressources

```
try (
  InputStream in = new FileInputStream("f1");
  OutputStream out = new FileOutputStream("f2")) {
  byte[] buffer = new byte[8192];
  int n; // nombre d'octets lus
  while ((n = in.read(buffer)) >= 0) {
    out.write(buffer, 0, n);
  }
}
```



31

## Avantages

- Les ressources sont fermées à la sortie du bloc `try`, quoi qu'il arrive
- Le code est plus lisible (surtout si plusieurs ressources doivent être fermées)



32

## Portée des variables

- Les variables déclarées dans les parenthèses d'un `try` avec ressources sont utilisables dans le `try` mais pas dans les `catch` et `finally`



33

## Interface `java.lang.AutoCloseable`

- Les classes qui peuvent être gérées par un `try` avec ressources doivent implémenter cette interface
- C'est le cas de `FileInputStream` et de `FileOutputStream` (liées aux fichiers) et de `Connection` (liée à JDBC/SQL)



34

## Types d'erreurs et exceptions



35

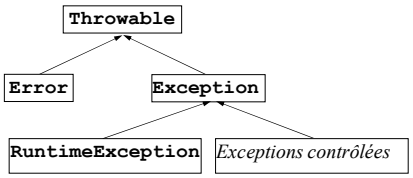
## Les classes d'erreurs/exceptions

- Héritent de la classe `java.lang.Throwable` (qui hérite de `Object`)
- Il y en a beaucoup dans le JDK
- Le programmeur peut en ajouter des nouvelles



36

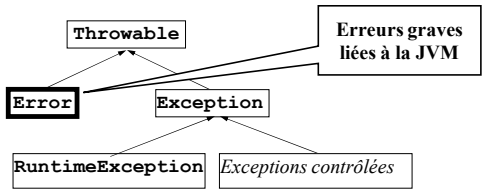
## Arbre d'héritage des exceptions






37

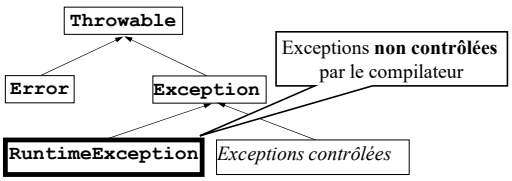
## Arbre d'héritage des exceptions






38

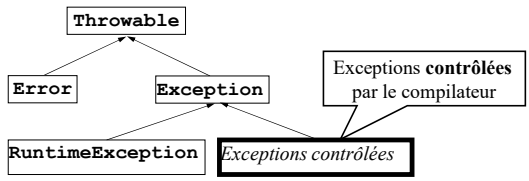
## Arbre d'héritage des exceptions






39

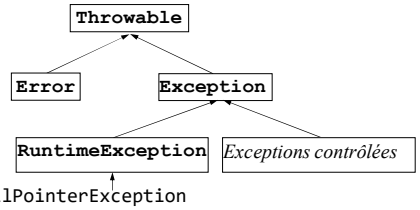
## Arbre d'héritage des exceptions






40

## Sous-classes de `RuntimeException`

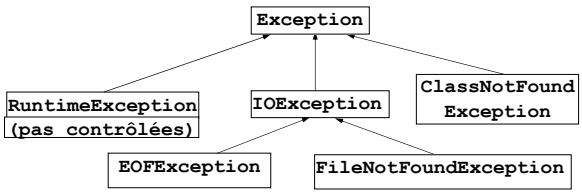



```
NullPointerException
ArrayIndexOutOfBoundsException
ArithmeticException
IllegalArgumentException
NumberFormatException
ClassCastException
```



41

## Quelques exceptions du JDK, contrôlées par le compilateur






42

## Exception « contrôlée » (1/2)

*Checked exception* en anglais

- Exception qui hérite de la classe `Exception` mais pas de `RuntimeException`
- Le compilateur vérifie que les méthodes utilisent correctement cette exception :

  si une méthode peut lancer une exception contrôlée, elle **doit** le déclarer

```
int m() throws TrucException {
  . . .
}
```





## Exception « contrôlée » (2/2)

- Soit la méthode `m2` qui déclare lancer une exception contrôlée :
  … `m2(…) throws TrucException {`
- Si une méthode `m1` fait un appel à `m2`
  - soit `m1` entoure l'appel de `m2` avec un bloc `try-catch(TrucException)`
  - soit `m1` déclare `throws TrucException`





## Pourquoi des exceptions contrôlées ?

- Pour obliger le développeur à tenir compte des cas d'exception
- En langage C, si une méthode retourne -1 pour un cas d'exception, il est facile d'oublier ce cas et de traiter -1 comme les autres valeurs





## Pourquoi des exceptions non contrôlées ?

- Les exceptions non contrôlées peuvent survenir n'importe où, `NullPointerException` par exemple
- Si ces exceptions étaient contrôlées, toutes les méthodes auraient une clause `throws` ou seraient truffées de blocs `try-catch`, ce qui nuirait à la lisibilité





## Exceptions et en-têtes de méthodes

- Une méthode peut déclarer pouvoir lancer plusieurs types d'exception :

```
public Object readObject()
    throws IOException, ClassNotFoundException {
  ...
  ...
}
```





## Regrouper des traitements d'erreurs

- On peut regrouper les traitements des erreurs liées aux sous-classes d'une classe d'exception
- `catch(IOException e) { . . .}` attrape toutes les exceptions liées aux entrées-sorties, qui héritent de `IOException` (`EOFException`, `FileNotFoundException`, …)

## Exception ou traitement normal ?

- Réserver les exceptions aux erreurs ou aux cas exceptionnels
- Éviter de les utiliser pour traiter un cas normal



49

## Exemple

- Si un fichier est lu du début à la fin, ne pas utiliser `EOFException` pour repérer la fin du fichier
- Utiliser plutôt la valeur de retour spéciale de la méthode de lecture quand elle rencontre la fin du fichier, par exemple
  `while ((n = in.read(buffer)) >= 0)`
- Mais si la fin du fichier est rencontrée avant d'avoir lu les 10 valeurs dont on a besoin, utiliser `EOFException`



50

## Constructeurs des exceptions

- Par convention, toutes les exceptions doivent avoir au moins 2 constructeurs :
  - un sans paramètre
  - un autre dont le paramètre est un message (type `String`) qui décrit le problème



51

## Méthodes de la classe `Throwable`

- `getMessage()` retourne le message d'erreur associé à l'instance de `Throwable`
- `printStackTrace()` affiche sur la sortie standard des erreurs (`System.err`), le message d'erreur et la pile des appels de méthodes qui ont conduit au problème



52

## Lancer une exception avec `throw`



53

## Lancer une exception

- Le programmeur peut lancer lui-même des exceptions
  - du JDK
  - ou de classes qu'il a écrites



54

## Lancer une exception du JDK

```
public class Employe {
  . . .
  public void setSalaire(double salaire) {
    if (salaire < 0) {
      throw new IllegalArgumentException(
                        "Salaire négatif !");
    }
    this.salaire = salaire;
  }
```

Création d'une instance



55

## Chaînage de `Throwable`

- Une exception a souvent un constructeur qui prend un `Throwable` en paramètre
- Ainsi, une API peut transformer une exception de bas niveau en exception de plus haut niveau, tout en gardant la cause première de l'exception
- La méthode `getCause()` de `Throwable` permet de récupérer la cause première de l'exception



56

## Exemple API sur les factures

- 
```
FactureException(String message,
                 Throwable cause) {
  super(message, cause);
}
```

`FactureException` hérite de `Exception`

Que fait cette instruction ?

- 
```
try {
  . . . // Fait une requête SQL pour trouver une facture
}
catch(SQLException ex) {
  throw new FactureException(
    "Facture de " + client + " pas trouvée", ex);
}
```

Chaînage de la `SQLException` de bas niveau



57

## Créer une nouvelle classe d'exception



58

## Créer une classe d'exception

- Le programmeur peut créer des classes d'exception adaptées aux classes de l'application
- Par convention, le nom d'une classe d'exception se termine par « `Exception` »



59

## Exemple

```
public class EtagerePleineException
    extends Exception {
  private Etagere etagere;

  public EtagerePleineException(Etagere etg) {
    super("Etagère pleine; " + etg.getNbMax());
    this.etagere = etg;
  }
  . . .
  public Etagere getEtagere() {
    return etagere;
  }
}
```

Référence à l'étagère pleine



60

## Utiliser la nouvelle classe d'exception

```
public class Etagere {
  . . .
  public void ajouteLivre(Livre livre)
      throws EtagerePleineException {
    if (nbLivres >= livres.length)
      throw new EtagerePleineException(this);
    livres[nbLivres++] = livre;
  }
  . . .
}
```

livres et nbLivres sont des variables d'instance de Etagere

Que représente ce this ?



61

## Autre façon d'écrire ajouteLivre

```
public void ajouteLivre(Livre livre)
    throws EtagerePleineException {
  try {
    livres[nbLivres] = livre;
    nbLivres++;
  }
  catch(ArrayIndexOutOfBoundsException e) {
    throw new EtagerePleineException(this, e);
  }
}
```

Quelle est la meilleure solution ?

passe l'exception de bas niveau



62

## Attraper la nouvelle exception

```
// étagère qui peut contenir 10 livres
Etagere etagere = new Etagere(10);
...
try {
  etagere.ajouteLivre(new Livre("Java", "Eckel"));
}
catch(EtagerePleineException e) {
  // Traitement de l'exception (par exemple mettre
  // le livre dans une autre étagère)
  ...
}
```



63

## Compléments sur les exceptions



64

## Exceptions et redéfinition

- Méthode `m` de `B` qui redéfinit une méthode de la classe mère `A`
- `m` ne peut pas déclarer lancer (`throws`) plus d'exceptions que `m` de `A` ; elle peut lancer
  - les mêmes exceptions
  - des sous-classes de ces exceptions
  - moins d'exceptions
  - aucune exception



65

## Exceptions et constructeurs

- Aucune instance n'est créée si une exception est levée par un constructeur



66

## Choix des exceptions à lancer



67

## Error

- Réservé aux erreurs qui surviennent dans le fonctionnement de la JVM
- Par exemple `OutOfMemoryError`
- Ne devrait jamais arriver
- Éviter de lancer une `Error`



68

## RuntimeException

- Lancer une sous-classe de `RuntimeException` est un bon choix si le problème ne peut pas être résolu par une des méthodes appelantes
- Inutile alors d'alourdir le code de ces méthodes avec des `catch` ou des `throws`
- Lié à une erreur indépendante du code qui l'a lancée (passage d'un mauvais paramètre,…)



69

## Si on ne sait pas réparer

- Laisser remonter l'exception jusqu'au début de l'action lancée par l'utilisateur
- La classe qui attrape l'exception
  - l'enregistre comme événement inattendu (*logging*)
  - stoppe proprement l'action qui a provoqué le problème (remet les choses en place)
  - prévient l'utilisateur s'il le faut



70

## Exception contrôlée

- Correspond à un scénario envisagé par le développeur, peu fréquent, mais pas inattendu
- Pour les problèmes qui peuvent être résolus (au moins partiellement) par une des méthodes de la pile d'exécution
- Par exemple, si une étagère est pleine, le code qui a voulu ajouter un livre pourra choisir une autre étagère



71

## Exceptions du JDK ou nouveau type d'exception ?

- Si une exception du JDK convient (bon niveau d'abstraction en particulier), il vaut mieux l'utiliser car ces exceptions sont bien connues des développeurs
- Sinon, écrire une nouvelle classe d'exception



72