Exceptions

Université Française d'Egypte
Version O 5.1 – 24/11/13
Richard Grin

Fiabilité d'un logiciel

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

Fiabilité en Java (1)

- 2 mécanismes principaux facilitent une bonne fiabilité :
  - les exceptions pour la robustesse
  - les assertions pour la correction

Fiabilité en Java (2)

- On peut aussi utiliser
  - API de journalisation (*logging*) ; **`java.util.logging`**
  - débuggeurs
  - outils de tests d'unités (comme JUnit) ou de tests fonctionnels

Erreurs/exceptions

- En Java, une erreur ne provoque pas l'arrêt brutal du programme mais la création d'un objet
- Cet objet est une instance d'une classe spéciale, une classe d'erreur ou d'exception

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

Traitement normal

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


Traitement avec exception

```
try {
  // lire la valeur de n au clavier;
  int n = lireEntierAuClavier();
  etagere.add(livre, n);
}
catch(NumberFormatException e) {
  System.err.println("Mauvais numéro");
}
catch {EtagerePleineException e) {
  System.err.println("Etagere pleine");
}
```


Multi-catch

- Depuis le JDK 7 il est possible d'attraper plusieurs types d'exceptions dans un bloc catch :

```
try {
  …
}
catch(Type1Exception | Type2Exception e) {
  ...
}
```


Vocabulaire (1)

- Une instruction, une méthode peut lever ou lancer une exception : une anomalie de fonctionnement provoque la création d'une exception

Vocabulaire (2)

- Une méthode peut attraper, traiter une exception par une clause **`catch`**
- Une méthode peut laisser se propager une exception :
  - elle ne l'attrape pas
  - l'erreur « remonte » vers la méthode appelante (qui peut la traiter ou la laisser remonter)

Mécanisme de traitement des exceptions

Exception levée en dehors d'un bloc `try`

1. La méthode retourne immédiatement ; l'exception remonte vers la méthode appelante
2. L'exception peut être attrapée par une des méthodes de la pile d'exécution

Exception levée dans un bloc `try`

- Quitte le bloc **`try`** et,
  - 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 retourne
    2. l'exception remonte vers la méthode appelante

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`**

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
- Faire un traitement partiel du problème et relancer (**`throw`**) la même exception, ou une autre
- Retourner (**`return`**) une valeur particulière

Laisser remonter 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 remonter une exception si elle ne sait pas comment la traiter, en espérant qu'une méthode appelante en saura assez pour la traiter

Exceptions non traitées

- Si une exception remonte jusqu'à la méthode **`main`** sans être traitée,
  - le programme est stoppé
  - le message associé à l'exception est affiché, avec une description de la pile des méthodes traversées par l'exception

Déclaration incorrecte

```
try {
  int n;
  n = 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++;
```


Déclaration correcte

```
int n = 0;
try {
  n = 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++;
```


Erreur fréquente à éviter

- **A NE PAS FAIRE !**
  - attraper une exception
  - afficher un message du type « Erreur ! » dans le bloc **`catch`**
- Le minimum à faire : appeler la méthode **`printStackTrace`**

Pourquoi ?

Clause `finally`

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

Traitement général des exceptions

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

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

Prédominance de `finally` sur `try` et `catch`

- Si **`finally`** lance une exception, la méthode lance cette exception
- Si **`finally`** se termine par une instruction **`return`**, la méthode retourne normalement ; aucune exception n'est levée
- Situations à éviter !

`try` avec ressources

`try` avec ressources

- Nouvelle syntaxe du JDK 7
- Facilite la fermeture automatique des ressources : flots, fichiers, connexion sur une base de données,…

Exemple 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;
  while ((n = in.read(buffer)) >= 0)
    out.write(buffer, 0, n);
} finally {
  if (in != null) in.close();
  if (out != null) out.close();
}
```


Exemple avec `try` avec ressources

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


Avantages

- Les ressources sont fermées à la sortie du bloc **`try`**, quoi qu'il arrive
- Le code est bien plus lisible (surtout si plusieurs ressources doivent être fermées) qu'avec un **`try`** – **`finally`** « normal »

Interface `java.lang.AutoCloseable`

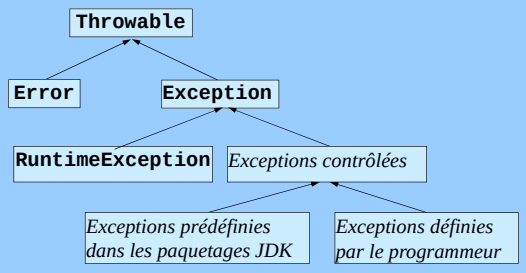
- Les classes qui peuvent être gérées par un **`try`** avec ressources doivent l'implémenter

Les différents types d'erreurs/ exceptions

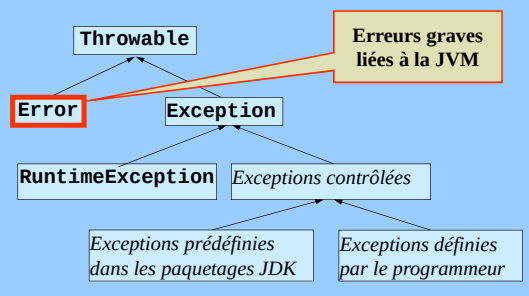
Les classes d'erreurs/exceptions

- Héritent de la classe **`java.lang.Throwable`**
- Le JDK en fournit de nombreuses
- Le programmeur peut en ajouter de nouvelles

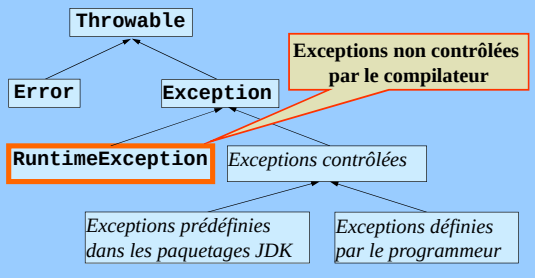
Arbre d'héritage des exceptions



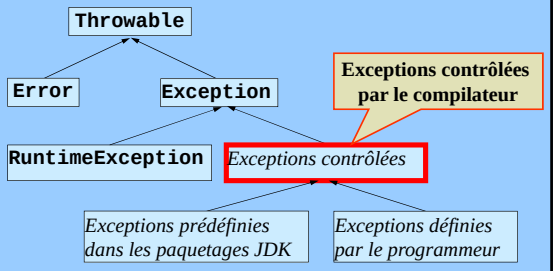
Arbre d'héritage des exceptions



Arbre d'héritage des exceptions



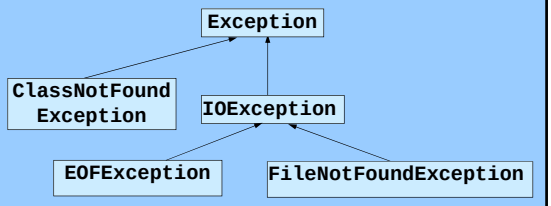
Arbre d'héritage des exceptions



Quelques sous-classes de `RuntimeException`

- `NullPointerException`
- `ArrayIndexOutOfBoundsException`
- `ArithmeticException`
- `IllegalArgumentException` et sa sous-classe `NumberFormatException`
- `ClassCastException`

Quelques exceptions du JDK, contrôlées par le compilateur



Exception « contrôlée » (1)

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

  toute méthode qui peut lancer une exception contrôlée, doit le déclarer dans sa déclaration

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


Exceptions « contrôlées » (2)

- Soit une méthode **`m2`** avec une en-tête « **`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 qu'elle peut lancer une **`TrucException`**

Pourquoi des exceptions contrôlées ?

- Pour obliger le développeur à tenir compte des cas d'exception
- En langage C, si une méthode renvoie -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`**

Exceptions et en-têtes de méthodes

- Une méthode peut déclarer pouvoir renvoyer 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) { . . .}` indique un traitement pour toutes les exceptions liées aux entrées-sorties (**`EOFException`**, **`FileNotFoundException`**, …)

Exception ou traitement normal ?

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

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
- Mais si la fin du fichier est rencontrée avant d'avoir lu les 10 valeurs dont on a besoin, utiliser **`EOFException`**

Exceptions et performances

- Si aucune exception n'est levée, l'impact sur les performances d'un bloc **`try-catch`** est négligeable
- La levée d'une exception peut au contraire avoir un impact non négligeable sur les performances
- Une autre bonne raison de ne pas utiliser les exceptions pour les cas normaux

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 qui décrit le problème

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

Lancer une exception avec `throw`

Lancer une exception

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

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

Quand lancer une exception

- Le plus près possible de l'endroit du code à l'origine du problème
- Par exemple, si une exception est due à un paramètre qui n'a pas une valeur correcte, il faut lancer l'exception au début de la méthode

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 (obtenue par `getCause()`)

Exemple d'une API sur les factures

```
FactureException(String message,
                 Throwable cause) {
  super(message, cause);
}
try {
  . . .
}
catch(IOException ex) {
  throw new FactureException(
    "Facture de " + client + " pas trouvée",
    ex);
}
```

Appel du constructeur de **`Exception`**

Où se trouve ce code ?

Créer une nouvelle classe d'exception

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`** »

Exemple

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

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

Référence à l'étagère pleine

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;
  }
  . . .
}
```

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

Attraper la nouvelle exception

```
// étagère de 10 livres
Etagere etagere = new Etagere(10);
...
try {
  etagere.ajouteLivre(new Livre("Java", "Eckel"));
}
catch(EtagerePleineException e) {
  System.err.println("L'étagère ne peut contenir que "
                     + e.getEtagere().getContenance());
  e.printStackTrace();
}
```


Compléments sur les exceptions

Exceptions et redéfinition

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

Exceptions et constructeurs

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

Choix des exceptions à lancer

`Error`

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

`RuntimeException`

- Bon choix si le problème ne peut être résolu par une des méthodes appelantes
- Inutile alors d'alourdir le code de ces méthodes avec des **`catch`** ou des **`throws`**
- Dû à une erreur indépendante du code qui l'a lancée (passage d'un mauvais paramètre,…)

Traitement des exceptions non contrôlées

- 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

Exceptions contrôlées

- Pour les cas peu fréquents, mais pas inattendus
- Correspondent à des scénarios envisagés par le développeur
- 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

Exceptions du JDK ou nouveau type d'exception ?

- Si une exception du JDK convient, il vaut mieux l'utiliser car ces exceptions sont bien connues des développeurs

Messages d'erreur

Savoir lire un message d'erreur

- Les messages d'erreurs affichés par la JVM correspondent à la sortie de la méthode **`printStackTrace`**
- Lire attentivement et comprendre ces messages fait gagner beaucoup de temps pour corriger du code

Savoir lire un message d'erreur

- Commence par le message d'erreur associé à l'erreur
- Ensuite la pile des méthodes traversée depuis la méthode **`main`** pour arriver à l'erreur
- La méthode **`main`** en dernier