Cours 6 : Exceptions

Les exceptions

Programmation défensive et par contrat

Cours 6 : Exceptions

Les exceptions

- Mécanisme souvent mal compris par les (apprentis) développeurs Java.
- Certainement car utilisé à des fins différentes.

Pourquoi ?

- Mécanisme permettant la **remontée d'erreurs** vers les méthodes appelantes.
- Problèmes d'un langage sans (par ex. le C) :
  - Prévoir une **plage de valeurs** de retours pour signaler les erreurs (problématique si on veut retourner aussi des choses...).
  - **Propager** les erreurs à la main...
  - Le programmeur peut décider d'**ignorer** les codes erreurs !
- Utilisé par Java, C++, Python, etc.

- Si bien utilisé, améliore la lecture, la maintenance et la fiabilité d'un programme.
- Si mal utilisé, effet inverse...

Exemple simple

```
1 public class Coucou {
2   public static void main(String[] args) {
3     System.out.println(args[0]) ;
4     if(Integer.parseInt(args[0]) > 5) {
5       throw new IllegalArgumentException(args[0] + " is > 5") ;
6     }
7   }
8 }
```

- Quelles exceptions possibles ?

Exemple simple

```
1 public class Coucou {
2   public static void main(String[] args) {
3     System.out.println(args[0]) ;
4     if(Integer.parseInt(args[0]) > 5) {
5       throw new IllegalArgumentException(args[0] + " is > 5") ;
6     }
7   }
8 }
```

- Quelles exceptions possibles ?

```
1 $ java Coucou
2 Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException : 0
3   at Coucou.main(Coucou.java :3)
```

```
1 $ java Coucou 10
2 10
3 Exception in thread "main" java.lang.IllegalArgumentException : 10 is > 5
4   at Coucou.main(Coucou.java :6)
```

```
1 $ java Coucou dsqd
2 Exception in thread "main" java.lang.NumberFormatException : For input string : "
      dsqd"
3   at java.lang.NumberFormatException.forInputString(NumberFormatException.java :65)
```

Lancer

- Méthodes du JDK lancent des exceptions...
- Possible d'en lancer également (`throw new BlablaException()`).

Lancer

- Exception est un Objet, possible d'en créer ! Rarement en fait.
- Il vaut mieux utiliser les exceptions existante.
  - Réutilisation du code.
  - Compréhension plus rapide par un relecteur car classes connues.

Exceptions classiques

- Exemples à utiliser :
  - `IllegalArgumentException` (argument négatif, ne respectant pas la doc, etc)
  - `IllegalStateException` (code illégal à cause de l'état de l'objet (non initialisé...), ouvrir une fenêtre déjà ouverte...)
  - `NullPointerException` (parametre passé null interdit)
  - `UnsupportedOperationException` (méthode set d'une implémentation read-only...)

Exceptions classiques

- Exemples à utiliser :
  - `IllegalArgumentException` (argument négatif, ne respectant pas la doc, etc)
  - `IllegalStateException` (code illégal à cause de l'état de l'objet (non initialisé...), ouvrir une fenêtre déjà ouverte...)
  - `NullPointerException` (parametre passé null interdit)
  - `UnsupportedOperationException` (méthode set d'une implémentation read-only...)
- Ne pas hésiter à **remplir l'argument String** pour donner des infos ! (les bornes, l'état d'une variable, etc)

Attraper

- Try/Catch pour attraper une exception.

```
1     int i ;
2     try {
3       i = Integer.parseInt(args[0]) ;
4     }
5     catch (NumberFormatException e) {
6       i = 0 ;
7     }
```

parseInt

```
public static int parseInt(String s)
                    throws NumberFormatException
```

Parses the string argument as a signed decimal integer. The characters in the string must all be decimal digits, except that the first character may be an ASCII minus sign `'-'` (`'\u002D'`) to indicate a negative value. The resulting integer value is returned, exactly as if the argument and the radix 10 were given as arguments to the `parseInt(java.lang.String, int)` method.

Parameters:
`s` - a `String` containing the `int` representation to be parsed

Returns:
the integer value represented by the argument in decimal.

Throws:
`NumberFormatException` - if the string does not contain a parsable integer.

Attraper

- Clause `catch` attrape les sous-types de l'exception déclarée.
  - Donc `catch (IOException e)` attrape aussi `SocketException`.
- L'**ordre compte** : l'exception est "testée" dans l'ordre de chaque clause catch (erreur de compilation si une clause haute en masque une plus basse).

Attraper

- Ne pas tout attraper n'importe comment !

```
1 public static int foo(String[] args) {
2   return Integer.parseInt(args[-1]) ;
3 }
4 public static void main(String[] args) {
5   try {
6     foo(args) ;
7   } catch(Throwable t) {}
8 }
```

Comment s'arracher les cheveux

Attraper

- Ne pas tout attraper n'importe comment !

```
1 public static int foo(String[] args) {
2   return Integer.parseInt(args[-1]) ;
3 }
4 public static void main(String[] args) {
5   try {
6     foo(args) ;
7   } catch(Throwable t) {}
8 }
```

Comment s'arracher les cheveux

- Oublier les `catch(Throwable t)` ou `catch(Exception e)` qui attrapent aussi les `Runtime` !

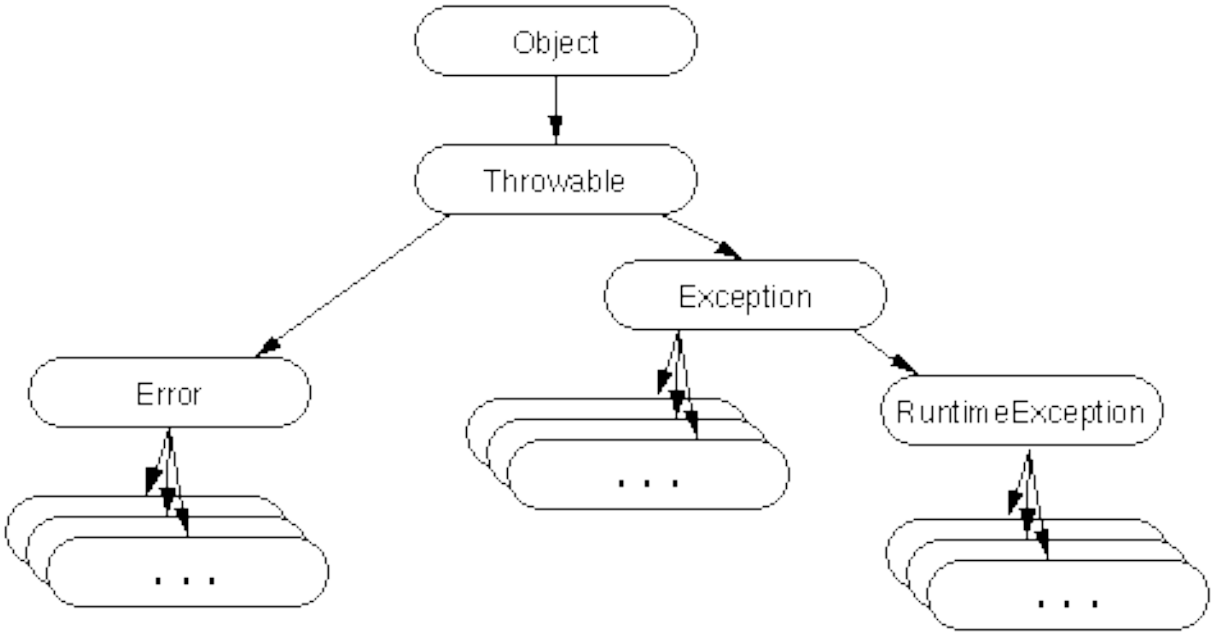
Utilisation multiple...

- Utilisé pour signaler :
  - Des **erreurs de programmation**.
    - Le développeur n'a pas lu la doc, a du mal avec null, les tailles de tableaux...
  - Des **erreurs fatales**.
    - `StackOverflowError, OutOfMemoryError...`
  - Des erreurs dépendant de **conditions externes**.
    - Erreurs d'entrée/sortie, de réseau...

...Traitement multiple !

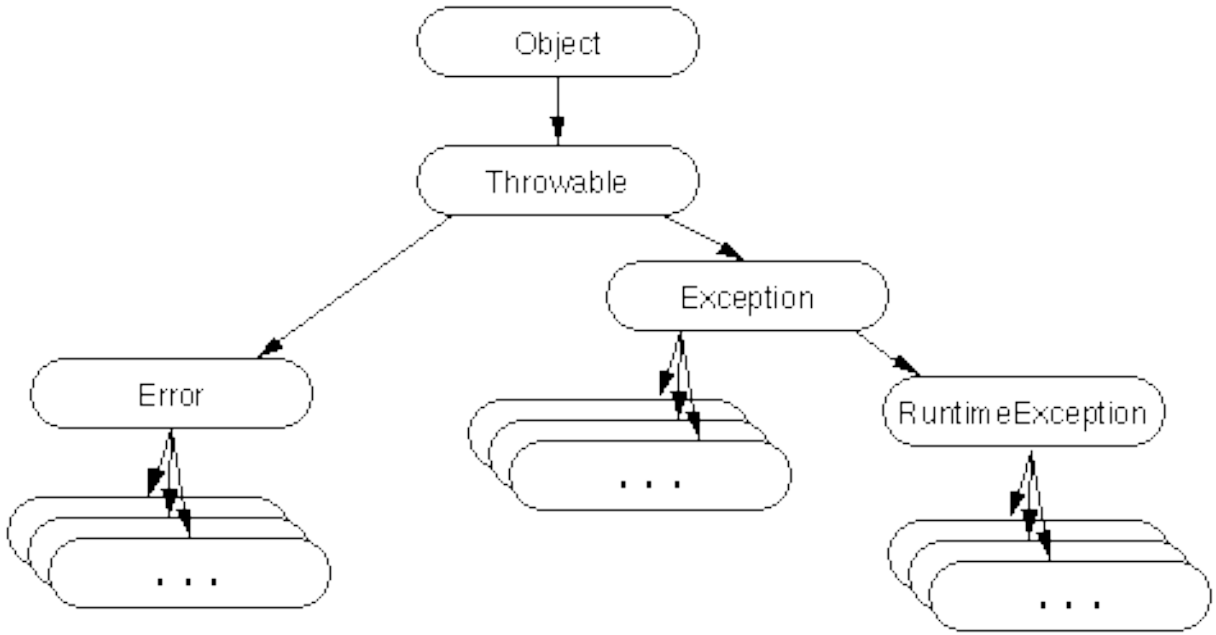
- Des **erreurs de programmation**.
  - Doit arriver uniquement en phase dev et pas en phase prod.
  - Ne pas reprendre l'erreur, mais corriger le bug !
- Des **erreurs fatales**.
  - Erreur de dev ou autre.
  - Ne pas reprendre.
- Des erreurs dépendant de **conditions externes**.
  - Indépendant du programme !
  - Reprendre l'erreur, au moins pour informer l'user.

Types



- `RuntimeException` : pas obligé de les traiter. Erreur de prog.
- `Error` : pas obligé de les traiter (rare de le faire). Erreur fatale.
- `Exception` : obligé de les traiter. Erreur externe.

Types



- `RuntimeException` : pas obligé de les traiter. Erreur de prog.
- `Error` : pas obligé de les traiter (rare de le faire). Erreur fatale.
- `Exception` : obligé de les traiter. Erreur externe.
- Problèmes :
  - `Exception` ne représente pas toutes les ... exceptions !
  - `RuntimeException` hérite d'`Exception` (donc catch(Exception) possible).

2 Types ?

- 2 types :
  - À traiter obligatoirement (checked).
  - Les autres (non-checked).
    - On peut aussi les attraper si on veut.

https://beginnersbook.com/2013/04/java-checked-unchecked-exceptions-with-examples/

Checked

- Java **oblige** à attraper les sous-types de `Throwable` (qui n'est pas `abstract` pour une raison inconnue) qui ne sont pas des sous-types de `Error` ou `RuntimeException`...
  - Soit l'attraper avec try/catch.
  - Soit dire à la méthode appelante de s'en occuper (throws) (penser à remplir le champ `@throws` de la **javadoc** !).

```
1   public void m() {
2     BufferedReader br = new BufferedReader(null) ;
3     try {
4       br.read() ;
5     } catch (IOException e) {
6       //quelque chose
7     }
8   }
9   //la methode qui a appelé m2 devra gérer
10  public void m2() throws IOException {
11    BufferedReader br = new BufferedReader(null) ;
12    br.read() ;
13  }
```

Throws ou catch ?

- Si on appelle une méthode levant une exception non runtime :
  - **Catch** si on peut reprendre l'erreur et faire quelque chose de cohérent.
  - **Throws** sinon : celui qui a appelé sera quoi faire.

Finally

- Ressources liées à des processus limitées par le système.
- Libération pas automatique (non fait par Java).
- Au développeur de fermer !

```
1    BufferedReader br =...
2    String l ;
3    while((l = br.readLine())  != null) {
4      unTrucSurLaLigne(l) ;
5    }
6
7    br.close() ; //argh ! ! Si exception, jamais fermé !
```

Finally

- Exécuter du code quoi qu'il arrive (fermer un fichier, une connexion...)
- Pas obligatoirement avec un catch.

```
1   try {
2     BufferedReader br =...
3     String l ;
4     while((l = br.readLine())  != null) {
5       unTrucSurLaLigne(l) ;
6     }
7   }
8   catch (IOException e) {
9     //report
10  }
11  finally {
12    //br non accessible  !
13    br.close() ;
14  }
```

Finally

- Exécuter du code quoi qu'il arrive (fermer un fichier, une connexion...)
- Pas obligatoirement avec un catch.

```
1   BufferedReader br =...
2   try {
3     //try apres l'assignation
4     String l ;
5     while((l = br.readLine())  != null) {
6       unTrucSurLaLigne(l) ;
7     }
8   }
9   catch (IOException e) {
10    //report
11  }
12  finally {
13    br.close() ; //dans tous les cas, close !
14  }
```

Finally

- Depuis Java7, utiliser plutôt les try-with-resources pour les objets “closable”.
- Syntaxe moins verbeuse.

```
1   try(BufferedReader br =...) {
2     //try avec l'assignation !
3     String l ;
4     while((l = br.readLine()) != null) {
5       unTrucSurLaLigne(l) ;
6     }
7   }
8   catch (IOException e) {
9     //report
10  }//appel àclose automatiquement !
```

Finally

- Peut aussi gérer plusieurs ressources !

```
1  try(BufferedReader br =... ;BufferedWriter bw = ...) { //point virgule
2    String l ;
3    while((l = br.readLine())  != null) {
4      unTrucSurLaLigne(l) ;
5    }
6  }
7  catch (IOException e) {
8    //report
9  }//appels àclose automatiquement dans l'ordre inverse (writer et reader)
```

Runtime ou checked ?

- Idée générale :
  - Si possibilité de réparer l'erreur et documenter l'API, **checked**.
  - Sinon (erreur de programmation, etc), faire planter le programme : **runtime**.

Exceptions

- Utiliser des exceptions... pour des cas exceptionnels !
- Ne jamais faire ça :

```
1 //argh
2 try {
3   int i=0 ;
4   while(true)
5     range[i++].up() ;
6 } catch(ArrayIndexOutOfBoundsException e) {}
```

Exceptions

- Utiliser des exceptions... pour des cas exceptionnels !
- Ne jamais faire ça :

```
1 //argh
2 try {
3   int i=0 ;
4   while(true)
5     range[i++].up() ;
6 } catch(ArrayIndexOutOfBoundsException e) {}
```

- Compile, fonctionne ?

Exceptions

- Utiliser des exceptions... pour des cas exceptionnels !
- Ne jamais faire ça :

```
1 //argh
2 try {
3   int i=0 ;
4   while(true)
5     range[i++].up() ;
6 } catch(ArrayIndexOutOfBoundsException e) {}
```

- Compile, fonctionne ?
- Oui, mais illisible, mauvaise utilise de l'exception, et même plus lent.

Paquet cadeau

- Si redéfinition d'une méthode, la signature joue (pour la levée d'exception, il faut un sous-type ou pas d'exception).

```
1 public interface Runnable {
2   public void run() ;
3 }
```

```
1 public class MiageRunnable implements Runnable {
2   public void run() throws IOException {
3     //compile pas
4   }
5 }
```

Paquet cadeau

- Possibilité d'emballer une exception dans une exception **runtime** pour la ressortir ensuite avec `getCause()` (méthode de Throwable).

```
1  public class MiageRunnable implements Runnable {
2    public void run() {
3      try {
4        // raise IOException
5      } catch(IOException e) {
6        throw new UncheckedIOException(e) ; // wrap
7      }
8    }
9  }
10 public static void main(String[] args) throws IOException {
11   Runnable runnable = new MiageRunnable() ;
12   try {
13     runnable.run() ;
14   } catch(UncheckedIOException e) { // unwrap
15     throw e.getCause() ;
16   }
17 }
```

Cours 6 : Exceptions

Les exceptions

Programmation défensive et par contrat

Programmation défensive et par contrat

- Plus un bug est découvert tard, plus il coûte cher à corriger.
  - Programmation défensive.
- Tous les arguments passés à une méthode publique doivent être validés avant utilisation.
  - Programmation par contrat.

Programmation défensive et par contrat

- On ne fait pas confiance aux arguments passés à un constructeur.
  - En POO, un objet doit toujours être valide.
  - Plus de lectures que d'écritures !
- Vérifier avant de stocker !

Exemple pas bien

```
1  public class Author {
2    private final /*maybe null*/ String firstName ; //null ?
3    private final /*maybe null*/ String lastName ; //null ?
4    public Author(String firstName, String lastName) {
5      this.firstName = firstName ;
6      this.lastName = lastName ;
7    }
8
9    public boolean equals(Object o) {
10     if ( !(o instanceof Author)) {
11       return false ;
12     }
13     Author author = (Author)o ;
14     //check null àchaque lecture  !
15     return ((author.firstName == null && firstName == null) ||
16         author.firstName.equals(firstName) &&
17         (author.lastName == null && lastName == null) ||
18         author.lastName.equals(lastName)) ;
19   }
20 }
```

Exemple bien

```
1  public class Author {
2    private final String firstName ;
3    private final String lastName ;
4    public Author(String firstName, String lastName) {
5      //check null àl'écriture seulement
6      this.firstName = Objects.requireNonNull(firstName) ;
7      this.lastName = Objects.requireNonNull(lastName) ;
8    }
9    public boolean equals(Object o) {
10     if ( !(o instanceof Author)) {
11       return false ;
12     }
13     Author author = (Author)o ;
14     return author.firstName.equals(firstName) &&
15         author.lastName.equals(lastName) ;
16   }
17 }
```

Programmation par contrat

- Une méthode publique doit documenter son contrat.
  - Ce qu'elle fait.
  - Les arguments attendus.
  - Les valeurs de retour.
  - Les exceptions levées et pourquoi.
- Utiliser la javadoc !
  - Doit être à jour par rapport au code.
  - Indiquer comment l'utilisateur se sert de la méthode.
  - Différent d'un commentaire de code (code inhabituel ou non lisible).

Exemple

```
1  public class IntStack {
2    private final int[] array ;
3    private int top ;
4
5    public IntStack(int capa) {
6      array = new int[capa] ;
7    }
8
9    public void push(int v) {
10     array[top++] = v ;
11   }
12   public int pop() {
13     return array[--top] ;
14   }
15 }
```

Exemple - Prog défensive

```
1  public class IntStack {
2    private final int[] array ;
3    private int top ;
4
5    public IntStack(int capa) {
6      if(capa < 0) {
7        throw new IllegalArgumentException("Negative capacity") ;
8      }
9      array = new int[capa] ;
10   }
11
12   public void push(int v) {
13     if(array.length==top) {
14       throw new IllegalStateException("Stack is full") ;
15     }
16     array[top++] = v ;
17   }
18   public int pop() {
19     if (top == 0) {
20       throw new IllegalStateException("Stack is empty") ;
21     }
22     return array[--top] ;
23   }
24 }
```

Exemple - Prog par contrat

```
1   /**
2    * Put an integer on the top of the stack
3    * @param v int to put on top
4    * @throws IllegalStateException if stack is full
5    */
6   public void push(int v) {
7     if(array.length==top) {
8       throw new IllegalStateException("Stack is full") ;
9     }
10    array[top++] = v ;
11  }
```