Programmation Objet. Cours 6

Marie-Pierre Béal et Carine Pivoteau
DUT 1

Exceptions.

Quel est le problème ?

```java
package cours6;

import java.util.List;

public class Test {
  public static int max(List<Integer> integers) {
    int max = 0;
    for (Integer integer : integers) {
      if (integer > max) {
        max = integer;
      }
    }
    return max;
  }

  public static void main(String[] args) {
    List<Integer> integers = List.of(-2, -1, -4, -3);
    System.out.println(max(integers));
  }
}
```

Solution ?

```java
package cours6;

import java.util.List;

public class Test {
  public static int max(List<Integer> integers) {
    int max = integers.get(0);
    for (Integer integer : integers) {
      if (integer > max) {
        max = integer;
      }
    }
    return max;
  }

  public static void main(String[] args) {
    List<Integer> integers2 = List.of();
    System.out.println(max(integers2));
  }
}
```

Non plus...

```
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException:
Index 0 out of bounds for length 0
  at java.base/java.util.ImmutableCollections$ListN
                            .get(ImmutableCollections.java:547)
  at cours6.Test.max(Test.java:7)
  at cours6.Test.main(Test.java:18)
```

Comment empêcher que cela arrive ?

```java
  public static int max(List<Integer> integers) {
    Objects.requireNonNull(integers); // comme ça ?
    int max = integers.get(0);
    for (Integer integer : integers) {
      ...
    }
  }
```

Non, ça ne règle pas le problème qui est la liste vide ...

Signaler à l'utilisateur que la liste ne doit pas être vide !

On peut lancer n'importe qu'elle exception avec `throw`

```
throw new WellChosenException("message d'explication du problème");
```

```
public static int max(List<Integer> integers) {
  if (integers.isEmpty()) { // NullPointerException si integers est null
    throw new IllegalArgumentException("can't use max on an empty list!");
  }
  int max = integers.get(0);
  ...
}
public static void main(String[] args) {
  System.out.println(max(List.of()));
}
```

```
Exception in thread "main" java.lang.IllegalArgumentException:
can't use max on an empty list!
  at cours6.Test.max(Test.java:10)
  at cours6.Test.main(Test.java:24)
```

La classe `Exception` et ses dérivées

Une exception est un objet complexe de la classe `Exception` ou d'une sous-classe, qui, elle-même étend `Throwable`. Un tel objet contient notamment un message, une cause et une pile d'appels de méthodes (*stack trace*).

```
java.lang.Object
 |_ java.lang.Throwable
      |_java.lang.Error
      |_java.lang.Exception
          |_java.lang.ClassNotFoundException
          |_ ...
          |_java.lang.RuntimeException
              |_java.lang.NullPointerException
              |_java.lang.IllegalArgumentException
              |_java.lang.IllegalStateException
              |_java.lang.IndexOutOfBoundsException
              |_ ...
```

Stack trace

```java
public static void one(List<Integer> list) { list.remove(0); }
public static void two(List<Integer> list) { list.remove(0); one(list);}
public static void three(List<Integer> list) { two(list); }

public static void main(String[] args) {
  List<Integer> integers = new ArrayList<>();
  integers.add(1);
  three(integers);
}
```

Lors de la création d'une exception (avec `new`), la pile d'appel des méthodes jusqu'à ce `new()` est conservée... ça peut coûter très cher.

```
Exception in thread "main" java.lang.IndexOutOfBoundsException: Index 0 out of bounds for length 0
  at java.base/jdk.internal.util.Preconditions.outOfBounds(Preconditions.java:64)
  at java.base/jdk.internal.util.Preconditions.outOfBoundsCheckIndex(Preconditions.java:70)
  at java.base/jdk.internal.util.Preconditions.checkIndex(Preconditions.java:248)
  at java.base/java.util.Objects.checkIndex(Objects.java:359)
  at java.base/java.util.ArrayList.remove(ArrayList.java:504)
  at cours6.Test.one(Test.java:8)
  at cours6.Test.two(Test.java:10)
  at cours6.Test.three(Test.java:12)
  at cours6.Test.main(Test.java:17)
```

Intérêt des exceptions

Les exceptions sont un mécanisme inventé pour indiquer un mode de fonctionnement **anormal**.

Certains langages ne disposent pas d'exceptions. Par exemple en C, on utilise les valeurs de retour de fonctions pour signaler d'éventuels problèmes.

Pour **signaler** un problème et/ou **interdire** une action, on **lève une exception**. Cela peut permettre d'arrêter l'exécution du programme.

La levée d'une exception provoque

- la sortie **immédiate** de la méthode ;
- la remontée dans l'*arbre d'appel des méthodes* jusqu'à celle qui *attrape* l'exception ou jusqu'à la sortie du programme.

Différentes utilisations des exceptions/erreurs

1. Signaler des erreurs de programmation :
   - mauvaise utilisation de l'API, mauvaise gestion de `null`, ...
   - débordement dans les tableaux, les collections, ...
   - ...

   Cela ne doit pouvoir arriver que lors de la phase de développement (jamais dans le logiciel final). On ne doit pas essayer d'attraper ce type d'exception mais plutôt corriger le problème qui l'a créée.

   → `RuntimeException` ou dérivée.
2. Signaler des erreurs *fatales* (on ne cherche pas à les attraper) : `StackOverflowError`, `OutOfMemoryError`, `InternalError`...

   → `Error`.
3. Signaler des erreurs indépendantes de notre volonté, en particulier celles qui viennent de "l'extérieur" les problèmes d'entrée/sortie (voir dernier cours), qu'il faut gérer (attraper ou propager).

   → `Exception` (mais pas `RuntimeException`).

Erreur de programmation : quelle exception ?

Ça dépend du contexte. Il y a suffisamment de choix dans les dérivées de `RuntimeException` pour la plupart des cas courants. Par exemple :

```
public class Lists {
  public static int max(List<Integer> integers) {
    Objects.requireNonNull(integers);
    if (integers.isEmpty()) {
      throw new IllegalArgumentException("can't use max on an empty list!");
    }
    int max = integers.get(0);
    ...
  }
}
```

```
public class MyIntList {
  private final ArrayList<Integer> inetegers = new ArrayList<>();

  public add(int i) { integers.add(i); }

  public int size() { return integers.size(); }

  public int max() {
    if (integers.isEmpty()) {
      throw new IllegalStateException("can't use max on an empty list!");
    }
    return Lists.max(integers);
  }
}
```

Quand lancer une exception ?

Là encore, ça dépend du contexte. L'idée c'est qu'une méthode peut lancer une exception pour empêcher des cas d'utilisation frauduleux :

- ils doivent pouvoir être anticipés par l'utilisateur,
- et ça ne doit pas remplacer un booléen ou une valeur par défaut.

Par exemple :

```java
public static void main(String[] args) {
  ArrayList<Integer> integers = new ArrayList<>();
  integers.add(42);
  System.out.println(integers.contains(43)); //false
  System.out.println(integers.indexOf(43));  // -1
  System.out.println(integers.remove((Integer) 43)); // false
  System.out.println(integers.remove(43));  // Exception
}
```

```
Exception in thread "main" java.lang.IndexOutOfBoundsException: Index 43 out of bounds for length 1
  at java.base/jdk.internal.util.Preconditions.outOfBounds(Preconditions.java:64)
  at java.base/jdk.internal.util.Preconditions.outOfBoundsCheckIndex(Preconditions.java:70)
  at java.base/jdk.internal.util.Preconditions.checkIndex(Preconditions.java:248)
  at java.base/java.util.Objects.checkIndex(Objects.java:359)
  at java.base/java.util.ArrayList.remove(ArrayList.java:504)
  at cours6.Test.main(Test.java:55)
```

Comment “gérer” une exception ?

Dans les exemples précédents, on utilise les exceptions pour faire de la programmation défensive : signaler les problèmes le plus tôt possible dans le code, pour faciliter leur détection. **L'objectif**, c'est que **ces exceptions ne soient pas levées !**

```
public static void main(String[] args) {
  ArrayList<String> strings = new ArrayList<>();
  integers.add("bob");
  integers.remove(2); // ArrayIndexOutOfBoundsException
}
```

Comment faire pour empêcher que le programme s'arrête ?
Ici, c'est simple, il ne faut pas utiliser `remove` avec un indice qui dépasse la taille de la liste, sinon, on sait que l'exception sera levée. Et c'est quelque chose de facile à tester.

```
if( integers.size() > 2 ){
  integers.remove(2); // l'exception ne peut pas arriver
}
```

Documenter les exceptions

Lorsque que l'on définit sa propre classe, toute méthode publique doit être documentée ; on doit savoir :

- Ce qu'elle fait.
- Quels sont les arguments attendus ?
- Quels sont les valeurs de retour possibles ?
- Quelles exceptions peuvent être levées et pourquoi ?

Java possède un format de documentation que l'on écrit directement dans le code : la javadoc. Elle est générée automatiquement à partir de commentaires spécifiques (à ne pas confondre avec les commentaires du code destinés au développeur de la classe). Voir l'exemple de `Stack`.

Pour bien utiliser les exceptions, il faut :

- Lire la documentation des classes que l'on utilise.
- Fournir la documentation de ses propres classes pour indiquer à l'utilisateur les exceptions qui sont susceptibles d'être levées.

Écrire de la javadoc d'une méthode

```java
/**
* Description détailée de ce que fait la méthode
*
*  @param nom1 description et contraintes
*  @param nom2 description et contraintes
*  @return (si ce n'est pas void) description succincte
*  @throws TypeException desciption de l'exception succeptible d'être levée
*  @see ...
*/
public Type0 method(Type1 nom1, Type2 nome2){
  ...
  throw new TypeException("humm...")
}
```

Pour une explication détaillée :

`https://www.oracle.com/technical-resources/articles/java/javadoc-tool.html`

Dans Eclipse : Menu Project > Generate Javadoc

Exemple : l'nterface `Stack`

Voici une interface pour une pile d'entiers (LIFO).

```java
package fr.uge.but.info;

public interface Stack {
  /**
   * Tests if this stack is empty.
   * @return true if and only if this stack contains no items;
   * false otherwise.
   */
  boolean isEmpty();

  /**
   * Tests if this stack is full.
   * @return true if and only if this stack can not contain
   * more items; false otherwise.
   */
  boolean isFull();

  ...
```

```
  ...

  /**
   * Pushes an item (int) onto the top of this stack.
   * @param item the element to push
   * @throws IllegalStateException if this stack is full.
   */
  void push(int item);

  /**
   * Looks at the int at the top of this stack without removing it.
   * @return the int at the top of the stack.
   * @throws IllegalStateException if this stack is empty.
   */
  int peek();

  /**
   * Removes the int at the top of this stack and returns that int.
   * @return the int at the top of the stack.
   * @throws IllegalStateException if this stack is empty.
   */
  int pop();
}
```

Package fr.uge.dut.info

Interface Stack

All Known Implementing Classes:

ArrayStack, ListStack

```
public interface Stack
```

Method Summary

All Methods | **Instance Methods** | **Abstract Methods**

Modifier and Type	Method	Description
boolean	**isEmpty**()	Tests if this stack is empty.
boolean	**isFull**()	Tests if this stack is full.
int	**peek**()	Looks at the int at the top of this stack without removing it from the stack.
int	**pop**()	Removes the int at the top of this stack and returns that int.
void	**push**(int item)	Pushes an item (int) onto the top of this stack.

Method Details

isEmpty

```
boolean isEmpty()
```

Tests if this stack is empty.

Returns:

true if and only if this stack contains no items; false otherwise.

isFull

```
boolean isFull()
```

Tests if this stack is full.

Returns:

true if and only if this stack can not contain more items; false otherwise.

push

```
void push(int item)
```

Pushes an item (int) onto the top of this stack.

Parameters:

`item` - the element to push

Throws:

`java.lang.IllegalStateException` - if this stack is full.

peek

`int peek()`

Looks at the int at the top of this stack without removing it from the stack.

Returns:

the int at the top of the stack.

Throws:

`java.lang.IllegalStateException` - if this stack is empty.

pop

`int pop()`

Removes the int at the top of this stack and returns that int.

Returns:

the int at the top of the stack.

Throws:

`java.lang.IllegalStateException` - if this stack is empty.

Deux implémentations possibles de cette interface (`ArrayStack` et `ListStack`) seront faites en TP.

Et si l'exception est imprévisible ?

delete

```
public static void delete(Path path)
                  throws IOException
```

Deletes a file.

An implementation may require to examine the file to determine if the file is a directory. Consequently this method may not be atomic with respect to other file system operations.

Throws:

`NoSuchFileException` - if the file does not exist *(optional specific exception)*

`DirectoryNotEmptyException` - if the file is a directory and could not otherwise be deleted because the directory is not empty *(optional specific exception)*

`IOException` - if an I/O error occurs

```java
public static void main(String[] args) {
  String name = "fichier.txt";
  Path path = Path.of(name);

  try {
    Files.delete(path);
    System.out.println("Le fichier " + name + " a été supprimé");
  } catch (IOException e) {
    System.out.println("Problème lors de la suprresion du fichier " + name);
  }
}
```

Remarque : une meilleure solution, c'est d'utiliser la méthode `Files.deleteIfExists()`.

Capturer une exception par un bloc try / catch

```
try {
  // code qui peut lever une excepetion
} catch (TypeDException e) {
  // code à executer en cas d'exception
}
```

Bloc try / catch

- Le bloc try lance une exécution contrôlée.
- En cas de levée d'exception dans le bloc try, ce bloc est quitté immédiatement, et l'exécution se poursuit par le bloc catch.
- L'argument du bloc catch est l'exception éventuellement levée dans le bloc try.
- Plusieurs catch sont possibles, et le premier dont l'argument est du bon type est exécuté.

Organisation des différents blocs `catch`

```
try {
   ...
} catch (Type1Exception e) {
   ...
} catch (Type2Exception e) {
   ... // attention, Type2 ne doit pas être sous-type de Type1
} catch (Exception e) {...} // cas par défaut, capture les
                            // exceptions non traitées avant
```

Exemple de blocs `catch` mal ordonnés

```
try { ... }
catch (Exception e) {...}
catch (StackException e) {...}
// le 2e bloc catch n'est jamais executé
```

`RuntimeException` ou pas (*checked exceptions*)

`RuntimeException`

Les exceptions dérivant de la classe `RuntimeException` n'ont **pas besoin d'être capturées**.

Exceptions ne dérivant pas de `RuntimeException`

Une levée d'exception se produit lors d'un appel à `throw` ou d'une méthode ayant levé une exception. Ainsi l'appel à une méthode pouvant lever une exception (qui n'est pas une `RuntimeException`) doit :

- être contenu dans un bloc `try / catch` pour capturer l'exception, si on sait la traiter ;
- ou bien être dans une méthode propageant cette classe d'exception (mot-clé `throws`).

Exemple d'utilisation de throws

```
public static void deleteVerbose(Path path) throws IOException{
  Files.delete(path);
  System.out.println("Le fichier " + name + " a été supprimé");
}
```

```
public static void main(String[] args) {
  String name = "fichier.txt";
  Path path = Path.of(name);
  try {
    deleteVerbose(path);
  } catch (NoSuchFileException  e) {
    System.out.println("Le fichier " + name + " n'existait pas");
  } catch (IOException e) {
    System.out.println("Problème d'entrée/sortie sur le fichier " + name);
  }
}
```

Bonus : exceptions personnalisées

On peut vouloir définir ses propres exceptions en dérivant de la classe `RuntimeException` :

```
public class MaxException extends RuntimeException {
  public MaxException() {
    super();
  }

  public MaxException(String msg) {
    super(msg);
  }
}
```

```
public static int max(List<Integer> integers) {
  if (integers.isEmpty()) {
    throw new MaxException("can't use max on an empty list!");
  }
  int max = integers.get(0);
  ...
 }
```