Programmation orientée objet. Cours 5

Marie-Pierre Béal et Carine Pivoteau
BUT 1

Exceptions.

Quel est le problème ?

```
import java.util.List;

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

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

Solution ?

```
import java.util.List;

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

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

Non plus

```
Exception in thread "main" java.util.NoSuchElementException
  at java.base/java.util.List.getFirst(List.java:825)
  at cours5.Test.max(Test.java:7)
  at cours5.Test.main(Test.java:18)
```

Comment empêcher que cela arrive ?

```
public static int max(List<Integer> integers) {
    Objects.requireNonNull(integers);
    var max = integers.getFirst();
    for (var 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 quelle exception avec throw

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

```
public static int max(List<Integer> integers) {  
    // NullPointerException si integers est null  
    Objects.requireNonNull(integers);  
    if (integers.isEmpty()) {  
        throw new IllegalArgumentException("empty list");  
    }  
    var max = integers.getFirst();  
    ...  
}  
public static void main(String[] args) {  
    System.out.println(max(List.<Integer>of()));  
}
```

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

```
Exception in thread "main" java.lang.IllegalArgumentException:  
    empty list  
    at cours5.Test.max(Test.java:10)  
    at cours5.Test.main(Test.java:23)
```

La classe Exception et ses dérivées (voir cours sur la dérivation)

Une **exception** est un objet 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

```
public class Test{
    public static void one(List<Integer> list) {
        Objects.requireNonNull(list);
        list.removeFirst();
    }
    public static void two(List<Integer> list) {
        Objects.requireNonNull(list);
        list.removeFirst();
        one(list);
    }
    public static void three(List<Integer> list) {
        Objects.requireNonNull(list);
        two(list);
    }

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

Stack trace

Lors de la création d'une exception (avec new), on remonte dans la pile d'appel des méthodes.

```
Exception in thread "main" java.util.NoSuchElementException
  at java.base/java.util.ArrayList.removeFirst(ArrayList.java:568)
  at cours5.Test.one(Test.java:10)
  at cours5.Test.two(Test.java:15)
  at cours5.Test.three(Test.java:19)
  at cours5.Test.main(Test.java:25)
```

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("empty list");
        }
        var max = integers.getFirst();
        ...
    }
}
```

Erreur de programmation : quelle exception ?

```
public class MyIntegerList {
    private final ArrayList<Integer> integers = new ArrayList<>();

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

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

    public int max() {
        if (integers.isEmpty()) {
            throw new IllegalStateException("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 :

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

Quand lancer une exception ?

```
false
-1
false
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:100)
    at java.base/jdk.internal.util.Preconditions.outOfBoundsCheckIndex(Preconditions.java:106)
    at java.base/jdk.internal.util.Preconditions.checkIndex(Preconditions.java:302)
    at java.base/java.util.Objects.checkIndex(Objects.java:385)
    at java.base/java.util.ArrayList.remove(ArrayList.java:551)
    at cours5.Test.main(Test.java:28)
```

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) {  
    var strings = new ArrayList<String>();  
    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

```
/**
 * Description détaillée 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 description de l'exception susceptible
 * d'être levée
 * @see ...
 */
public Type0 method(Type1 nom1, Type2 nom2){
    ...
    throw new TypeException("hummm...")
}
```

Pour une explication détaillée :

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

Dans Eclipse : Menu Project > Generate Javadoc

Exemple : l'interface Stack

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

```
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	<code>isEmpty()</code>	Tests if this stack is empty.
boolean	<code>isFull()</code>	Tests if this stack is full.
int	<code>peek()</code>	Looks at the int at the top of this stack without removing it from the stack.
int	<code>pop()</code>	Removes the int at the top of this stack and returns that int.
void	<code>push(int item)</code>	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 ?

Les Checked Exception

```
public static void main(String[] args) {
    var name = "fichier.txt";
    var path = Path.of(name); // voir le cours sur les entrées-sorties
    try {
        Files.delete(path);
        System.out.println("Le fichier " + name + " a été supprimé");
    } catch (IOException e) {
        System.out.println("Problème à la suppression du fichier " + name);
    }
}
```

Remarque : pour une meilleure solution, utiliser la méthode `Files.deleteIfExists()`.

Capter une exception par un bloc `try / catch`

```
try {  
    // code qui peut lever une exception  
} catch (TypeDException e) {  
    // code à exécuter 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 (StackErrorException e) {...}
// le 2e bloc catch n'est jamais exécuté
```

Runtime exception ou checked exception

RuntimeException

Les exceptions dérivant de la classe RuntimeException **ne doivent pas 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) {
    var name = "fichier.txt";
    var 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 (voir dernier cours) de la classe `RuntimeException` :

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

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

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