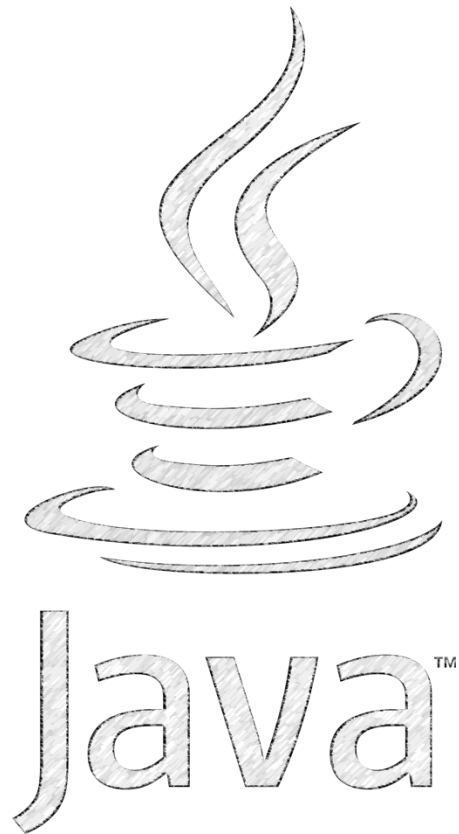


# Programmation Orientée Objets avec Java



## Chapitre 6

### Gestions des incohérences

# Concepts traités

2

- Les exceptions
- L'absence de valeurs
- Le type `Optional<T>`

# Gestion des incohérences

# Gestion des incohérences

4

- La gestion des incohérences consiste à gérer toute situation qui mène notre système dans un état incohérent
- Quelques exemples :
  - récupérer un index d'une lettre dans un `String` même si la lettre ne s'y trouve pas
  - récupérer un élément d'une liste à l'aide d'un index plus grand que la taille de la liste
  - récupérer un utilisateur à l'aide d'un identifiant inconnu
  - planifier une réservation un 30 février
  - créer un utilisateur qui a -33 ans
  - déposer un montant négatif sur un compte
  - attribuer une salle déjà occupée à un professeur
  - ...
- Pour éviter les états incohérents, on doit savoir si le problème peut être géré à l'exécution. Si ce n'est pas le cas, il est préférable de planter avec une erreur visible, c' est la technique du **fail fast**.

# Gestion des incohérences

5

Le meilleur moyen d'identifier un état incohérent est à la compilation.

Un bon typage aide grandement.

Une autre technique est d'utiliser une structure simple pour représenter l'absence de valeur (les `Optional`) plutôt que de produire des valeurs arbitraires.

Enfin, les mécanismes d'exceptions doivent être employés en dernier recours

Ils permettent de séparer la gestion des erreurs de la logique du programme.

# Les exceptions

# Les exceptions

7

L'utilité des exceptions => un exemple sans la gestion des exceptions :

Voici un exemple de pseudo-code contenant des données altimétriques et qui calcule la moyenne des altitudes :

```
1  File file = ???  
2  String content = readFile(file)  
3  double[] alts = convertToDouble(content)  
4  double mean = sum(alts) / count(alts)  
5
```

Dans cet exemple, plusieurs problèmes peuvent survenir :

- le fichier n'existe pas ou sa lecture est impossible,
- le contenu contient des valeurs non numériques impossibles à convertir en double,
- une division par zéro est effectuée (le fichier ne contient peut-être aucune altitude),
- ...

# Les exceptions

- Les langages qui n'ont pas d'exceptions retournent généralement une valeur arbitraire, appelée également valeur passerelle, à la place du résultat.
- Un serveur http retourne un résultat avec un code indiquant l'erreur éventuelle.



# Les exceptions

En supposant qu'il soit possible de retourner un couple contenant le résultat et un code d'erreur, voici comment les anomalies devraient être gérées :

```
1  File file = ???
2  (String content, Status statusRead) = readFile(file)
3  if ( statusRead.isOk() ){
4      (double[] alts, Status statusConvert) = convertToDouble(content)
5      if ( statusConvert.isOk() ){
6          (double mean, Status statusDiv) = sum(alts) / count(alts)
7          if ( statusDiv.isOk() ){
8              /* do something with mean */
9          } else {
10             // oups
11         }
12     } else {
13         // oups
14     }
15 } else {
16     // oups
17 }
```

# Les exceptions

10

C'est dans ce cas où un mécanisme de gestion d'exception est très utile.

Un mécanisme de gestion d'exception apporte un avantage considérable :

**=> il permet de séparer la logique de la gestion des erreurs.**

# Les exceptions

11

Voici, en pseudo-code, une gestion avec exceptions :

```
1  try {
2      /* La logique est décrite dans ce bloc */
3      File f = ???
4      String content = readFile(file)
5      double[] alts = convertToDouble(content)
6      double mean = sum(alts) / count(alts)
7      // do something with mean
8
9      /* le traitement des exceptions est complètement séparé: */
10 } catch (ReadException e) { // oups }
11 } catch (ConvertException e) { // oups }
12 } catch (DivisionException e) { // oups }
13
```

# Les exceptions

12

Lever sa propre exception :

Utiliser le plus possible une exception la plus spécifique possible en fonction du contexte

```
Something getMeSomething(int id) throws NoSuchElementException;
```

Est beaucoup plus intuitif que

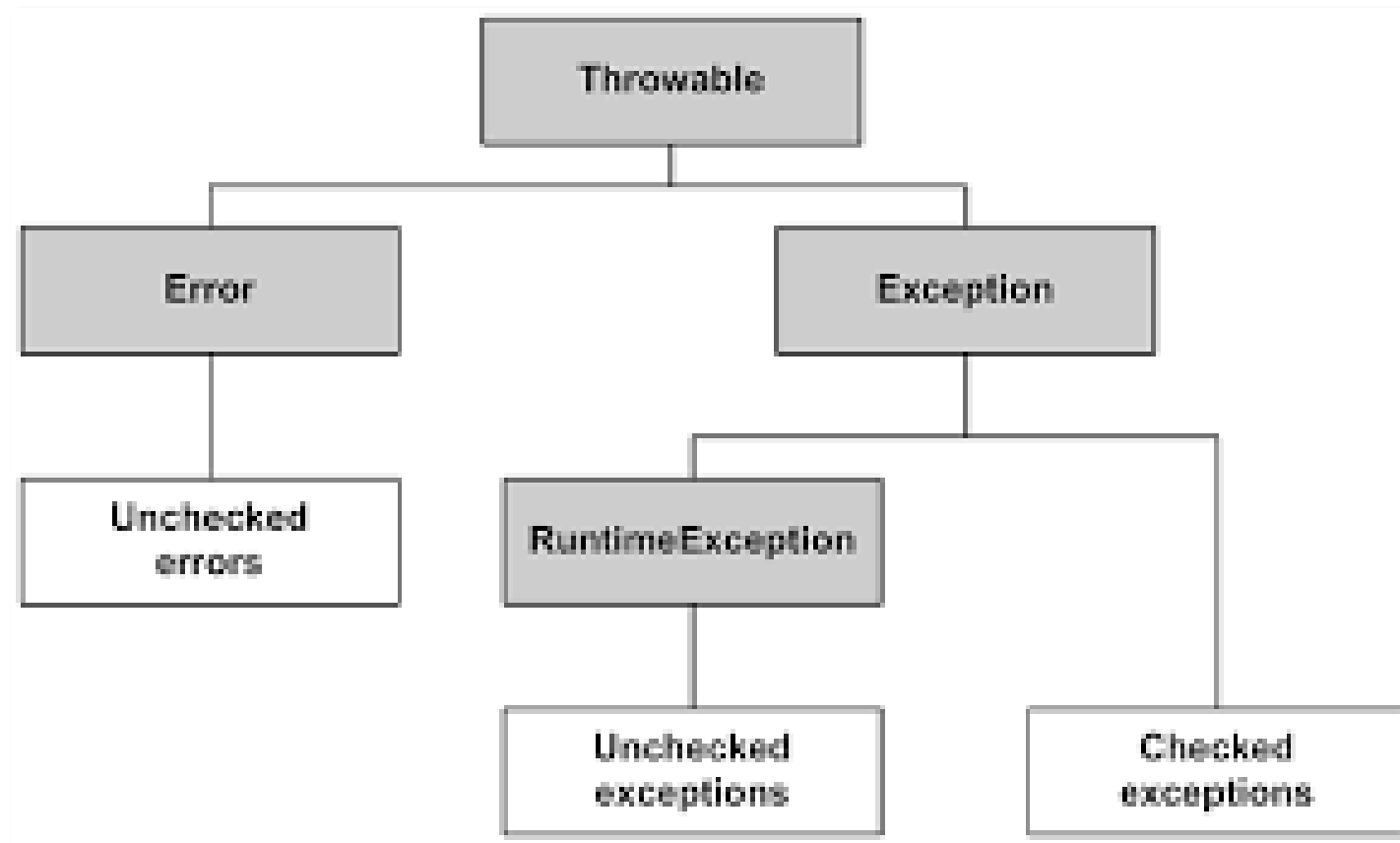
```
Something getMeSomething(int id) throws RuntimeException;
```

# Throwable

13

En Java, une exception est une classe qui hérite de `Throwable`.

Le langage propose une multitude d'exceptions organisées autour de 3 classes principales :



# Les exceptions

14

- `Throwable` : classe de base de `Error` et `Exception`
- `Error` : cette classe et ses sous-classes sont généralement intraitables
  - Le problème est tellement grave que l'arrêt du programme est nécessaire (par ex: `VirtualMachineError`, `StackOverflowError`, `OutOfMemoryError`...)
- `Exception` : toute la hiérarchie des exceptions qui **doivent être traitées**, à l'exception de la branche `RuntimeException`.
  - Exemple d'exceptions qui doivent être traitées obligatoirement (avec un bloc `try/catch` par exemple): `IOException`, `SQLException`, `IllegalClassFormatException`...

# Les exceptions

15

- `RuntimeException` : hérite de `Exception`. Cette classe et ses sous-classes font partie des **unchecked exceptions**.
- Elles n'ont pas besoin d'être traitées. Ce sont les classes comme `NullPointerException`, `ArithmeticException`, `IndexOutOfBoundsException`, `EmptyStackException`...
- Il n'est pas nécessaire non plus de les déclarer dans les méthodes et constructeurs qui pourraient les lever.

# Exceptions

16

Toutes les classes qui héritent d'`Exception`, à l'exception de `RuntimeException`, doivent être **traitées et déclarées**.

La méthode `test` doit déclarer qu'elle retourne une exception.

```
1  void test() throws MandatoryException {  
2      throw new MandatoryException("message");  
3  }
```

L'appelant doit traiter l'exception obligatoirement.



# Exceptions

17

- A l'aide d'un bloc try/catch :

```
1 void useTest() {  
2     try {  
3         test();  
4     } catch ( MandatoryException e) {...}  
5 }
```

- En propageant l'exception plus haut :

```
1 void useTest() throws MandatoryException {  
2     test();  
3 }
```

# Exceptions

18

- Ou en la transformant en une nouvelle exception :

```
1  void useTest() {  
2      try {  
3          test();  
4      } catch ( MandatoryException e) {  
5          throw new OtherException( e.getMessage() );  
6      }  
7  }
```

# Unchecked Exceptions

19

(Runtime Exceptions et enfants)

- Elles n'ont pas besoin d'être traitées.

```
1  int i = 10;
2  int j = 0;
3  System.out.println( i / j );
4  |  Exception java.lang.ArithmeticException: / by zero
5  |      at (#23:1)
6
7  Deque<Integer> q = new LinkedList<>();
8  System.out.println( q.getFirst() );
9  |  Exception java.util.NoSuchElementException
10 |      at LinkedList.getFirst (LinkedList.java:248)
11 |      at (#25:1)
```

# Unchecked Exceptions

20

Il est possible d'éviter de manière traditionnelle ces erreurs :

```
1   int i = 10;
2   int j = 0;
3   if (j != 0) {
4       System.out.println( i / j );
5   }
6
7   Deque<Integer> q = new LinkedList<>();
8   if ( !q.isEmpty() ) {
9       System.out.println( q.getFirst() );
10  }
11
```

Pensez donc à offrir des fonctionnalités qui permettent aux utilisateurs de vérifier l'état d'un objet avant de faire une opération "dangereuse".

# Traitement des exceptions

21

Le traitement se fait à l'aide d'un bloc try/catch :

```
1  try {  
2      /* try something */  
3  } catch (NullPointerException ex) {  
4      System.err.println("Error : " + ex.getMessage());  
5  } catch (NoSuchElementException ex) {  
6      System.err.println("Error : " + ex.getMessage());  
7  }
```

# Traitement des exceptions

22

Si le traitement des exceptions est le même, il est possible de les regrouper :

```
1  try {  
2      /* try something */  
3  } catch (NullPointerException | NoSuchElementException ex) {  
4      System.err.println("Error : " + ex.getMessage());  
5  }
```

# Traitement des exceptions

23

Un bloc `finally` peut être utilisé. C'est le cas pour les ressources qui doivent être fermées dans tous les cas. Voici un exemple pour la lecture de fichier :

```
1  public String readFirstLine() {
2      BufferedReader br = null;
3      try {
4          br = new BufferedReader(new FileReader("Filename.txt"));
5          while ((str = br.readLine()) != null) {
6              System.out.println(str);
7          }
8      } catch (FileNotFoundException f) {
9          System.err.println("File not found"); // affiche l'erreur sur stderr
10     } catch (IOException e) {
11         e.printStackTrace(); // crash en affichant les détails de la stack sur stderr
12     } finally { // dans tous les cas, les ressources doivent être fermées
13         if (br != null) {
14             try { // et oui, une exception peut être levée lors d'un close
15                 br.close();
16             } catch (IOException e) {
17                 e.printStackTrace();
18             }
19         }
20     }
21 }
```

# Traitement des exceptions

24

La gestion est simplifiée depuis Java 8 grâce à l'instruction `try-with-resources`. Cette instruction nous évite de devoir créer un objet `null` au préalable et aussi de devoir fermer les ressources explicitement. Cette instruction fonctionne pour tous les objets qui héritent de l'interface `Autoclosable`

```
1  try ( BufferedReader br = new BufferedReader(new FileReader("Filename.txt")) ) {  
2      while ((str = br.readLine()) != null) {  
3          System.out.println(str);  
4      }  
5  } catch (FileNotFoundException f) {  
6      System.err.println("File not found");  
7  } catch (IOException e) {  
8      e.printStackTrace();  
9  }
```



# Vos propres exceptions

25

Vous pouvez directement hériter votre classe de l'exception la plus adéquate.

```
1  // Unchecked Exception avec message commun
2  public class UncheckedException extends RuntimeException {
3      public UncheckedException() { super("Message specific"); }
4  }
5      /* Exception qui doit être traitée,
6      * dans cet exemple, le message doit être spécifié par l'appelant */
7  public class MandatoryException extends Exception {
8      public MandatoryException(String msg) { super(msg); }
9  }
```

# Emploi d'exceptions existantes

26

Avant de créer vos exceptions, vérifiez toujours si une exception appropriée existe. Sinon, réalisez les vôtres. Rappelez-vous de toujours être spécifique.

Evitez également de traiter un cas trop général.

```
1  try {  
2      return Integer.parseInt( br.readLine() );  
3  } catch (Exception e) {  
4      /* ... */  
5  }
```

Préférez indiquer quel type d'exception vous tentez de traiter.

```
1  try {  
2      return Integer.parseInt( br.readLine() );  
3  } catch (NumberFormatException) { /* ... */ }
```

# Emploi d'exceptions existantes

27

Idem lorsque vous levez des exceptions; Evitez un cas trop général.

```
1  Something getMeSomething(int id) throws RuntimeException;
```

Préférez une exception adaptée. Dans l'extrait ci-dessous, si un élément peut être inexistant par exemple.

```
1  // si l'élément peut être inexistant par exemple
2  Something getMeSomething(int id) throws NoSuchElementException;
3
```

**L'absence de valeurs**

# L'absence de valeurs

29

Les trois techniques les plus courantes pour représenter l'absence de valeurs sont généralement mauvaises. Ce sont :

- Les valeurs passerelles (ou valeurs arbitraires)
- le `null`
- Les exceptions

# Les valeurs passerelles

30

Ceci offre une très mauvaise séparation de la logique et du traitement de l'absence de valeur avec un risque de bugs accru. De plus, il devient très difficile de retrouver la source de cette erreur.

C'est la stratégie proposée par `List<E>` et sa méthode `indexOf` par exemple. Cette stratégie est très utilisée en C également.

méthode	description
<code>int indexOf(Object o)</code>	Returns the index of the first occurrence of the specified element in this list, <b>or -1</b> if this list does not contain the element.

# La référence null

31

Ceci oblige à constamment vérifier qu'une référence n'est pas nulle et provoque très fréquemment l'exception la plus courante : le `NullPointerException`. Il devient également très difficile de retrouver qui est la cause de cette référence nulle.

Selon une étude (source OverOps: <https://blog.takipi.com> ), 97% des exceptions loguées sont causées par 10 exceptions les plus courantes. En tête se trouve le `NullPointerException` loguées dans 70% des 1000 applications étudiées!

Un constat démontre généralement que le `null` est très souvent utilisé à **tort** pour représenter une absence de valeurs.

C'est la stratégie adoptée par `Map<K, V>` et sa méthode `get` par exemple.

méthode	description
<code>V get(Object key)</code>	Returns the value to which the specified key is mapped, <b>or null</b> if this map contains no mapping for the key.

# Les exceptions

32

Cette technique est sensiblement meilleure. Tous les outils sont mis à disposition de l'utilisateur pour vérifier un état avant d'appliquer une action, mais si l'utilisateur ne l'emploie pas correctement, le mieux est peut-être de lever une exception. Le programme peut planter, mais il sera beaucoup plus simple à identifier le code fautif et donc à corriger rapidement celui-ci.

C'est la stratégie adoptée par `Deque<E>` qui s'utilise comme une queue ou comme une file.

méthode	description
<code>public E getFirst()</code>	Retrieves, but does not remove, the first element of this deque. Throws: <code>NoSuchElementException</code> - if this deque is empty.



# L'absence de valeurs dans les collections

33

Pour retourner une liste vide, plusieurs solutions existent :

```
1 // retourne une liste mutable:  
2 return new ArrayList<>();  
3  
4 // retourne une liste non modifiable:  
5 return List.of();  
6 return Collections.emptyList();  
7
```

# L'absence de valeurs dans les collections

34

Vous pouvez donc faire de même dans vos classes :

```
1  public class Test {  
2      /* A éviter sauf si l'instanciation se fait dans les constructeurs  
3      *  
4      * private List<Client> cs; // ici cs référence null  
5      */  
6      private List<Client> cs = new ArrayList<>();  
7      ...  
8  }
```

Vous pouvez donc faire de même pour les tableaux statiques, et d'une manière générale, pour toutes vos collections :

```
1  public int[] emptyArray() {  
2      return {};  
3  }
```

# L'absence de valeurs pour un type particulier (hors collection)

35

L'absence de valeur est une valeur qui existe ou qui n'existe pas. Cette valeur est donc optionnelle.

Si une méthode doit retourner un utilisateur à l'aide d'un identifiant, mais qu'aucun identifiant ne correspond à un utilisateur, plusieurs problèmes peuvent survenir :

```
1  public User get(int id) {  
2      ...  
3      /* que faut-il retourner si l'identifiant ne correspond à aucune utilisateur ? */  
4      return ???;  
5  }
```

```
1  User user = get(42); // retourne null ou lève une exception  
2  System.out.println( user.email() ); // /\ \ NullPointerException si user null  
3
```

## L'absence de valeurs pour un type particulier (hors collection)

36

Pour éviter de retourner `null` ou une exception, une bonne idée consiste à encapsuler l'utilisateur à retourner dans une collection qui ne peut contenir au + un élément.

- Si l'utilisateur demandé existe, la liste retournée contient uniquement celui-ci
- Si l'identifiant ne correspond à aucun utilisateur, une liste vide est retournée.

Nous avons donc deux états possibles ainsi que des méthodes à disposition pour vérifier l'état.

Ainsi, notre méthode deviendrait :

```
1  /* returns a list that contains at most one user */  
2  List<User> get(int id) { ... }
```

## L'absence de valeurs pour un type particulier (hors collection)

37

Son utilisation nous oblige à vérifier si l'utilisateur existe dans la liste :

```
1 List<User> maybeUser = get(42);  
2 if ( !maybeUser.isEmpty() ) {  
3     User user = maybeUser.get(0);  
4     System.out.println( user.email() );  
5 }
```

Il devient même possible d'utiliser le `foreach` :

```
1 get(42).foreach( user -> System.out.println(user.email()) );
```

# L'absence de valeurs pour un type particulier (hors collection)

38

L'approche est intéressante mais comporte des défauts :

- On utilise une structure non adaptée pour représenter un comportement sensiblement différent;
- La valeur optionnelle doit exister ou non, alors qu'une liste peut contenir entre 0 et une infinités d'élément, avec en plus, la possibilité d'ajouter et de supprimer des éléments.

Le type `Optional<T>`

# Le type `Optional<T>`

40

Le type `Optional<T>` est une nouvelle structure qui comporte zéro ou une valeur au plus. Celui-ci agit comme une collection qui peut avoir zéro ou un élément.

La méthode précédente devient :

```
1 public Optional<User> get(int id) { ... }
```

Avec plusieurs utilisations possibles :

```
1 Optional<User> maybeUser = get(42);
2 if( maybeUser.isPresent() ) {
3     User user = maybeUser.get();
4     System.out.println( user.email() );
5 }
```



# Le type Optional<T>

41

Et l'équivalent du `forEach` :

```
1 maybeUser.ifPresent( user -> System.out.println(user.email()) );
```

Pour retourner un optionnel, il existe des fabriques. Imaginons que nous avons utilisé une `Map` (dont la méthode `get` peut retourner `null`) pour stocker des utilisateurs :

```
1 private Map<Integer, User> users = ...;
2
3 public Optional<User> get(int id) {
4     User u = users.get(id);
5     if (u == null) {
6         return Optional.empty();
7     }
8     return Optional.of(u);
9 }
```

## Le type Optional<T>

42

Vous pouvez directement employer la fabrique `ofNullable` pour simplifier le code ci-dessus. Cette méthode retourne un optionnel vide si l'argument est `null`.

```
1  public Optional<User> get(int id) {  
2      return Optional.ofNullable( users.get(id) );  
3  }
```

On a défini au début le ***Fail Fast*** qui consiste à planter le programme le plus rapidement possible tout en offrant une bonne compréhension des raisons de ce plantage.

Le type `Optional` permet de planter encore plus vite : **à la compilation**.

Dans l'exemple suivant, on a une `PendingReservation` qui se transforme en `Reservation` uniquement si elle est planifiée (contrainte imposée par l'argument et vérifiée à la compilation) et si il y a au moins une ressource.

Cette dernière est vérifiée lors de l'exécution et lève une exception le cas échéant :

# Système de typage robuste

44

```
1  class PendingReservation {
2      ...
3
4      public Reservation scheduleAt(LocalDateTime dateTime) {
5          /* à compléter */
6
7          if( ressources().isEmpty() ) {
8              throw new IllegalStateException("A reservation should have at least one resource");
9          }
10
11         /* à compléter */
12     }
13     ...
14 }
```

Une bonne technique pour éviter un problème et une levée d'exception lors de l'exécution est de l'éviter à la compilation :

```
1  class PendingReservation {
2      ...
3      public Optional<Reservation> scheduleAt(LocalDateTime dateTime) {
4          /* à compléter */
5
6          if( ressources().isEmpty() ) {
7              return Optional.empty();
8          }
9          /* à compléter */
10     }
11     ...
12 }
```

Retourner un `Optional` contraint l'utilisateur à gérer cette incohérence. C'est un avantage de plus des fabriques : contrairement à un constructeur qui retourne un objet, une fabrique peut retourner un autre type. Dans notre cas, **un optionnel**.

Un autre avantage des `Optional` est la composition, chose impossible avec des exceptions; Il est possible par exemple, de chaîner les actions :

```
Optional<Email> emailOfToto = getUser(42).map( user -> user.email() );  
// ou un autre exemple  
  
getUser(42).filter( u -> u.age() >= 18 )  
|  
|  
| .map( u -> u.email() )  
|  
| .ifPresent( email -> System.out.println(email) );
```