Exception

Rémi Forax
forax@univ-mlv.fr

Exception

Les exceptions en Java sont un des mécanismes les moins compris par les developpeurs Java

parce que le mécanisme d'exceptions est utilisé pour des choses bien différentes

Pourquoi des exceptions ?

Mécanisme inventé pour indiquer un mode **non-normal** de fonctionnement

Il n'y a pas d'exceptions dans tous les langages

- En C, on utilise la valeur de retour, problème si toutes les valeurs du domaine sont correctes
- En Go, on utilise 2 valeurs de retour,
  cela n'empèche pas les devs d'oublier les valeurs de retour car ils ne lisent pas la doc

C++, Java, Python, Ruby, etc ont des exceptions

Il y a plusieurs types d'exceptions

Les exceptions servent a trois choses bien différentes

- Signaler des **erreurs de programmation**
  - Le developpeur n'a pas lu la doc
  - Le developpeur a du mal avec null, les bornes de tableaux, etc
- Signaler des **erreurs fatales**
  - StackOverflowError, OutOfMemoryError, InternalError
- Signaler des erreurs qui **dépendent de condition externe**
  - Erreurs d'entrée/sortie

Les traitements sont différents

- Erreurs de **programmation**
  - Ne doivent arriver que lors de la phase de dev pas en production
  - On ne doit pas essayer de reprendre sur l'erreur mais de corriger le bug qui fait que l'exception est lancée
- Erreurs **fatales**
  - Peuvent arriver à cause d'une erreur du dev ou d'une erreur de l'ops
  - On ne doit pas reprendre
- Erreurs **externes**
  - Indépendant de l'état d'un programme
  - On doit reprendre sur l'erreur, au moins pour afficher un beau message à l'utilisateur

En Java

Les 3 types d'erreurs ont des classes distinctes

RuntimeException (pas obliger de les traiter)

- Erreur de programmation

Error (pas obliger de les traiter)

- Erreur fatale

Exception (obligation de les traiter)

- Erreur externe

Il y a deux problèmes

- Exception ne représente pas toute les exceptions
- RuntimeException hérite de exception

java.lang.Throwable

java.lang.Throwable est la classe de base de toute les exceptions

comme java.lang.Exception, le compilateur oblige à traiter les exceptions typés java.lang.Throwable

pour une raison inconnue, Throwable n'est pas abstrait ??

Lever une exception

La VM lève des exceptions elle-même,
NPE, AIOOBE, CCE, OutOfMemoryError, etc

La syntaxe **throw** permet de levée une exception

```
throw new IllegalArgumentException("invalid value");
```

L'exception remonte la pile d'exécution jusqu'à être attrapée par une méthode ou par la VM si l'exception sort du main().

StackTrace

Lors de la **création** d'une exception (lors de l'appel au constructor), la VM enregistre la pile d'appel des méthodes jusqu'au new de l'exception

La création d'une exception coûte en temps d'exécution car il faut créer le stack trace

Jeter ou attraper une exception par contre coûte peu chère

- il y a un petit coût lorsque l'on rentre dans un try + un instanceof par catch

Checked Exception

Le compilateur oblige de faire quelque chose pour toutes les sous-classes de Exception qui ne sont pas des RuntimeException

- On appel ces exceptions les **checked** exceptions

Il y a deux facons de **traiter les checked exceptions**

- Déclarer que la méthode lève une exception avec le mot clé throws
- Gérer l'exception avec la syntaxe try-catch

Le mot-clé throws

Permet d'indiquer que la méthode peut lever une checked-exception

```
public static void sayHello(Writer writer)
                                        throws IOException {
  writer.write("hello");
}
public static void main(String[] args) throws IOException {
  sayHello(System.console().writer());
}
```

Si on utilise throws sur une unchecked-exception, le compilateur n'en tient pas compte

La syntaxe try/catch

Permet d'indiquer un code de reprise sur erreur

```
public static void sayHello() throws IOException {
  writer.write("hello");
}
public static void main(String[] args) {
  try {
    sayHello(System.console().writer());
  } catch(IOException e) {
    System.err.println("can't write on console\n" +
          e.getMessage);
    System.exit(1);
  }
}
```

Multiple catch

On peut écrire plusieurs block catch

```
try {
  writeOnHDOrNetwork();
} catch(IOException e) {
  // ...
} catch(NetworkException e) {
  // ...
}
```

Si le code du catch est commun, on peut joindre les deux exceptions (avec un '|')

```
try {
  writeOnHDOrNetwork();
} catch(IOException | NetworkException e) {
  // un seul code commun
}
```

Throws ou try/catch ?

Quand doit-on utiliser throws et quand doit-on utiliser try/catch ?

- Si on peut écrire quelque chose dans le catch pour reprendre sur l'erreur, on écrit un try/catch
- Sinon, on écrit un throws

statistiquement on doit écrire beaucoup plus de throws que de try/catch !

et que pour des checked-exceptions !

try/catch de la mort

La hiérarchie des exceptions est foireuse en Java, RuntimeException hérite de Exception

- Si on fait un catch(Exception), on a peu de chance d'écrire un code qui puisse reprendre sur l'erreur car on ne sait pas si l'erreur est une erreur de prog ou une erreur externe
  - On peut juste faire semblant que tout va bien :)
- Même problème avec catch(Throwable)

Checked-exception et redéfinition

Le compilateur vérifie qu'une méthode redéfinie ne peut pas levée des checked exceptions que la méthode de base ne déclare pas

```
public interface Runnable {
  public void run();
}

public class HelloRunnable implements Runnable {
  public void run() throws IOException {
    // ne compile pas
  }
}
```

Tunneling d'exception

Mais on peut wrapper une exception dans une exception puis la ressortir avec getCause()

```
public class HelloRunnable implements Runnable {
  public void run() {
    try {
      // raise IOException
    } catch(IOException e) {
      throw new UncheckedIOException(e);    // wrap
    }
  }
}
public static void main(String[] args) throws IOException {
  Runnable runnable = new HelloRunnable();
  try {
    runnable.run();
  } catch(UncheckedIOException e) {    // unwrap
     throw e.getCause();
  }
}
```

Prog. défensive & prog. par contrat

blow early, blow often
-- Josh Bloch

Correction de bug

Plus un bug est découvert tard dans le cycle de création d'un programme, plus il coûte chère à être corrigé

=> programmation défensive

Tous arguments passés à une méthode publique doit être validés avant d'être utiliser

=> programmation par contrat

Rôle d'un constructeur

Ne pas faire confiance aux valeurs passées en argument

- On fait plus de lecture sur la valeur d'un champ que d'écriture
- POO: l'état d'un objet doit toujours être valide

donc un constructeur doit vérifier les arguments avant de les stocker dans les champs

Exemple

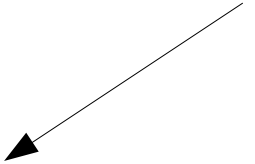
Code qui fait pleurer :(

```
public class Author {
  private final /*maybe null*/ String firstName;
  private final /*maybe null*/ String lastName;

  public Author(String firstName, String lastName) {
    this.firstName = firstName;
    this.lastName = lastName;
  }

  public boolean equals(Object o) {
    if (!(o instanceof Author)) {
      return false;
    }
    Author author = (Author)o;
    return ((author.firstName == null && firstName == null) ||
              author.firstName.equals(firstName)) &&
             (author.lastName == null && lastName == null) ||
              author.lastName.equals(lastName)));
  }
}
```

On fait les check à null à chaque lecture



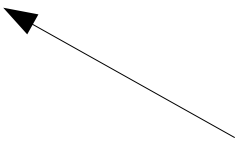
Exemple

Même code écrit normalement

```
public class Author {
  private final String firstName;
  private final String lastName;

  public Author(String firstName, String lastName) {
    this.firstName = Objects.requireNonNull(firstName);
    this.lastName = Objects.requireNonNull(lastName);
  }

  public boolean equals(Object o) {
    if (!(o instanceof Author)) {
      return false;
    }
    Author author = (Author)o;
    return author.firstName.equals(firstName) &&
           author.lastName.equals(lastName);
  }
}
```



On fait les check à null à l'écriture

Programmation par contrat

Toutes méthodes publiques doit documentées son contrat

- Ce quelle fait
- Quels sont les arguments attendus
- Quels sont les valeurs de retour possible
- Quels sont les exceptions qui sont levée et pourquoi sont-elles levée ?

Normalement, les devs doivent lire cette doc, dans la pratique, la doc est lu lorsque le comportement n'est pas celui attendu par le développeur :(

Javadoc

Java possède un format de documentation directement dans la langage

- La doc est toujours à jour par rapport au code

Ne pas confondre un commentaire et un commentaire de doc

- Le commentaire de doc indique comment on se sert de la méthode d'un point de vue utilisateur
- Le commentaire classique qu'il y a quelque chose de pas habituel dans le code
  (ou de pas super lisible)

Exemple

Le code d'une pile

```
public class IntStack {
  private final int[] array;
  private int top;

  public IntStack(int capacity) {
    array = new int[capacity];
  }

  public void push(int value) {
    array[top++] = value;
  }

  public int pop() {
    return array[--top];
  }
}
```

Programmation défensive

```
public class IntStack {
  private final int[] array;
  private int top;

  public IntStack(int capacity) {
    if (capacity < 0) {
      throw new IllegalArgumentException("capacity < 0");
    }
    array = new int[capacity];
  }

  public void push(int value) {
    if (array.length == top) {
      throw new IllegalStateException("stack is full");
    }
    array[top++] = value;
  }

  public int pop() {
    if (top == 0) {
       throw new IllegalStateException("stack is empty");
    }
    return array[--top];
  }
}
```

Programmation par contrat

```
public class IntStack {
  private final int[] array;
  private int top;

  /**
   * Create an integer stack with a maximum capacity.
   * @param capacity the capacity of the stack
   * @throws IllegalArgumentException if the capacity is less than 0
   */
  public IntStack(int capacity) {
    if (capacity < 0) {
      throw new IllegalArgumentException("capacity < 0");
    }
    array = new int[capacity];
  }

  ...
}
```

Programmation par contrat

```
public class IntStack {
  private final int[] array;
  private int top;

  /**
   * Put the value on top of the stack.
   * @param value the value to push on the stack.
   * @throws IllegalStateException if the stack is full
   */
  public void push(int value) {
    if (array.length == top) {
      throw new IllegalStateException("stack is full");
    }
    array[top++] = value;
  }

  ...
}
```

Programmation par contrat

```
public class IntStack {
  private final int[] array;
  private int top;

  /**
   * Extract and return the value on top of the stack.
   * @return the value on top of the stack.
   * @throws IllegalStateException if the stack is empty
   */
  public int pop() {
    if (top == 0) {
       throw new IllegalStateException("stack is empty");
    }
    return array[--top];
  }

  ...
}
```

Prog. par contrat

La programmation défensive consiste à tester les pré-conditions

On peut aussi vouloir tester si le code a bien fait son travail en testant les post-conditions et les invariants

- Post-condition: état de sortie d'un algo
- Invariant: condition toujours vrai pour l'implantation de la classe

La syntaxe assert

La syntaxe assert permet de tester un code lors de son exécution

- assert i == j;
- assert i == j: "message d'erreur";

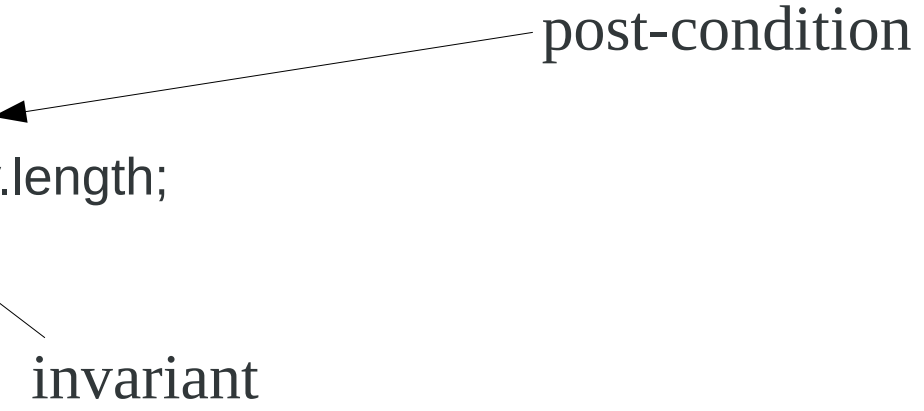
en Java, les assert ne s'exécute qui si on lance le programme avec java -ea
(donc lors du developpement)

Invariants & post-condition

```
public class IntStack {
  private final int[] array;
  private int top;

  /**
   * Put the value on top of the stack.
   * @param value the value to push on the stack.
   * @throws IllegalStateException if the stack is full
   */
  public void push(int value) {
    if (array.length == top) {
      throw new IllegalStateException("stack is full");
    }
    array[top++] = value;
    assert array[top – 1] == value;
    assert top >= 0 && top <= array.length;
  }

  ...
}
```



Prog. par contrat et test unitaire

Les tests unitaires comme les post-conditions et invariant test aussi l'exécution du programme

Post-condition et invariant

- Test avec des données réel à l'intérieur du code

Test unitaire

- Test avec des donées limites à l'extérieur du code

donc il faut les deux !