

Les Exceptions

Rémi Forax

Les exceptions

Les exceptions en Java sont en fait assez compliquées à utiliser et donc très mal utilisées dans beaucoup de code

- A quoi servent les exceptions ?
- Comment doit-on les utiliser ?
- Comment propager correctement des exceptions ?
- Qu'est-ce que le 'design by contract' ?

Exception

Mécanisme qui permet de reporter des erreurs vers la méthode appelante (et la méthode appelante de la méthode appelante, etc.)

Problème en C

- Prévoir une plage de valeurs parmi les valeurs de retour possible pour signaler les erreurs
- Les erreurs doivent être gérées et propagées manuellement

Exemple

```
static class Option {
    Path outputPath;
    boolean all;
    int level;
}

private static Option parseArguments(String[] args) {
    Option options = new Option();
    Map<String, Consumer<Iterator<String>>> actions = Map.of(
        "-output", it -> options.outputPath = Paths.get(it.next()),
        "-all",    it -> options.all = true,
        "-level",  it -> options.level = Integer.parseInt(it.next()));
    for(Iterator<String> it = List.of(args).iterator(); it.hasNext();) {
        actions.get(it.next()).accept(it);
    }
    if (options.outputPath == null) { /* FIXME */ }
    return options;
}

public static void main(String[] args) {
    Option options = parseArguments(args);
    ...
}
```

Ce programme peut planter à 4 endroits
(sans parler du FIXME)

Exemple (2)

Une exception possède

- un type (NoSuchElementException)
- un message d'erreur
- un stacktrace (séquence des méthodes menant à l'erreur)

- \$ java Example -output

```
Exception in thread "main" java.util.NoSuchElementException
at java.util.AbstractList$Itr.next(.../AbstractList.java:377)
at exception.Example.lambda$0(Example.java:20)
at exception.Example.parseArguments(Example.java:24)
at exception.Example.main(Example.java:32)
```

Lire un stacktrace

```
at java.util.ArrayList$Itr.next(.../ArrayList.java:377)
at exception.Example.lambda$0(Example.java:20)
at exception.Example.parseArguments(Example.java:24)
at exception.Example.main(Example.java:32)
```

```
private static Option parseArguments(String[] args) {
    Option options = new Option();
    Map<String, Consumer<Iterator<String>>> actions = Map.of(
        "-output", it -> options.outputPath = Paths.get(it.next()),
        "-all",    it -> options.all = true,
        "-level", it -> options.level = Integer.parseInt(it.next()));
    for(Iterator<String> it = List.of(args).iterator(); it.hasNext();) {
        actions.get(it.next()).accept(it);
    }
    if (options.outputPath == null) { /* FIXME */ }
    return options;
}

public static void main(String[] args) {
    Option options = parseArguments(args);
}
```

Les exceptions et doc

La méthode `AbstractList.Itr.next()` est une méthode du JDK, il faut aller lire la doc !

Ici, la doc de la méthode `Iterator<E>.next()`

E next()

Returns the next element in the iteration.

Returns: the next element in the iteration

Throws: **NoSuchElementException** - if the iteration has no more elements

Jeter une exception

Pour améliorer le message d'erreur, on va changer un peu le code

```
private static Path parseOutputPath(Iterator<String> it) {
    if (!it.hasNext()) {
        throw new IllegalArgumentException(
            "output path requires an argument");
    }
    return Paths.get(it.next());
}

private static Option parseArguments(String[] args) {
    Option options = new Option();
    Map<String, Consumer<Iterator<String>>> actions = Map.of(
        "-output", it -> options.outputPath = parseOutputPath(it),
        "-all",    it -> options.all = true,
        "-level", it -> options.level = Integer.parseInt(it.next()));
    for(Iterator<String> it = List.of(args).iterator(); it.hasNext();) {
        actions.get(it.next()).accept(it);
    }
    if (options.outputPath == null) { /* FIXME */ }
    return options;
}
```


Jeter une exception

L'exception indique alors quel est le vrai problème

```
$ java Example -output  
Exception in thread "main" java.lang.IllegalArgumentException:  
  output path requires an argument  
at exception.Example.parseOutputPath(Example.java:19)  
at exception.Example.lambda$0(Example.java:27)  
at exception.Example.parseArguments(Example.java:31)  
at exception.Example.main(Example.java:39)
```

Mais pour un utilisateur cela reste un bug pas un vrai message d'erreur.

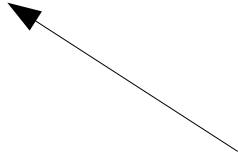
Reprendre sur l'erreur

Il est possible de reprendre sur une erreur en utilisant la syntaxe try/catch, catch() fait un test instanceof sur le type de l'exception

```
private static Path parseOutputPath(Iterator<String> it) {  
    if (!it.hasNext()) {  
        throw new IllegalArgumentException(  
            "output path requires an argument");  
    }  
    return Paths.get(it.next());  
}
```

```
private static Option parseArguments(String[] args) { ... }
```

```
public static void main(String[] args) {  
    Option options;  
    try {  
        options = parseArguments(args);  
    } catch (IllegalArgumentException e) {  
        System.err.println(e.getMessage());  
        return;  
    }  
    ...  
}
```



On a rien besoin de faire dans parseArguments, l'exception suit la pile en sens inverse

Reprendre sur l'erreur

Il est possible de mettre plusieurs blocs catch pour un seul try, les tests instanceof sont fait dans l'ordre de déclaration

```
private static Path parseOutputPath(Iterator<String> it) { ... }
private static Option parseArguments(String[] args) { ... }
public static void main(String[] args) {
    Option options;
    try {
        options = parseArguments(args);
    } catch(InvalidArgumentException e) {
        System.err.println(e.getMessage());
        return;
    } catch(NoSuchElementException e) {
        System.err.println("invalid argument parsing");
        return;
    }
    ...
}
```

Ordre des blocs catch

Comme les catches font des tests instanceof et que les exceptions forment une hiérarchie de classes, il est possible de créer des blocs catch inatteignables

```
private static Path parseOutputPath(Iterator<String> it) { ... }
private static Option parseArguments(String[] args) { ... }
public static void main(String[] args) {
    Option options;
    try {
        options = parseArguments(args);
    } catch(RuntimeException e) {
        System.err.println(e.getMessage());
        return;
    } catch(NoSuchElementException e) {
        System.err.println("invalid argument parsing");
        return;
    }
    ...
}
```

Comme NSEE hérite de RuntimeException, le compilateur plante !

Multi catch

Si la façon de reprendre sur l'erreur est la même, il est possible de séparer le type des exceptions par un ou '|' dans le catch

```
private static Path parseOutputPath(Iterator<String> it) { ... }
private static Option parseArguments(String[] args) { ... }
public static void main(String[] args) {
    Option options;
    try {
        options = parseArguments(args);
    } catch (IllegalArgumentException e) {
        System.err.println(e.getMessage());
        return;
    } catch (NoSuchElementException | NumberFormatException e) {
        System.err.println("invalid argument parsing");
        return;
    }
    ...
}
```

Créer sa propre exception

On peut vouloir différencier les exceptions lever par les méthodes de l'API avec les exceptions liés à un problème métier spécifique (ici, le parsing des options)

```
public class OptionParsingException
    extends RuntimeException ← Cf plus tard
{
    public OptionParsingException(String message) {
        super(message);
    }

    public OptionParsingException(Throwable cause) {
        super(cause);
    }

    public OptionParsingException(String message, Throwable cause) {
        super(message, cause);
    }
}
```

La spécification demande d'implanter les 3 constructeurs !

Chainer les exceptions

Les constructeurs qui prennent une **cause** en paramètre permettent de debugger plus facilement le code en indiquant l'exception qui a généré le problème

```
private static int parseLevel(Iterator<String> it) {
    if (!it.hasNext()) {
        throw new OptionParsingException(
            "level requires an argument");
    }
    try {
        return Integer.parseInt(it.next());
    } catch (NumberFormatException cause) {
        throw new OptionParsingException(
            "level argument is not an integer", cause);
    }
}
```

Les exceptions sont alors chaînées

Chainer les exceptions

Les deux exceptions figurent dans le stacktrace

```
$ java Example -level foo
```

```
Exception in thread "main" exception.OptionParsingException: level  
argument is not an integer
```

```
at exception.Example.parseLevel(Example.java:31)
```

```
at exception.Example.lambda$2(Example.java:40)
```

```
at exception.Example.parseArguments(Example.java:42)
```

```
at exception.Example.main(Example.java:51)
```

```
Caused by: java.lang.NumberFormatException: For input string: "foo"
```

```
at java.lang.NumberFormatException.forInputString(...:65)
```

```
at java.lang.Integer.parseInt(java.base@9-ea/Integer.java:695)
```

```
at java.lang.Integer.parseInt(java.base@9-ea/Integer.java:813)
```

```
at exception.Example.parseLevel(Example.java:29)
```

```
... 3 more
```


initCause(Throwable cause)

initCause() permet de spécifier une cause dans le cas où il n'existe pas de constructeur prenant une cause en paramètre

```
private static int parseLevel(Iterator<String> it) {  
    ...  
    try {  
        return Integer.parseInt(it.next());  
    } catch (NumberFormatException cause) {  
        OptionParsingException e = new OptionParsingException(  
            "level argument is not an integer");  
        e.initCause(cause);  
        throw e;  
    }  
}
```

initCause() ne peut être appelée qu'une fois

Exceptions et le Compilateur

Checked Exception

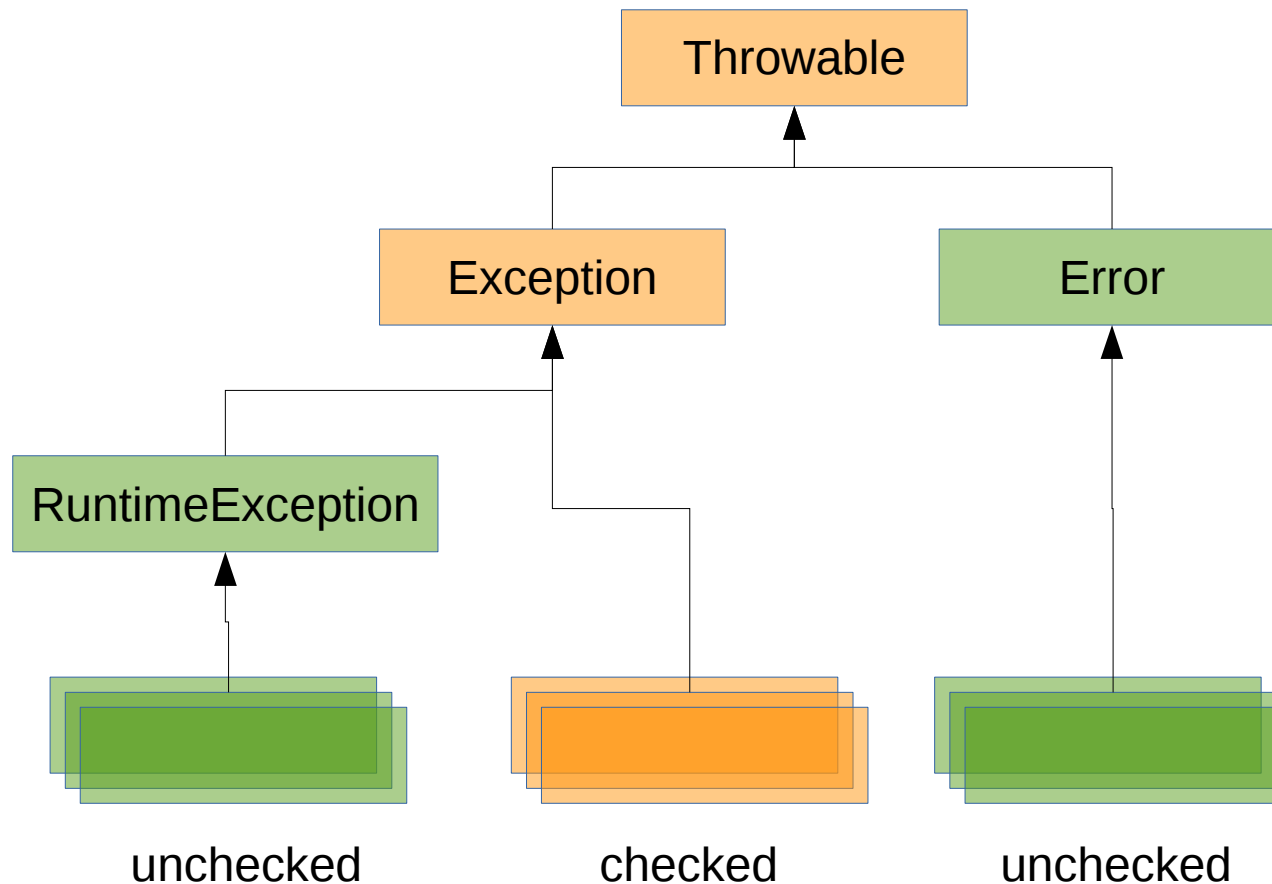
Pour le compilateur, il n'existe deux sortes d'exceptions

- Les exceptions que l'on doit vérifier (*checked exceptions*) qui nécessite l'utilisation de la clause **throws**
- Les exceptions qui ne nécessite pas de vérification (*unchecked exceptions*)

Les classes héritant de **Throwable** sont à vérifier, sauf celles héritant de **RuntimeException** et **Error**

Hierarchie des Throwable

Il faut hériter de Throwable pour pouvoir être lancer par un throw



Usages des Throwable

Exception

- Erreurs dus à des conditions externes, il faut reprendre sur ces erreurs pour avoir un programme stable
 - IOException, InterruptedException

Error

- Erreurs qu'il ne faut jamais attraper
 - OutOfMemoryError, InternalError

RuntimeException

- Les autres, erreur de prog + erreur métier

catch(Exception)

Ahhhhhhhhhhhhhhhhhhhh,
dans ce cas on attrape, les erreurs de prog., les
erreurs métiers et les erreurs d'I/O.

- Aucune chance d'écrire un code correcte dans le catch !

On écrit pas un code qui fait un
catch(RuntimeException), catch(Error),
catch(Exception) ou catch(Throwable)

- Si l'on veut gérer plusieurs exceptions différentes, on utilise soit le muti-catch, soit on crée une sous-classe !

La directive throws

Une exception checkée doit être soit déclarée avec la directive throws pour être propagé

```
private static long parseTime(Iterator<String> it) throws IOException {  
    if (!it.hasNext()) {  
        Path current = Paths.get(".");  
        return Files.getLastModifiedTime(current).toMillis();  
    }  
    ...  
}
```

Files.getLastModifiedTime déclare qu'elle throws IOException, il faut déclarer que l'on propage l'exception dans la signature de parseTime.

Methodes qui throws

Mais les méthodes qui throws une exception (checkée) ne marche pas bien avec le reste du code

```
private static long parseTime(Iterator<String> it) throws IOException {
    if (!it.hasNext()) {
        Path current = Paths.get(".");
        return Files.getLastModifiedTime(current).toMillis();
    }
    ...
}

private static Option parseArguments(String[] args) {
    Option options = new Option();
    Map<String, Consumer<Iterator<String>>> actions = Map.of(
        "-time", it -> options.time = parseTime(it),
        ...);
    ...
    return options;
}
```

Le code ne compile pas ? Pourquoi ?

Methodes qui throws (2)

La lambda, `it → options.time = parseTime(it)`, doit être converti vers un `Consumer<...>` hors la méthode `accept()` d'un `Consumer` ne déclare pas propager d'`IOException`

```
private static long parseTime(Iterator<String> it) throws IOException {
    ...
}
private static Option parseArguments(String[] args) {
    Option options = new Option();
    Map<String, Consumer<Iterator<String>>> actions = Map.of(
        "-time", it -> options.time = parseTime(it),
        ...);
    for(Iterator<String> it = List.of(args).iterator(); it.hasNext();) {
        Consumer<Iterator<String>> consumer = actions.get(it.next());
        consumer.accept(it);
    }
    ...
    return options;
}
```

Comment résoudre le problème (à part modifier `java.util.function.Consumer`) ?

Tunneling d'exception

On met l'exception checké dans une exception non checké et on récupère la cause de l'exception non checké dans le catch.

```
private static long parseTime(Iterator<String> it) throws IOException {
    try {
        ...
    } catch(IOException e) {
        throw new UncheckedIOException(e);
    }
}

private static Option parseArguments(String[] args) throws IOException {
    Option options = new Option();
    Map<String, Consumer<Iterator<String>>> actions = Map.of(
        "-time", it -> options.time = parseTime(it),
        ...);
    for(Iterator<String> it = List.of(args).iterator(); it.hasNext();) {
        Consumer<Iterator<String>> consumer = actions.get(it.next());
        try {
            consumer.accept(it);
        } catch(UncheckedIOException e) {
            throw e.getCause();
        }
    }
    ...
    return options;
}
```

UncheckedIOException vs IOError

Certaines méthodes du JDK comme `Files.lines()`, `Files.list()` ou `Files.walk()` lève une `IOException` au lieu de `UncheckedIOException`

`IOException` est une erreur et pas une `Exception`, elle n'est donc pas attrapé par un `catch(Exception)`

Throws ou catch

Si on ne peut pas reprendre sur l'erreur à cette endroit, on écrit un throws

Si on peut reprendre sur l'erreur, on écrit un catch

Dans le run() d'un Runnable ou le main, on doit écrire un catch

On essaye de reprendre sur l'erreur le plus bas possible dans la pile d'exécution pour éviter de dupliquer le code des catches

Exceptions et entrée/sortie

Le bloc finally

Sert à exécuter du code quoi qu'il arrive
(fermer un fichier, une connection, libérer une ressource)

```
BufferedReader reader =  
    Files.newBufferedReader(path);  
try {  
    doSomething(reader);  
} finally {  
    reader.close();  
}
```

Le bloc catch n'est pas nécessaire

Le try-with-resources

Le code précédent ne marche pas bien car `close()` peut aussi lever une exception qui peut masquer l'exception lancer dans le try

```
try(BufferedReader reader =  
    Files.newBufferedReader(path)) {  
    doSomething(reader);  
}
```

L'appel à `close` est fait implicitement à la sortie du bloc

Si `close()` lève une exception elle est stocké en tant que `suppressed exception` (cf `getSuppressed()`) de l'exception levée dans le bloc try

Le try-with-resources (2)

On peut initialiser plusieurs variables dans le try(...) dans ce cas, les close() sont appelés dans l'ordre inverse des déclarations

```
try(BufferedReader reader =  
    Files.newBufferedReader(input);  
    BufferedWriter writer =  
        Files.newBufferedWriter(output)) {  
    doSomething(reader, writer);  
} // appel writer.close() puis reader.close()
```

Les initialisation des ressources dans le try(...) sont séparées par des ‘;’

Programmation par contrat

Design by contract

L'idée est que chaque méthode publique fournie un contrat d'utilisation (sa javadoc) et que l'implantation va vérifier

- Les préconditions
 - Propriété des arguments (non-null, positif, etc)
 - Propriété de l'objet (close() n'a pas été appelée)
- Les post-condition
 - Propriété de la valeur de retour (non-null, positif, etc)
 - Propriété de l'objet après appel de la méthode
- Les invariants (de la classe)
 - Propriété de l'objet qui sont toujours vrai

Design by contract

Contrairement au test unitaire qui test si le code fait bien ce qu'il faut en appelant le code

La programmation par contrat permet de détecter les codes d'appels invalides (précondition) en plus d'une mauvaise implantation (postcondition et invariant) en ajoutant les tests directement dans le code

Dans la pratique, on n'écrit souvent que les préconditions, le reste étant attrapé par les tests unitaires

Exemple de contrat

```
public class Stack {  
    /**  
     * Create a stack with a given capacity  
     * @param capacity the stack capacity  
     * @throws IllegalArgumentException is capacity is negative or null  
     */  
    public Stack(int capacity) { ... }  
  
    /**  
     * Push an object on top of the stack  
     * @param object an object  
     * @throws NullPointerException if the object is null  
     * @throws IllegalStateException if the stack is full  
     */  
    public void push(Object object) { ... }  
  
    /**  
     * Remove and return the object on top of the stack  
     * @return the object on top of the stack  
     * @throws IllegalStateException if the stack is empty  
     */  
    public Object pop() { ... }  
}
```

Exemple de code

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

    public Stack(int capacity) {
        if (capacity <= 0) {
            throw new IllegalArgumentException("capacity is negative");
        }
        array = new Object[capacity];
    }

    public void push(Object object) {
        if (object == null) {
            throw new NullPointerException();
        }
        if (top >= array.length) {
            throw new IllegalStateException("stack is full");
        }
        array[top++] = object;
    }

    public Object pop() {
        if (top <= 0) {
            throw new IllegalStateException("stack is empty");
        }
        Object object = array[top];
        array[top--] = null; // GC !
        return object;
    }
}
```

Exemple de code (2)

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

    public Stack(int capacity) {
        if (capacity <= 0) {
            throw new IllegalArgumentException("capacity is negative");
        }
        array = new Object[capacity];
    }

    public void push(Object object) {
        Objects.requireNonNull(object);
        Objects.checkIndex(top, array.length);
        array[top++] = object;
    }

    public Object pop() {
        if (top <= 0) {
            throw new IllegalStateException("stack is empty");
        }
        Object object = array[top];
        array[top--] = null; // GC !
        return object;
    }
}
```

java.util.Objects possède des méthodes prédéfinies pour écrire des préconditions



Exception et performance

Performance et Exception

En terme de performance, **créer** une exception coûte cher car il faut remplir le stacktrace

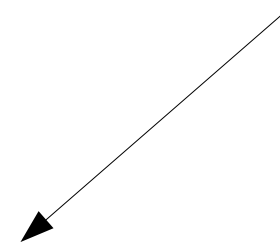
- Donc parcourir la pile à l'envers
- Et allouer les StackTraceElement qui vont bien

Faire un throw coûte pas grand chose

On utilise pas des Exceptions dans le flow normal d'exécution

```
int[] array = ...
try {
    for(int i = 0; ; i++) {
        // faire un truc avec array[i]
    }
} catch(ArrayIndexOutOfBoundsException e) {
    // on est arrivé à la fin du tableau, wink, wink
}
```

Par défaut, le JIT ne compile pas le code des blocs catch



Exception sans stacktrace

Il est possible de créer des exceptions sans stacktrace

```
public class StackLessException extends RuntimeException {  
    private StackLessException() {  
        super(null, null,  
            /*enableSuppression*/ false,  
            /*writableStackTrace*/ false);  
    }  
  
    public static final StackLessException GLOBAL =  
        new StackLessException();  
  
    public static void main(String[] args) {  
        throw GLOBAL; // affiche juste le type de l'exception  
    } // pas pratique pour débbugger  
} // mais pas d'allocation !
```